DeepLearning.ai

深度学习课程笔记（V5.1）

你不是一个人在战斗！

摘要

本文档是针对吴恩达老师深度学习课程(deeplearning.ai)视频做的笔记

主编：黄海广

haiguang2000@qq.com

机器学习爱好者 qq 群：554839127

www.ai-start.com

最后修改：2017-12-20

Coursera 深度学习教程中文笔记

课程概述

这些课程专为已有一定基础（基本的编程知识，熟悉 Python、对机器学习有基本了解），想要尝试进入人工智能领域的计算机专业人士准备。介绍显示：“深度学习是科技业最热门的技能之一，本课程将帮你掌握深度学习。”

在这 5 堂课中，学生将可以学习到深度学习的基础，学会构建神经网络，并用在包括吴恩达本人在内的多位业界顶尖专家指导下创建自己的机器学习项目。Deep Learning Specialization 对卷积神经网络 (CNN)、递归神经网络 (RNN)、长短期记忆 (LSTM) 等深度学习常用的网络结构、工具和知识都有涉及。

课程中也会有很多实操项目，帮助学生更好地应用自己学到的深度学习技术，解决真实世界问题。这些项目将涵盖医疗、自动驾驶、和自然语言处理等时髦领域，以及音乐生成等等。Coursera 上有一些特定方向和知识的资料，但一直没有比较全面、深入浅出的深度学习课程——《深度学习专业》的推出补上了这一空缺。

课程的语言是 Python，使用的框架是 Google 开源的 TensorFlow。最吸引人之处在于，课程导师就是吴恩达本人，两名助教均来自斯坦福计算机系。完成课程所需时间根据不同的学习进度，大约需要 3-4 个月左右。学生结课后，Coursera 将授予他们 Deep Learning Specialization 结业证书。

“我们将帮助你掌握深度学习，理解如何应用深度学习，在人工智能业界开启你的职业生涯。”吴恩达在课程页面中提到。

本人黄海广博士，以前写过吴恩达老师的机器学习个人笔记。有朋友报名了课程，下载了这次课程的视频给大家分享。Coursera 的字幕不全，同学们在学习上感觉非常不方便，因此我找志同道合的朋友翻译和整理字幕，中英文字幕来自于由我和曹骁威同学组织爱好者翻译，希望对大家有所帮助。（**备注：自网易公开课翻译深度学习课程后，我们不再翻译**）

目前我正在组织团队整理中文笔记，由热心的朋友无偿帮忙制作整理，并持续更新。我们的团队的劳动致力于 AI 在国内的推广，不会损害 Coursera 以及吴恩达老师的商业利益。

本人水平有限，如有公式、算法错误，请及时指出，发邮件给我，也可以加我 qq。

黄海广

2017-08-15

吴恩达的公开信

朋友们，

我在做三个全新的 AI 项目。现在，我十分兴奋地宣布其中的第一个：deeplearning.ai，一个立志于扩散 AI 知识的项目。该项目在 Coursera 上发布了一系列深度学习课程，这些课程将帮助你掌握深度学习、对它高效地应用，并打造属于你自己的 AI 事业。

AI 是新一轮电力革命

就像一百年前电力改造了每个主流行业，当今的 AI 技术在做着相同的事。好几个大型科技公司都设立了 AI 部门，用 AI 革新他们的业务。接下来的几年里，各个行业、规模大小各不相同的公司也都会意识到-----在由 AI 驱动的未来，他们必须成为其中的一份子。

创建由 AI 驱动的社会

我希望，我们可以建立一个由 AI 驱动的社会：让每个人看得起病，给每个孩子个性化的教育，让所有人都能坐上价格亲民的自动驾驶汽车，并向男人和女人提供有意义的工作。总而言之，是一个让每个人的生活变得更好的社会。

但是，任何一个公司都不可能单独完成这些任务。就像现在每一个计算机专业的毕业生都知道怎么用云，将来，每个程序员也必须懂得怎么用 AI。用深度学习改善人类生活的方法有数百万种，社会也需要数百万个人----即来自世界各国的你们，来创造出了不起的 AI 系统。不管你是加州的一个软件工程师，一名中国的研究员，还是印度的 ML 工程师，我希望都能用深度学习来解决世界上的各种挑战。

你会学到什么

任何一个掌握了机器学习基础知识的人，都可以学习这五门系列课程，它们组成了 Coursera 的全新深度学习专业。

你会学到深度学习的基础，理解如何创建神经网络，学习怎么成功地领导机器学习项目。你会学习卷积神经网络、 RNNs、LSTM、Adam、Dropout、BatchNorm、Xavier/He initialization 以及更多。学习过程中，你会接触到医疗、自动驾驶、读手语、音乐生成、自然语言处理的案例。

你不仅会掌握深度学习理论，还会看到它是怎样在行业应用落地的。你会在 Python 和 TensorFlow 里试验这些想法，你还会听到各位深度学习领袖人物的意见，他们会分享

各自的学习经历，并提供职业规划建议。

当你拿到 Coursera 的深度学习专业证书，就可以自信得把“深度学习”四个字写进你的简历。

加入我，建立一个由 AI 驱动的社会

从 2011 年到现在，已经有 180 万人加入了我的机器学习课程。当时，我和四名斯坦福的学生发布了这门课程，它随即成为了 Coursera 的第一门公开课。那之后，我受到你们之中许多人的启发----当我看到你们是如何努力地理解机器学习，开发优秀的 AI 系统，并开启令人惊艳的事业。

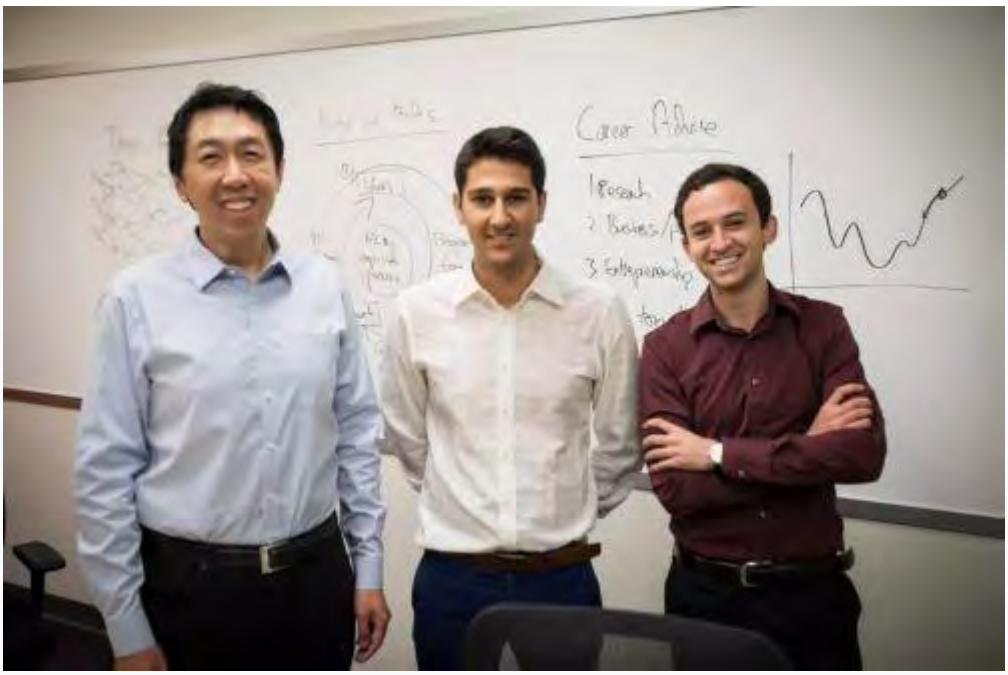
我希望深度学习专业能帮助你们实现更了不起的事，让你们为社会贡献更多，在职业道路上走得更远。

我希望大家和我一道，建立一个由 AI 驱动的社会。

我会通知大家另外两个项目的进展，并不断探索，为全世界 AI 社区的每一个人提供更多支持的途径。

Sincerely，

吴恩达



吴恩达与 deeplearning.ai 团队

■ 文档修改历史

版本号	版本日期	修改总结	修订人
1.0	2017.8.15	创建初稿	黄海广
1.1	2017.8.18	完成第一章	黄海广
1.2	2017.8.28	完成第二章	黄海广
1.3	2017.9.11	完成第三章	黄海广
1.4	2017.9.11	完成第四章，第一门课已完成	黄海广
1.5	2017.10.09	完成第二、三门课初稿	黄海广
3.0	2017.10.11	完成第二、三门课初稿	黄海广
3.1	2017.10.19	第二、三门课补充	黄海广
3.2	2017.11.11	增加人工智能大师访谈	黄海广
4.0	2017.11.22	第四门课更新	黄海广
4.1	2017.11.24	完成第四门课初稿	黄海广
5.0	2017.12.10	完成前四课，并补充前三课	黄海广
5.1	2017.12.20	前四课修改完成，笔记更名为 DeepLearning.ai 深度学习课程笔记	黄海广

主要编写人员：	**黄海广、王翔（3）、胡瀚文:（2）、 余笑:（2）、 郑浩:（2）、李怀松:（1）、 朱越鹏:（2）、陈伟贺:（1）、 曹越:（1）、 路皓翔:（1）、邱牧宸:（1）、 唐天泽:（2）、 张浩:（3）、 陈志豪:（1）、 游忍:（1）、 泽霖:（1）、沈伟臣:（1）、 贾红顺:（2）、 时超:（1）、 陈哲:（2）、赵一帆:（2）、 胡潇杨:（1）、祝彦森:（第三课所有底稿）、 段希:（1）、于冲:（1）、张鑫倩（6）、林木（第四课所有底稿）**
主要编辑人员：	黄海广 陈康凯 Taurus_Moon Arvinky EdwinXiang shichen 王翔 uirboyan Yanfenglong **LeoTsui** 贺志尧 段希 陈瑶

视频教程地址：https://mooc.study.163.com/university/deeplearning_ai#/c

笔记公布网站：**www.ai-start.com**

此文档免费，请不要用于商业用途，可以自由传播。

赠人玫瑰，手有余香！

目录

第一门课 神经网络和深度学习(Neural Networks and Deep Learning)

一、 第一周：深度学习引言(Introduction to Deep Learning)

1.1 欢迎(Welcome)

第一个视频主要讲了什么是深度学习，深度学习能做些什么事情。以下是吴恩达老师的原话：

深度学习改变了传统互联网业务，例如如网络搜索和广告。但是深度学习同时也使得许多新产品和企业以很多方式帮助人们，从获得更好的健康关注。

深度学习做的非常好的一个方面就是读取 X 光图像，到生活中的个性化教育，到精准化农业，甚至到驾驶汽车以及其它一些方面。如果你想要学习深度学习的这些工具，并应用它们来做这些令人窒息的操作，本课程将帮助你做到这一点。当你完成 **cousera** 上面的这一系列专项课程，你将能更加自信的继续深度学习之路。在接下来的十年中，我认为我们所有人都有机会创造一个惊人的世界和社会，这就是 **AI**（人工智能）的力量。我希望你们能在创建 **AI**（人工智能）社会的过程中发挥重要作用。

我认为 AI 是最新的电力，大约在一百年前，我们社会的电气化改变了每个主要行业，从交通运输行业到制造业、医疗保健、通讯等方面，我认为如今我们见到了 **AI** 明显的令人惊讶的能量，带来了同样巨大的转变。显然，AI 的各个分支中，发展的最为迅速的就是深度学习。因此现在，深度学习是在科技世界中广受欢迎的一种技巧。

通过这个课程，以及这门课程后面的几门课程，你将获取并且掌握那些技能。

下面是你将学习到的内容：

在 **cousera** 的这一系列也叫做专项课程中,在第一门课中（**神经网络和深度学习**），你将学习神经网络的基础，你将学习神经网络和深度学习，这门课将持续四周，专项课程中的每门课将持续 2 至 4 周。

但是在第一门课程中，你将学习如何建立神经网络（包含一个深度神经网络），以及如何在数据上面训练他们。在这门课程的结尾，你将用一个深度神经网络进行辨认猫。

由于某种原因，第一门课会以猫作为对象识别。

接下来在第二门课中，我们将使用三周时间。你将进行深度学习方面的实践，学习严密地构建神经网络，如何真正让它表现良好，因此你将要学习超参数调整、正则化、诊断偏差和方差以及一些高级优化算法，比如 **Momentum** 和 **Adam** 算法，犹如黑魔法一样根据你建立网络的方式。第二门课只有三周学习时间。

在第三门课中，我们将使用两周时间来学习如何结构化你的机器学习工程。事实证明，构建机器学习系统的策略改变了深度学习的错误。

举个例子：你分割数据的方式，分割成训练集、比较集或改变的验证集，以及测试集合，改变了深度学习的错误。

所以最好的实践方式是什么呢？

你的训练集和测试集来自不同的贡献度在深度学习中的影响很大，那么你应该怎么处理呢？

如果你听说过端对端深度学习，你也会在第三门课中了解到更多，进而了解到你是否需要使用它，第三课的资料是相对比较独特的，我将和你分享。我们了解到的所有的热门领域的建立并且改良许多的深度学习问题。这些当今热门的资料，绝大部分大学在他们的深度学习课堂上面里面不会教的，我认为它会提供你帮助，让深度学习系统工作的更好。

在第四门课程中，我们将会提到卷积神经网络(**CNN(s)**)，它经常被用于图像领域，你将

会在第四门课程中学到如何搭建这样的模型。

最后在第五门课中，你将会学习到序列模型，以及如何将它们应用于自然语言处理，以及其它问题。

序列模型包括的模型有循环神经网络（**RNN**）、全称是长短期记忆网络（**LSTM**）。你将在课程五中了解其中的时期是什么含义，并且有能力应用到自然语言处理（**NLP**）问题。

总之你将在课程五中学习这些模型，以及能够将它们应用于序列数据。比如说，自然语言就是一个单词序列。你也将能够理解这些模型如何应用到语音识别或者是编曲以及其它问题。

因此，通过这些课程，你将学习深度学习的这些工具，你将能够去使用它们去做一些神奇的事情，并借此来提升你的职业生涯。

吴恩达

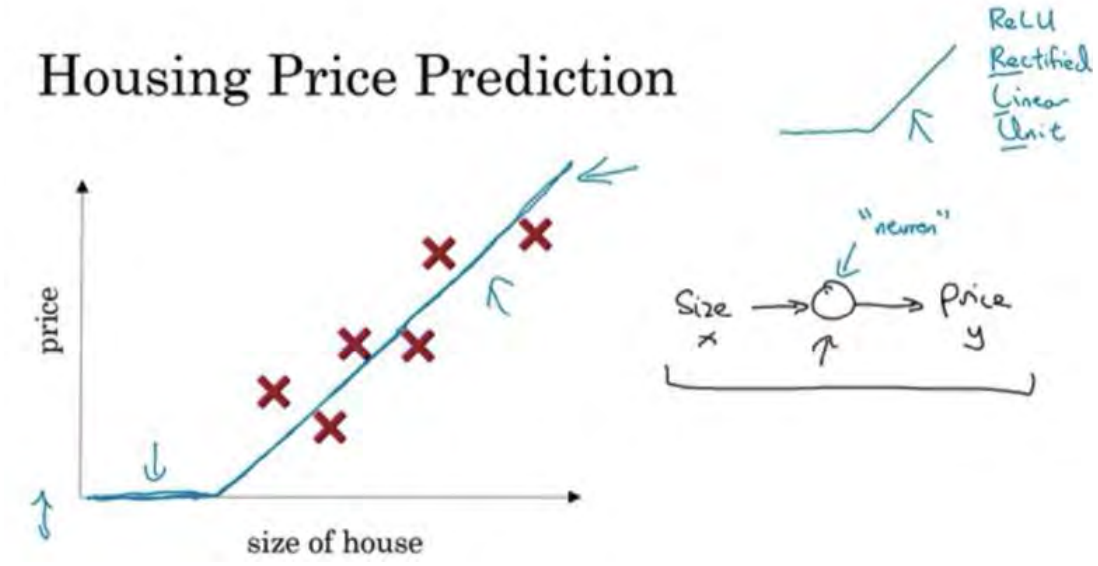
1.2 什么是神经网络？(What is a Neural Network)

我们常常用深度学习这个术语来指训练神经网络的过程。有时它指的是特别大规模的神经网络训练。那么神经网络究竟是什么呢？在这个视频中，我会讲解一些直观的基础知识。

让我们从一个房价预测的例子开始讲起。

假设你有一个数据集，它包含了六栋房子的信息。所以，你知道房屋的面积是多少平方英尺或者平方米，并且知道房屋价格。这时，你想要拟合一个根据房屋面积预测房价的函数。

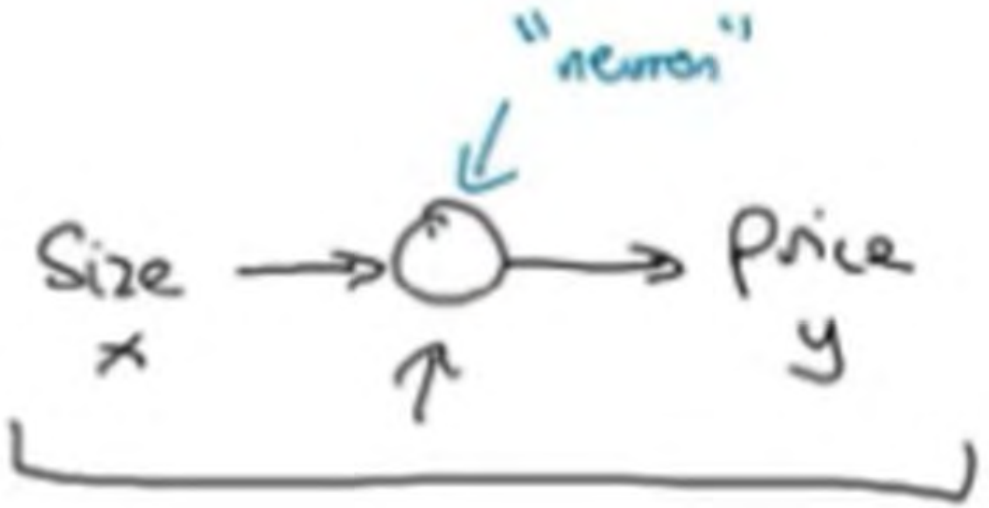
如果你对线性回归很熟悉，你可能会说：“好吧，让我们用这些数据拟合一条直线。”于是你可能会得到这样一条直线。



但有点奇怪的是，你可能也发现了，我们知道价格永远不会是负数的。因此，为了替代一条可能会让价格为负的直线，我们把直线弯曲一点，让它最终在零结束。这条粗的蓝线最终就是你的函数，用于根据房屋面积预测价格。有部分是零，而直线的部分拟合的很好。你也许认为这个函数只拟合房屋价格。

作为一个神经网络，这几乎可能是最简单的神经网络。我们把房屋的面积作为神经网络的输入（我们称之为x），通过一个节点（一个小圆圈），最终输出了价格（我们用y表示）。其实这个小圆圈就是一个单独的神经元。接着你的网络实现了左边这个函数的功能。

在有关神经网络的文献中，你经常看得到这个函数。从趋近于零开始，然后变成一条直线。这个函数被称作 **ReLU** 激活函数，它的全称是 **Rectified Linear Unit**。rectify（修正）可以理解成$max(0,x)$，这也是你得到一个这种形状的函数的原因。

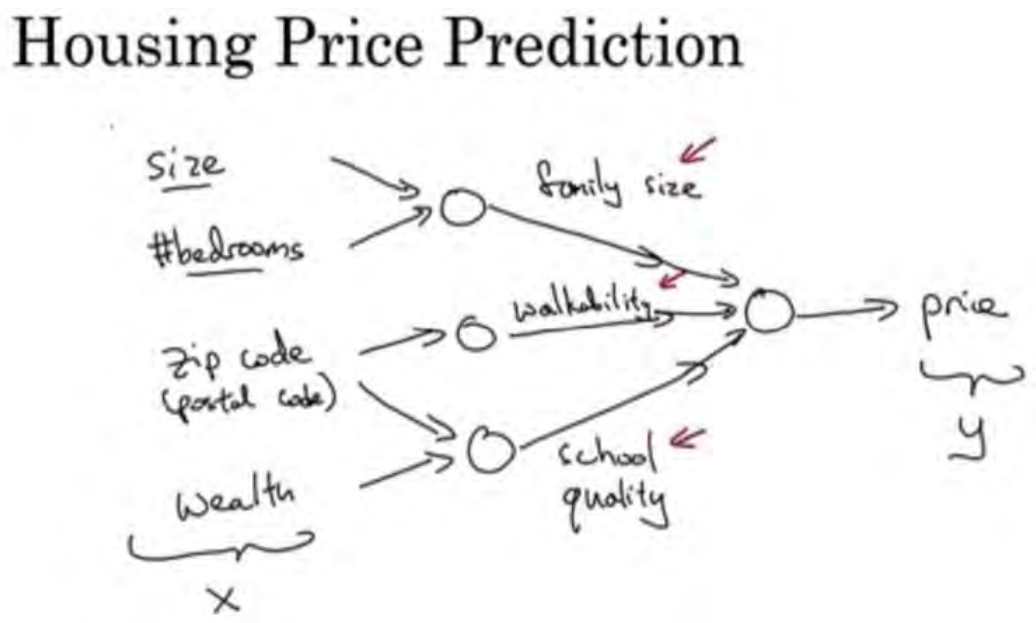


你现在不用担心不理解 **ReLU** 函数，你将会在这门课的后面再次看到它。

如果这是一个单神经元网络，不管规模大小，它正是通过把这些单个神经元叠加在一起来形成。如果你把这些神经元想象成单独的乐高积木，你就通过搭积木来完成一个更大的神经网络。

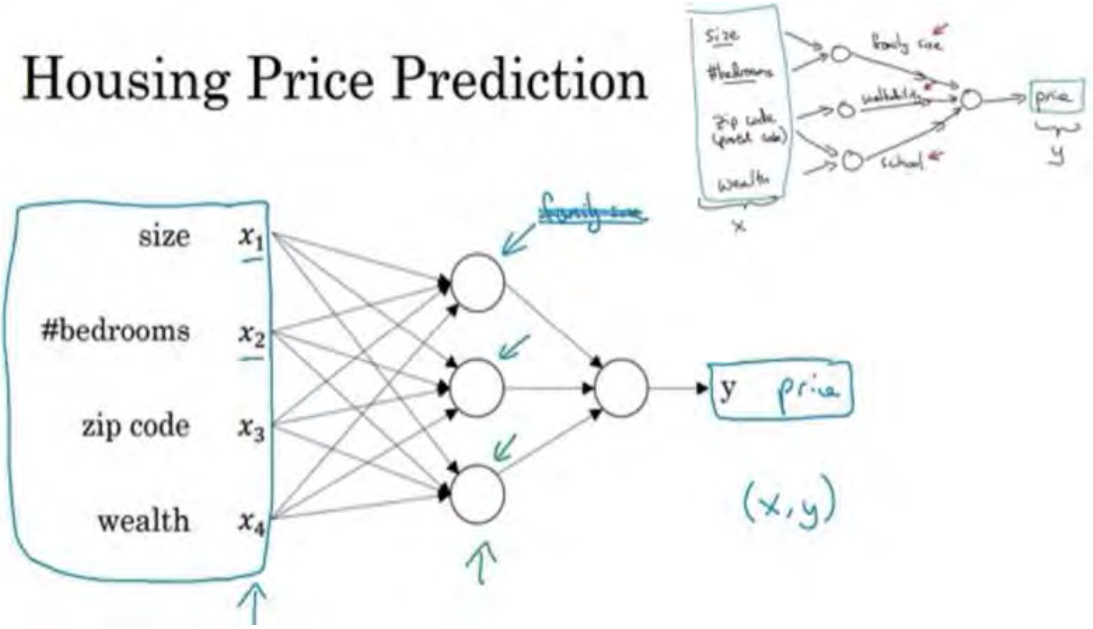
让我们来看一个例子，我们不仅仅用房屋的面积来预测它的价格，现在你有了一些有关房屋的其它特征，比如卧室的数量，或许有一个很重要的因素，一家人的数量也会影响房屋价格，这个房屋能住下一家人或者是四五个人的家庭吗？而这确实是基于房屋大小，以及真正决定一栋房子是否能适合你们家庭人数的卧室数。

换个话题，你可能知道邮政编码或许能作为一个特征，告诉你步行化程度。比如这附近是不是高度步行化，你是否能步行去杂货店或者是学校，以及你是否需要驾驶汽车。有些人喜欢居住在以步行为主的区域，另外根据邮政编码还和富裕程度相关（在美国是这样的）。但在其它国家也可能体现出附近学校的水平有多好。



在图上每一个画的小圆圈都可以是 **ReLU** 的一部分，也就是指修正线性单元，或者其它稍微非线性的函数。基于房屋面积和卧室数量，可以估算家庭人口，基于邮编，可以估测步行化程度或者学校的质量。最后你可能会这样想，这些决定人们乐意花费多少钱。

对于一个房子来说，这些都是与它息息相关的事情。在这个情景里，家庭人口、步行化程度以及学校的质量都能帮助你预测房屋的价格。以此为例，x 是所有的这四个输入，y 是你尝试预测的价格，把这些单个的神经元叠加在一起，我们就有了一个稍微大一点的神经网络。这显示了神经网络的神奇之处，虽然我已经描述了一个神经网络，它可以需要你得到房屋面积、步行化程度和学校的质量，或者其它影响价格的因素。



神经网络的一部分神奇之处在于，当你实现它之后，你要做的只是输入x，就能得到输出y。

因为它可以自己计算你训练集中样本的数目以及所有的中间过程。所以，你实际上要做的就是：这里有四个输入的神经网络，这输入的特征可能是房屋的大小、卧室的数量、邮政编码和区域的富裕程度。给出这些输入的特征之后，神经网络的工作就是预测对应的价格。同时也注意到这些被叫做隐藏单元圆圈，在一个神经网络中，它们每个都从输入的四个特征获得自身输入，比如说，第一个结点代表家庭人口，而家庭人口仅仅取决于x_1和x_2特征，换句话说，在神经网络中，你决定在这个结点中想要得到什么，然后用所有的四个输入来计算想要得到的。因此，我们说输入层和中间层被紧密的连接起来了。

值得注意的是神经网络给予了足够多的关于x和y的数据，给予了足够的训练样本有关x和y。神经网络非常擅长计算从x到y的精准映射函数。

这就是一个基础的神经网络。你可能发现你自己的神经网络在监督学习的环境下是如此的有效和强大，也就是说你只要尝试输入一个x，即可把它映射成y，就好像我们在刚才房价预测的例子中看到的效果。

在下一个视频中，让我们复习一下更多监督学习的例子，有些例子会让你觉得你的网络会十分有用，并且你实际应用起来也是如此。

1.3 神经网络的监督学习(Supervised Learning with Neural Networks)

关于神经网络也有很多的种类，考虑到它们的使用效果，有些使用起来恰到好处，但事实表明，到目前几乎所有由神经网络创造的经济价值，本质上都离不开一种叫做监督学习的机器学习类别，让我们举例看看。

在监督学习中你有一些输入x，你想学习到一个函数来映射到一些输出y，比如我们之前提到的房价预测的例子，你只要输入有关房屋的一些特征，试着去输出或者估计价格y。我们举一些其它的例子，来说明神经网络已经被高效应用到其它地方。

Supervised Learning

Input(x)	Output (y)	Application
Home features	Price	Real Estate
Ad, user info	Click on ad? (0/1)	Online Advertising
Image	Object (1,...,1000)	Photo tagging
Audio	Text transcript	Speech recognition
English	Chinese	Machine translation
Image, Radar info	Position of other cars	Autonomous driving

如今应用深度学习获利最多的一个领域，就是在线广告。这也许不是最鼓舞人心的，但真的很赚钱。具体就是通过在网站上输入一个广告的相关信息，因为也输入了用户的信息，于是网站就会考虑是否向你展示广告。

神经网络已经非常擅长预测你是否会点开这个广告，通过向用户展示最有可能点开的广告，这就是神经网络在很多家公司难以置信地提高获利的一种应用。因为有了这种向你展示你最有可能点击的广告的能力，而这一点击的行为的改变会直接影响到一些大型的在线广告公司的收入。

计算机视觉在过去的几年里也取得了长足的进步，这也多亏了深度学习。你可以输入一个图像，然后想输出一个索引，范围从 1 到 1000 来试着告诉你这张照片，它可能是，比方说，1000 个不同的图像中的任何一个，所以你可能会选择用它来给照片打标签。

深度学习最近在语音识别方面的进步也是非常令人兴奋的，你现在可以将音频片段输入

神经网络，然后让它输出文本记录。得益于深度学习，机器翻译也有很大的发展。你可以利用神经网络输入英语句子，接着输出一个中文句子。

在自动驾驶技术中，你可以输入一幅图像，就好像一个信息雷达展示汽车前方有什么，据此，你可以训练一个神经网络，来告诉汽车在马路上面具体的位置，这就是神经网络在自动驾驶系统中的一个关键成分。

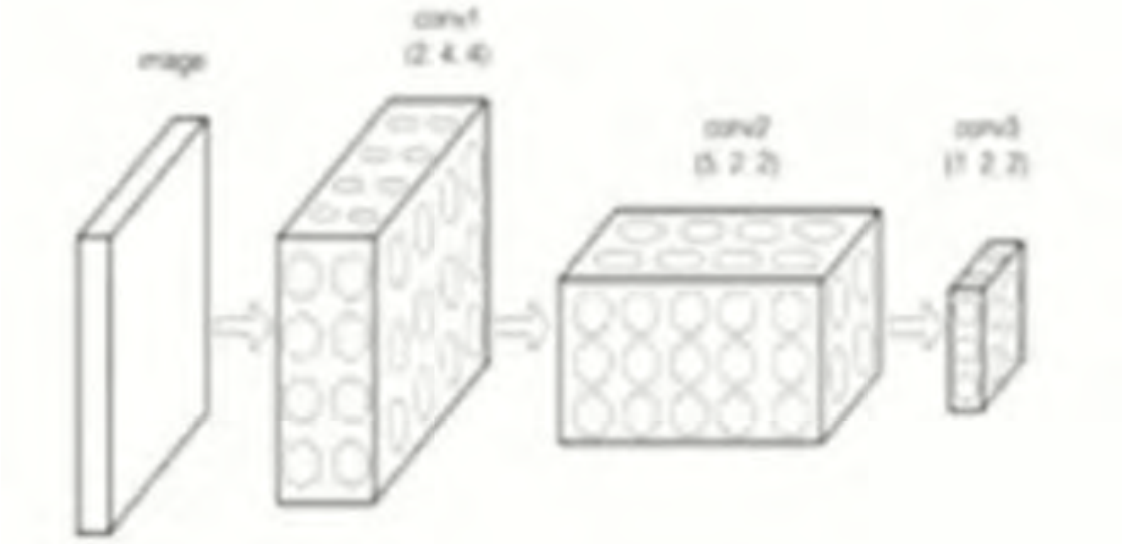
那么深度学习系统已经可以创造如此多的价值，通过智能的选择，哪些作为x哪些作为y，来针对于你当前的问题，然后拟合监督学习部分，往往是一个更大的系统，比如自动驾驶。这表明神经网络类型的轻微不同，也可以产生不同的应用，比如说，应用到我们在上一个视频提到的房地产领域，我们不就使用了一个普遍标准神经网络架构吗？

也许对于房地产和在线广告来说可能是相对的标准一些的神经网络，正如我们之前见到的。对于图像应用，我们经常在神经网络上使用卷积（**Convolutional Neural Network**），通常缩写为 **CNN**。对于序列数据，例如音频，有一个时间组件，随着时间的推移，音频被播放出来，所以音频是最自然的表现。作为一维时间序列（两种英文说法 **one-dimensional time series / temporal sequence**）.对于序列数据，经常使用 **RNN**，一种递归神经网络（**Recurrent Neural Network**），语言，英语和汉语字母表或单词都是逐个出现的，所以语言也是最自然的序列数据，因此更复杂的 **RNNs** 版本经常用于这些应用。

对于更复杂的应用比如自动驾驶，你有一张图片，可能会显示更多的 **CNN** 卷积神经网络结构，其中的雷达信息是完全不同的，你可能会有一个更定制的，或者一些更复杂的混合的神经网络结构。所以为了更具体地说明什么是标准的 **CNN** 和 **RNN** 结构，在文献中你可能见过这样的图片，这是一个标准的神经网络。

Standard NN

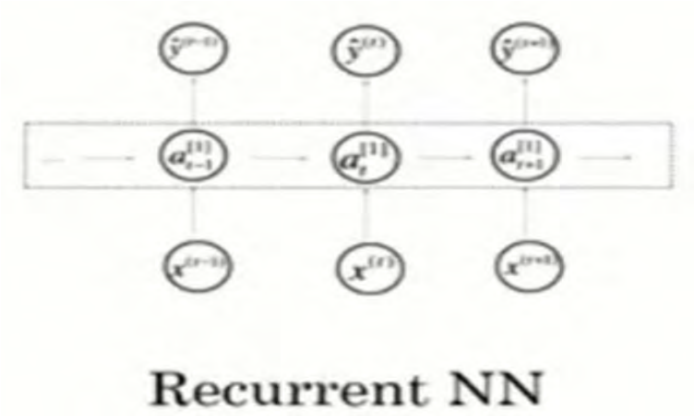
你也可能见过这样的图片，这是一个卷积神经网络的例子。



Convolutional NN

我们会在后面的课程了解这幅图的原理和实现，卷积网络(**CNN**)通常用于图像数据。

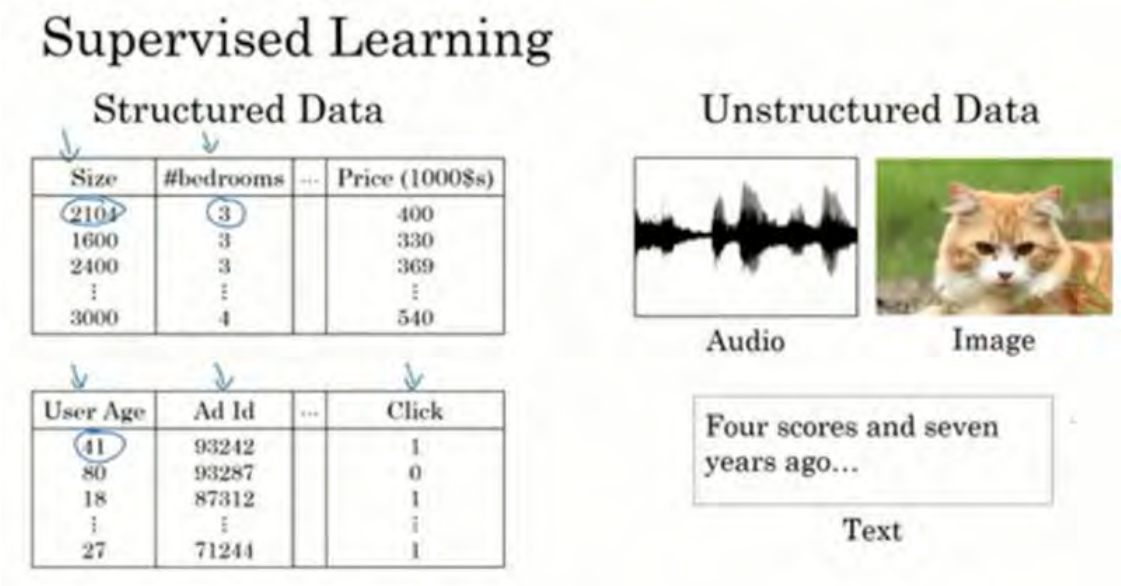
你可能也会看到这样的图片，而且你将在以后的课程中学习如何实现它。



递归神经网络(**RNN**)非常适合这种一维序列，数据可能是一个时间组成部分。

你可能也听说过机器学习对于结构化数据和非结构化数据的应用，结构化数据意味着数据的基本数据库。例如在房价预测中，你可能有一个数据库，有专门的几列数据告诉你卧室的大小和数量，这就是结构化数据。或预测用户是否会点击广告，你可能会得到关于用户的信息，比如年龄以及关于广告的一些信息，然后对你的预测分类标注，这就是结构化数据，意思是每个特征，比如说房屋大小卧室数量，或者是一个用户的年龄，都有一个很好的定义。

相反非结构化数据是指比如音频，原始音频或者你想要识别的图像或文本中的内容。这里的特征可能是图像中的像素值或文本中的单个单词。



从历史经验上看，处理非结构化数据是很难的，与结构化数据比较，让计算机理解非结构化数据很难，而人类进化得非常善于理解音频信号和图像，文本是一个更近代的发明，但是人们真的很擅长解读非结构化数据。

神经网络的兴起就是这样最令人兴奋的事情之一，多亏了深度学习和神经网络，计算机现在能更好地解释非结构化数据，这是与几年前相比的结果，这为我们创造了机会。许多新的令人兴奋的应用被使用，语音识别、图像识别、自然语言文字处理，甚至可能比两三年前的还要多。因为人们天生就有本领去理解非结构化数据，你可能听说了神经网络更多在媒体非结构化数据的成功，当神经网络识别了一只猫时那真的很酷，我们都知道那意味着什么。

但结果也表明，神经网络在许多短期经济价值的创造，也是基于结构化数据的。比如更好的广告系统、更好的利润建议，还有更好的处理大数据的能力。许多公司不得不根据神经网络做出准确的预测。

因此在这门课中，我们将要讨论的许多技术都将适用，不论是对结构化数据还是非结构化数据。为了解释算法，我们将在使用非结构化数据的示例中多画一点图片，但正如你所想的，你自己团队里通过运用神经网络，我希望你能发现，神经网络算法对于结构化和非结构化数据都有用处。

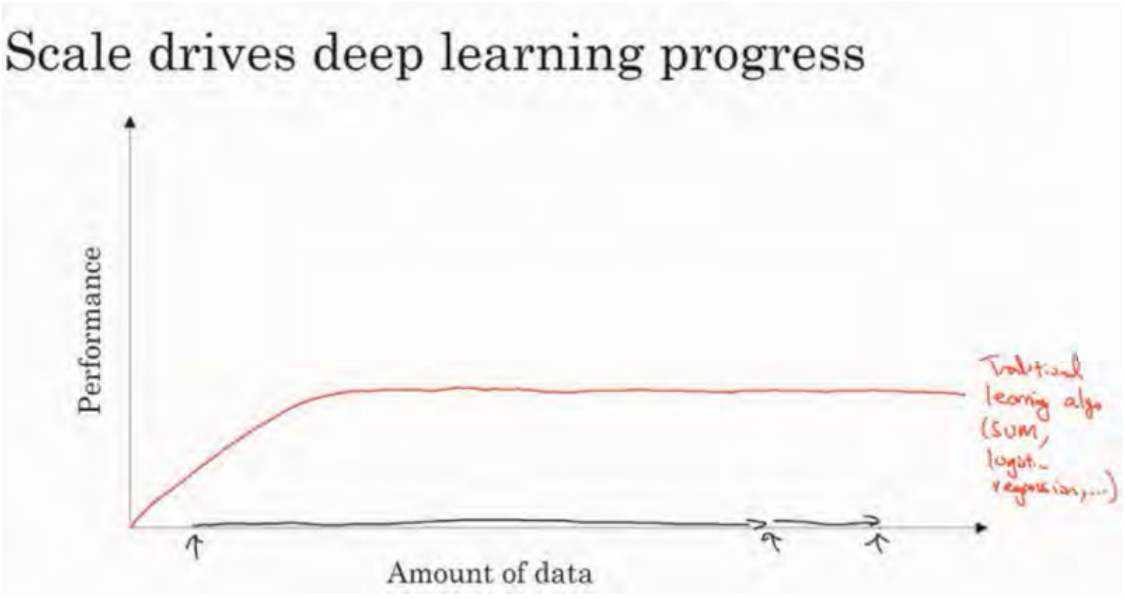
神经网络已经改变了监督学习，正创造着巨大的经济价值，事实证明，基本的神经网络背后的技术理念大部分都离我们不遥远，有的是几十年，那么为什么他们现在才刚刚起步，效果那么好，下一集视频中我们将讨论为什么最近的神经网络已经成为你可以使用的强大工具。

1.4 为什么深度学习会兴起？(Why is Deep Learning taking off?)

本节视频主要讲了推动深度学习变得如此热门的主要因素。包括数据规模、计算量及算法的创新。

深度学习和神经网络之前的基础技术理念已经存在大概几十年了，为什么它们现在才突然流行起来呢？本节课程主要讲述一些使得深度学习变得如此热门的主要驱动因素，这将会帮助你在你的组织机构内发现最好的时机来应用这些东西。

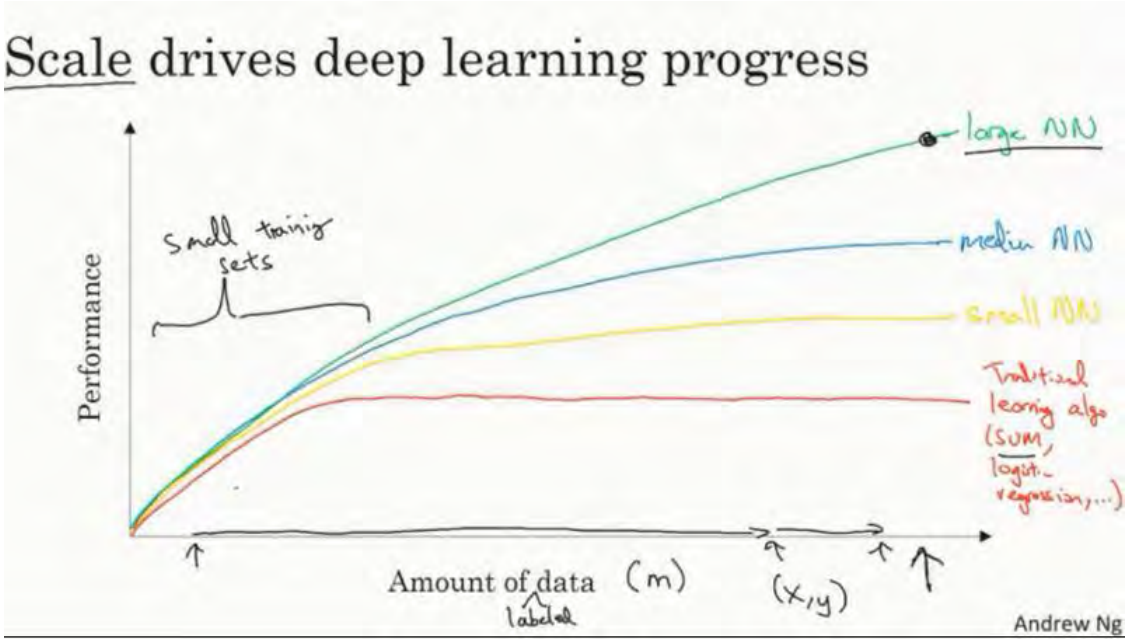
在过去的几年里，很多人都问我为什么深度学习能够如此有效。当我回答这个问题时，我通常给他们画个图，在水平轴上画一个形状，在此绘制出所有任务的数据量，而在垂直轴上，画出机器学习算法的性能。比如说准确率体现在垃圾邮件过滤或者广告点击预测，或者是神经网络在自动驾驶汽车时判断位置的准确性，根据图像可以发现，如果你把一个传统机器学习算法的性能画出来，作为数据量的一个函数，你可能得到一个弯曲的线，就像图中这样，它的性能一开始在增加更多数据时会上升，但是一段变化后它的性能就会像一个高原一样。假设你的水平轴拉的很长很长，它们不知道如何处理规模巨大的数据，而过去十年的社会里，我们遇到的很多问题只有相对较少的数据量。



多亏数字化社会的来临，现在的数据量都非常巨大，我们花了很多时间活动在这些数字的领域，比如在电脑网站上、在手机软件上以及其它数字化的服务，它们都能创建数据，同时便宜的相机被配置到移动电话，还有加速仪及各类各样的传感器，同时在物联网领域我们

也收集到了越来越多的数据。仅仅在过去的 20 年里对于很多应用，我们便收集到了大量的数据，远超过机器学习算法能够高效发挥它们优势的规模。

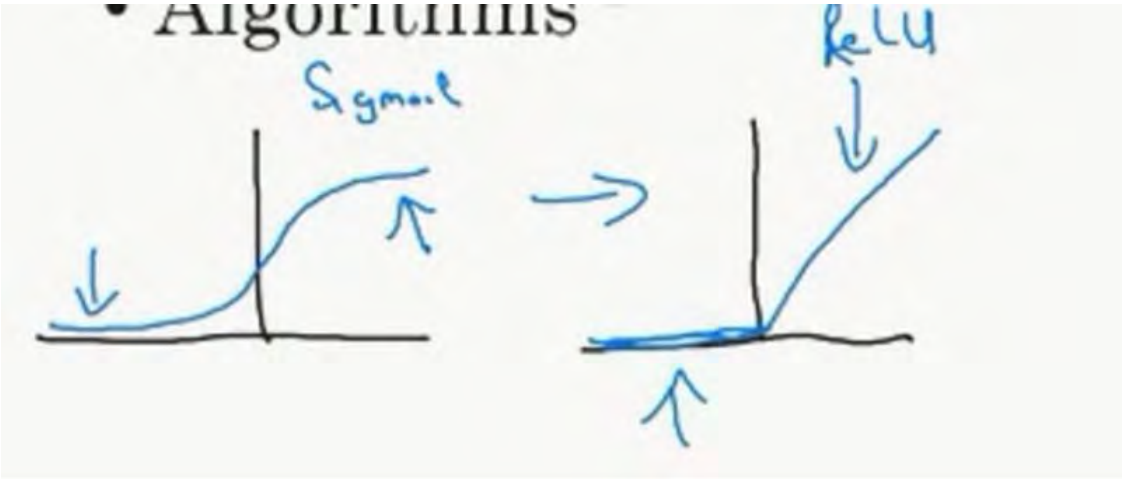
神经网络展现出的是，如果你训练一个小型的神经网络，那么这个性能可能会像下图黄色曲线表示那样；如果你训练一个稍微大一点的神经网络，比如说一个中等规模的神经网络（下图蓝色曲线），它在某些数据上面的性能也会更好一些；如果你训练一个非常大的神经网络，它就会变成下图绿色曲线那样，并且保持变得越来越好。因此可以注意到两点：如果你想要获得较高的性能体现，那么你有两个条件要完成，第一个是你需要训练一个规模足够大的神经网络，以发挥数据规模量巨大的优点，另外你需要能画到x轴的这个位置，所以你需要很多的数据。因此我们经常说规模一直在推动深度学习的进步，这里的规模指的也同时是神经网络的规模，我们需要一个带有许多隐藏单元的神经网络，也有许多的参数及关联性，就如同需要大规模的数据一样。事实上如今最可靠的方法来在神经网络上获得更好的性能，往往就是**要么训练一个更大的神经网络，要么投入更多的数据**，这只能在一定程度上起作用，因为最终你耗尽了数据，或者最终你的网络是如此大规模导致将要用太久的时间去训练，但是仅仅提升规模的的确确地让我们在深度学习的世界中摸索了很多时间。为了使这个图更加从技术上讲更精确一点，我在x轴下面已经写明的数据量，这儿加上一个标签（label）量，通过添加这个标签量，也就是指在训练样本时，我们同时输入x和标签y，接下来引入一点符号，使用小写的字母m表示训练集的规模，或者说训练样本的数量，这个小写字母m就横轴结合其他一些细节到这个图像中。



在这个小的训练集中，各种算法的优先级事实上定义的也不是很明确，所以如果你没有大量的训练集，那效果会取决于你的特征工程能力，那将决定最终的性能。假设有些人训练出了一个 **SVM**（支持向量机）表现的更接近正确特征，然而有些人训练的规模大一些，可能在这个小的训练集中 **SVM** 算法可以做的更好。因此你知道在这个图形区域的左边，各种算法之间的优先级并不是定义的很明确，最终的性能更多的是取决于你在用工程选择特征方面的能力以及算法处理方面的一些细节，只是在某些大数据规模非常庞大的训练集，也就是在右边这个m会非常的大时，我们能更加持续地看到更大的由神经网络控制的其它方法，因此如果你的任何某个朋友问你为什么神经网络这么流行，我会鼓励你也替他们画这样一个图形。

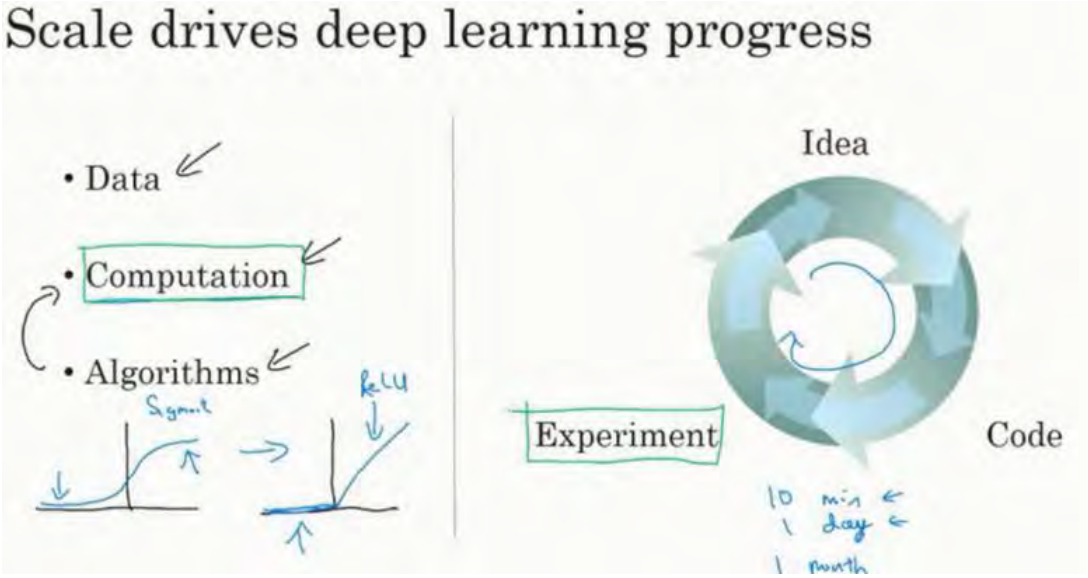
所以可以这么说，在深度学习萌芽的初期，数据的规模以及计算量，局限在我们对于训练一个特别大的神经网络的能力，无论是在 CPU 还是 GPU 上面，那都使得我们取得了巨大的进步。但是渐渐地，尤其是在最近这几年，我们也见证了算法方面的极大创新。许多算法方面的创新，一直是在尝试着使得神经网络运行的更快。

作为一个具体的例子，神经网络方面的一个巨大突破是从 **sigmoid** 函数转换到一个 **ReLU** 函数，这个函数我们在之前的课程里提到过。



如果你无法理解刚才我说的某个细节，也不需要担心，可以知道的一个使用 **sigmoid** 函数和机器学习问题是，在这个区域，也就是这个 **sigmoid** 函数的梯度会接近零，所以学习的速度会变得非常缓慢，因为当你实现梯度下降以及梯度接近零的时候，参数会更新的很慢，所以学习的速率也会变的很慢，而通过改变这个被叫做激活函数的东西，神经网络换用这一个函数，叫做 **ReLU** 的函数（修正线性单元），**ReLU** 它的梯度对于所有输入的负值都是零，因此梯度更加不会趋向逐渐减少到零。而这里的梯度，这条线的斜率在这左边是零，仅仅通过将 **Sigmod** 函数转换成 **ReLU** 函数，便能够使得一个叫做梯度下降（**gradient descent**）的算

法运行的更快，这就是一个或许相对比较简单的算法创新的例子。但是根本上算法创新所带来的影响，实际上是对计算带来的优化，所以有很多像这样的例子，我们通过改变算法，使得代码运行的更快，这也使得我们能够训练规模更大的神经网络，或者是多端口的网络。即使我们从所有的数据中拥有了大规模的神经网络，快速计算显得更加重要的另一个原因是，训练你的神经网络的过程，很多时候是凭借直觉的，往往你对神经网络架构有了一个想法，于是你尝试写代码实现你的想法，然后让你运行一个试验环境来告诉你，你的神经网络效果有多好，通过参考这个结果再返回去修改你的神经网络里面的一些细节，然后你不断的重复上面的操作，当你的神经网络需要很长时间去训练，需要很长时间重复这一循环，在这里就有很大的区别，根据你的生产效率去构建更高效的神经网络。当你能够有一个想法，试一试，看效果如何。在 10 分钟内，或者也许要花上一整天，如果你训练你的神经网络用了一个月的时间，有时候发生这样的事情，也是值得的，因为你很快得到了一个结果。在 10 分钟内或者一天内，你应该尝试更多的想法，那极有可能使得你的神经网络在你的应用方面工作的更好、更快的计算，在提高速度方面真的有帮助，那样你就能更快地得到你的实验结果。这也同时帮助了神经网络的实验人员和有关项目的研究人员在深度学习的工作中迭代的更快，也能够更快的改进你的想法，所有这些都使得整个深度学习的研究社群变的如此繁荣，包括令人难以置信地发明新的算法和取得不间断的进步，这些都是开拓者在做的事情，这些力量使得深度学习不断壮大。

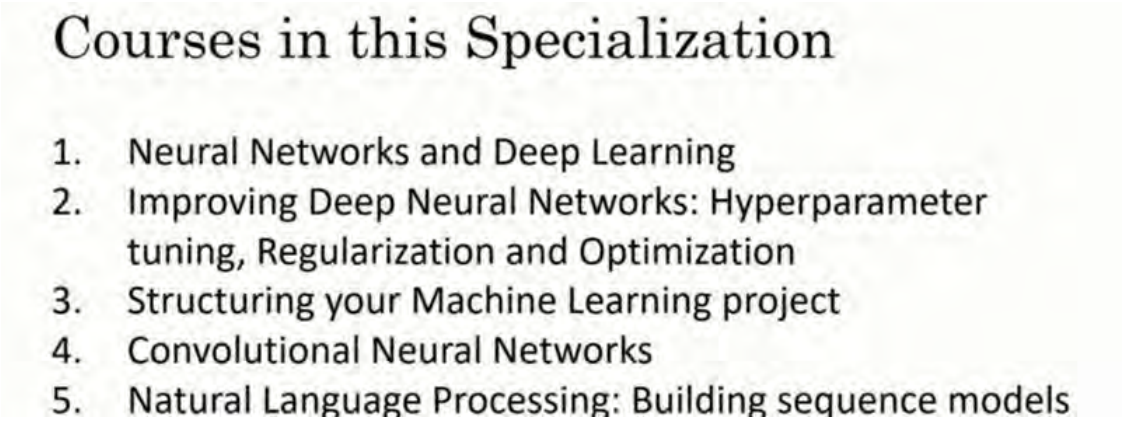


好消息是这些力量目前也正常不断的奏效，使得深度学习越来越好。研究表明我们的社会仍然正在抛出越来越多的数字化数据，或者用一些特殊的硬件来进行计算，比如说 **GPU**，以及更快的网络连接各种硬件。我非常有信心，我们可以做一个超级大规模的神经网络，而

计算的能力也会进一步的得到改善，还有算法相对的学习研究社区连续不断的在算法前沿产生非凡的创新。根据这些我们可以乐观地回答，同时对深度学习保持乐观态度，在接下来的这些年它都会变的越来越好。

1.5 关于这门课(About this Course)

你的学习进度已经快接近这个专项课程的第一门课的第一周结尾了，首先，快速地介绍一下下周的学习内容：



在第一个视频已经提到，这个专项有五门课程，目前正处于第一门课：神经网络与深度学习。在这门课中将教会你最重要的基础知识。当学习到第一门课末尾，你将学到如何建立一个深度神经网络并且使之奏效。

下面是关于第一门课的一些细节，这门课有四周的学习资料：

第一周：关于深度学习的介绍。在每一周的结尾也会有十个多选题用来检验自己对材料的理解；

第二周：关于神经网络的编程知识，了解神经网络的结构，逐步完善算法并思考如何使得神经网络高效地实现。从第二周开始做一些编程训练（付费项目），自己实现算法；

第三周：在学习了神经网络编程的框架之后，你将可以编写一个隐藏层神经网络，所以需要学习所有必须的关键概念来实现神经网络的工作；

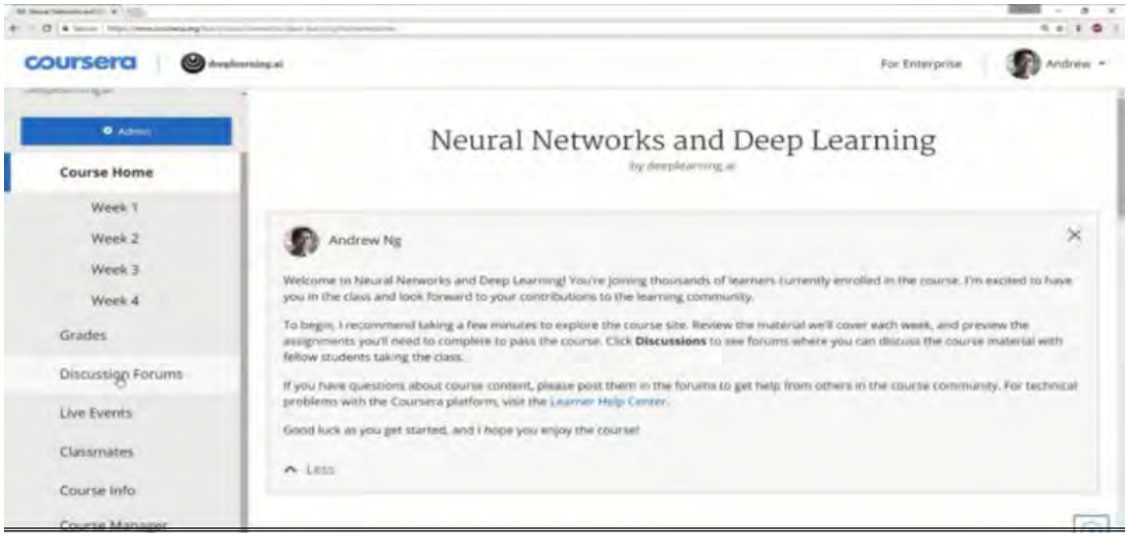
第四周：建立一个深层的神经网络。

这段视频即将结束，希望在这段视频之后，你们可以看看课程网站的十道选择题来检查自己的理解，不必复习前面的知识，有的知识是你现在不知道的，可以不断尝试，直到全部做对以理解全部概念。

1.6 课程资源(Course Resources)

我希望你们喜欢这门课程，为了帮助你们完成课程，本次课程将列举一些课程资源。

首先，如果你有任何疑问，或是想和其他同学讨论问题，或是想和包括我在内的教学人员讨论任何问题，或是想要归档一个错误，论坛是最好的去处，我和其他教学人员将定期关注论坛的内容。论坛也是一个你从同学那里得到问题答案的好地方，如果想要回答同学的问题，可以从课程首页来到论坛：



点击论坛标签可以进入论坛

在论坛上是提问的最佳途径，但是出于一些原因，可能要直接联系我们，可以将邮件发送到这个地址，我们会尽力阅读每一份邮件并尝试解决普遍出现的问。由于邮件数量庞大，不一定能迅速回复每一封邮件。另外，一些公司会尝试给员工做深度学习培训，如果你们想对员工负责，聘请专家培训上百甚至更多员工深度学习，请用企业邮箱与我们联系。我们处在大学学术开发的初始阶段，如果你是大学领导或者是管理人员，并且希望在你们学校开设一门深度学习课程，请通过大学邮箱联系我们。邮箱地址如下，祝你们好运！

Contact us: feedback@deeplearning.ai

Companies: enterprise@deeplearning.ai

Universities: academic@deeplearning.ai

第二周：神经网络的编程基础(Basics of Neural Network programming)

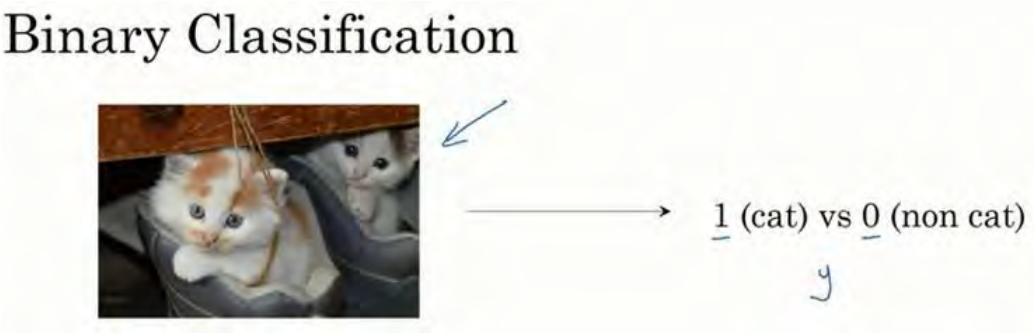
2.1 二分类(Binary Classification)

这周我们将学习神经网络的基础知识，其中需要注意的是，当实现一个神经网络的时候，我们需要知道一些非常重要的技术和技巧。例如有一个包含m个样本的训练集，你很可能习惯于用一个 **for** 循环来遍历训练集中的每个样本，但是当实现一个神经网络的时候，我们通常不直接使用 **for** 循环来遍历整个训练集，所以在这周的课程中你将学会如何处理训练集。

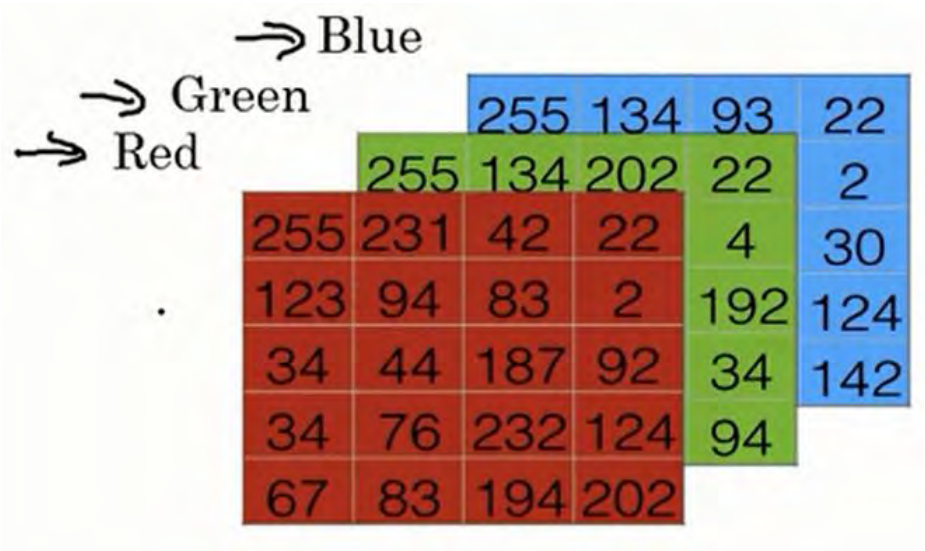
另外在神经网络的计算中，通常先有一个叫做前向暂停(**forward pause**)或叫做前向传播(**foward propagation**)的步骤，接着有一个叫做反向暂停(**backward pause**) 或叫做反向传播(**backward propagation**)的步骤。所以这周我也会向你介绍为什么神经网络的训练过程可以分为前向传播和反向传播两个独立的部分。

在课程中我将使用逻辑回归(**logistic regression**)来传达这些想法，以使大家能够更加容易地理解这些概念。即使你之前了解过逻辑回归，我认为这里还是有些新的、有趣的东西等着你去发现和了解，所以现在开始进入正题。

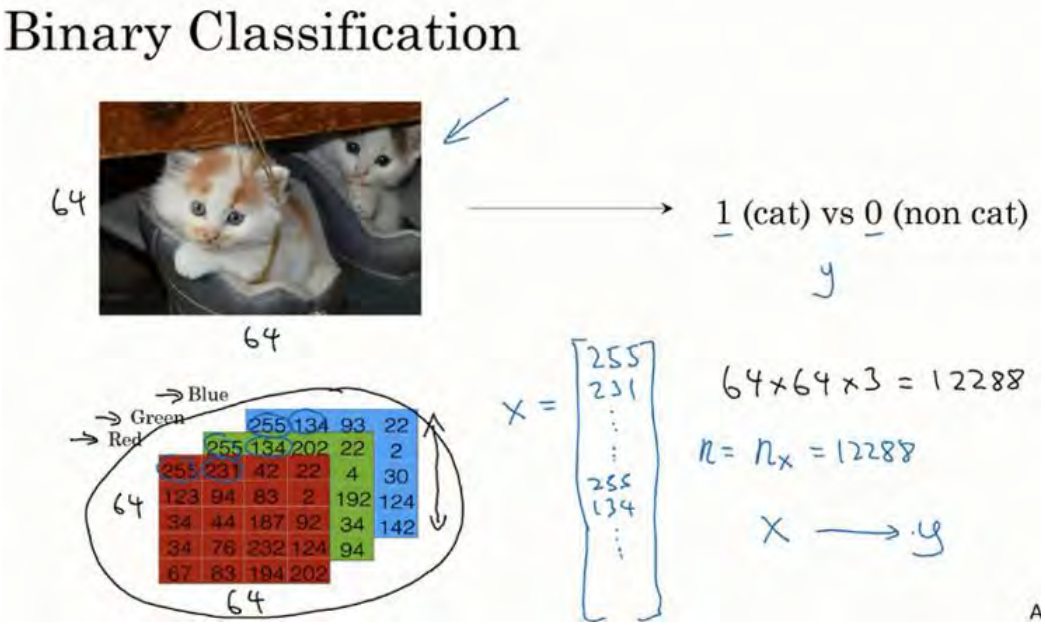
逻辑回归是一个用于二分类(**binary classification**)的算法。首先我们从一个问题开始说起，这里有一个二分类问题的例子，假如你有一张图片作为输入，比如这只猫，如果识别这张图片为猫，则输出标签 1 作为结果；如果识别出不是猫，那么输出标签 0 作为结果。现在我们可以用字母 y来 表示输出的结果标签，如下图所示：



我们来看看一张图片在计算机中是如何表示的，为了保存一张图片，需要保存三个矩阵，它们分别对应图片中的红、绿、蓝三种颜色通道，如果你的图片大小为 64x64 像素，那么你就有三个规模为 64x64 的矩阵，分别对应图片中红、绿、蓝三种像素的强度值。为了便于表示，这里我画了三个很小的矩阵，注意它们的规模为 5x4 而不是 64x64，如下图所示：



为了把这些像素值放到一个特征向量中，我们需要把这些像素值提取出来，然后放入一个特征向量x。为了把这些像素值转换为特征向量 x，我们需要像下面这样定义一个特征向量 x 来表示这张图片，我们把所有的像素都取出来，例如 255、231 等等，直到取完所有的红色像素，接着最后是 255、134、…、255、134 等等，直到得到一个特征向量，把图片中所有的红、绿、蓝像素值都列出来。如果图片的大小为 64x64 像素，那么向量 x 的总维度，将是 64 乘以 64 乘以 3，这是三个像素矩阵中像素的总量。在这个例子中结果为 12,288。现在我们用$n_x = 12{,}288$，来表示输入特征向量的维度，有时候为了简洁，我会直接用小写的n来表示输入特征向量x的维度。所以在二分类问题中，我们的目标就是习得一个分类器，它以图片的特征向量作为输入，然后预测输出结果y为 1 还是 0，也就是预测图片中是否有猫：



接下来我们说明一些在余下课程中，需要用到的一些符号。

符号定义 ：

x：表示一个n_x维数据，为输入数据，维度为$(n_x, 1)$；

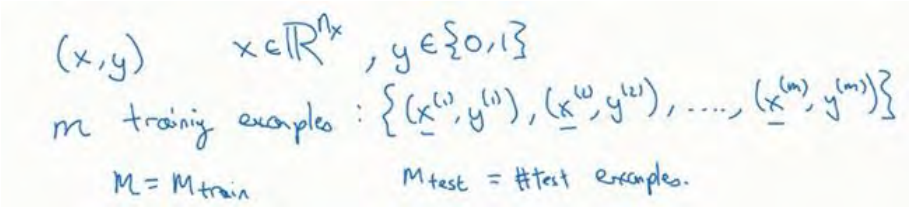
y：表示输出结果，取值为$(0,1)$；

$(x^{(i)}, y^{(i)})$：表示第i组数据，可能是训练数据，也可能是测试数据，此处默认为训练数据；

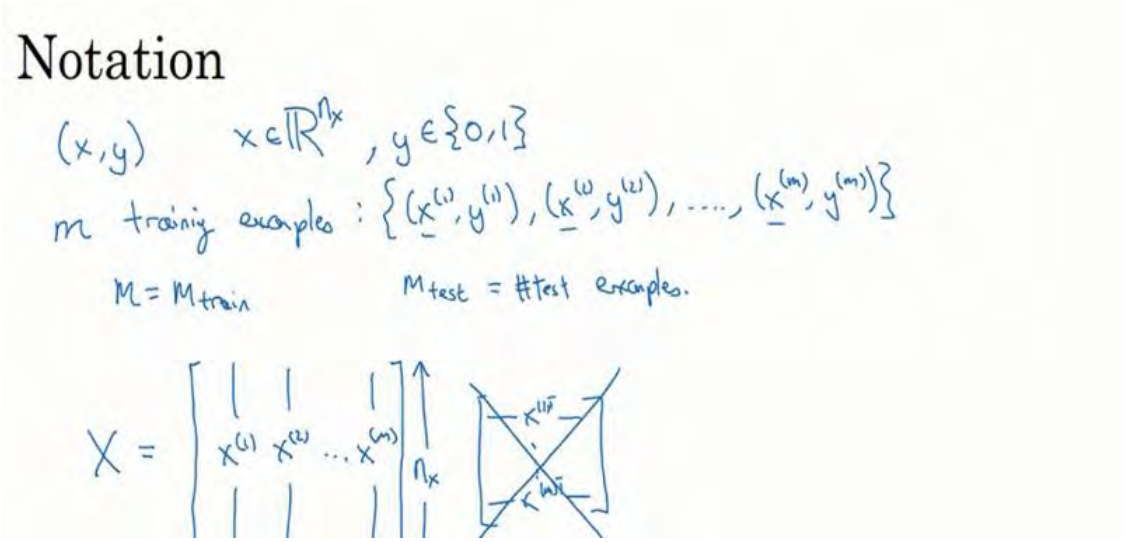
$X = [x^{(1)}, x^{(2)}, \ldots, x^{(m)}]$：表示所有的训练数据集的输入值，放在一个 $n_x \times m$的矩阵中，其中m表示样本数目;

$Y = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]$：对应表示所有训练数据集的输出值，维度为$1 \times m$。

用一对(x, y)来表示一个单独的样本，x代表n_x维的特征向量，y 表示标签(输出结果)只能为 0 或 1。 而训练集将由m个训练样本组成，其中$(x^{(1)}, y^{(1)})$表示第一个样本的输入和输出，$(x^{(2)}, y^{(2)})$表示第二个样本的输入和输出，直到最后一个样本$(x^{(m)}, y^{(m)})$，然后所有的这些一起表示整个训练集。有时候为了强调这是训练样本的个数，会写作M_{train}，当涉及到测试集的时候，我们会使用M_{test}来表示测试集的样本数，所以这是测试集的样本数：

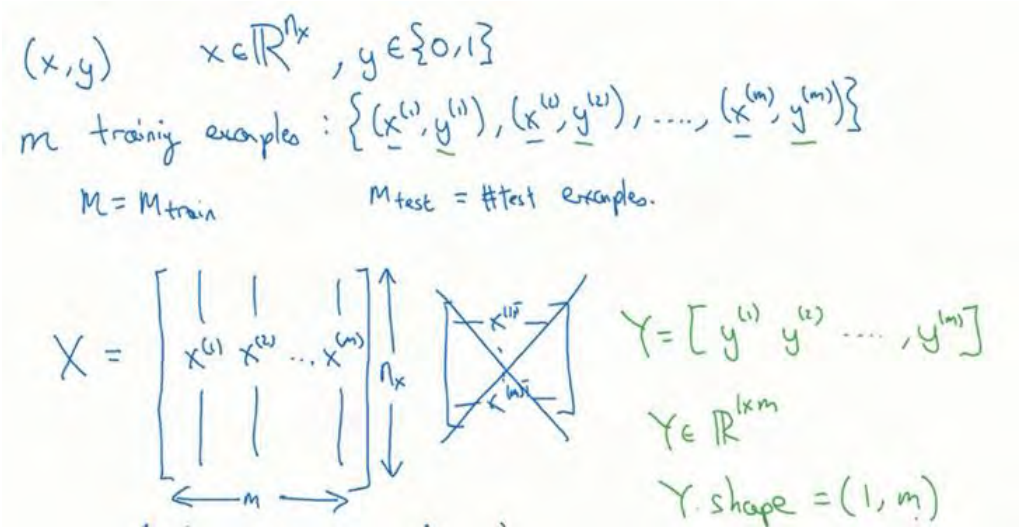


最后为了能把训练集表示得更紧凑一点，我们会定义一个矩阵用大写X的表示，它由输入向量$x^{(1)}$、$x^{(2)}$等组成，如下图放在矩阵的列中，所以现在我们把$x^{(1)}$作为第一列放在矩阵中，$x^{(2)}$作为第二列，$x^{(m)}$放到第m列，然后我们就得到了训练集矩阵X。所以这个矩阵有m列，m是训练集的样本数量，然后这个矩阵的高度记为n_x，注意有时候可能因为其他某些原因，矩阵X会由训练样本按照行堆叠起来而不是列，如下图所示：$x^{(1)}$的转置直到$x^{(m)}$的转置，但是在实现神经网络的时候，使用左边的这种形式，会让整个实现的过程变得更加简单：



现在来简单温习一下:X是一个规模为n_x乘以m的矩阵，当你用 **Python** 实现的时候，你会看到 `X.shape`，这是一条 **Python** 命令，用于显示矩阵的规模，即 `X.shape` 等于(n_x, m)，X是一个规模为n_x乘以m的矩阵。所以综上所述，这就是如何将训练样本（输入向量X的集合）表示为一个矩阵。

那么输出标签y呢？同样的道理，为了能更加容易地实现一个神经网络，将标签y放在列中将会使得后续计算非常方便，所以我们定义大写的Y等于$y^{(1)}, y^{(m)}, \ldots, y^{(m)}$，所以在这里是一个规模为 1 乘以$m$的矩阵，同样地使用 **Python** 将表示为 `Y.shape` 等于$(1, m)$，表示这是一个规模为 1 乘以m的矩阵。



当你在后面的课程中实现神经网络的时候，你会发现，一个好的符号约定能够将不同训练样本的数据很好地组织起来。而我所说的数据不仅包括 x 或者 y 还包括之后你会看到的其他的量。将不同的训练样本的数据提取出来，然后就像刚刚我们对 x 或者 y 所做的那样，将他们堆叠在矩阵的列中，形成我们之后会在逻辑回归和神经网络上要用到的符号表示。

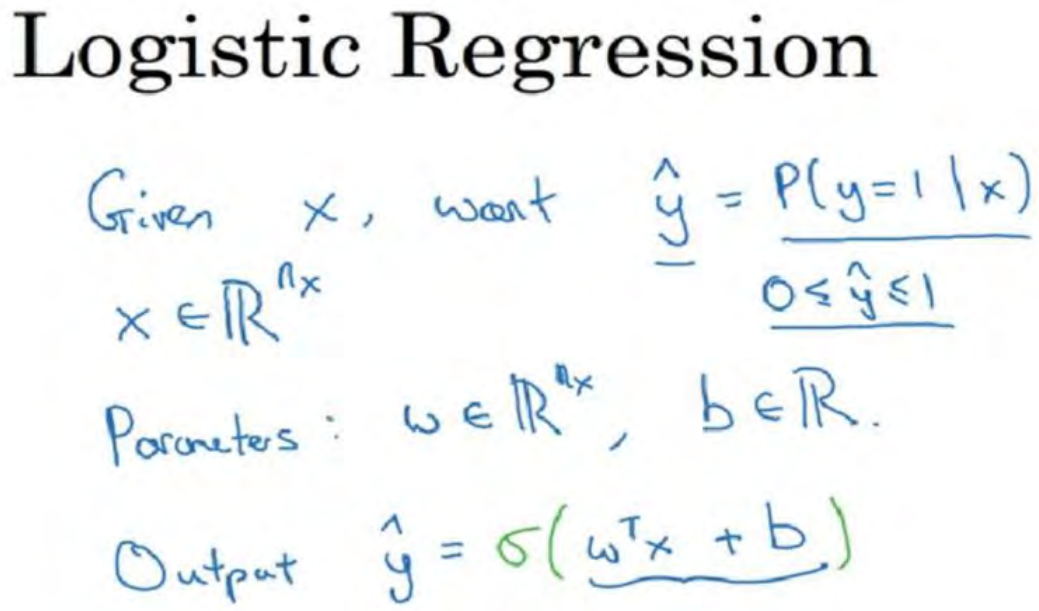
如果有时候你忘了这些符号的意思，比如什么是 m，或者什么是 n，或者忘了其他一些东西，我们也会在课程的网站上放上符号说明，然后你可以快速地查阅每个具体的符号代表什么意思，好了，我们接着到下一个视频，在下个视频中，我们将以逻辑回归作为开始。

备注：附录里也写了符号说明。

2.2 逻辑回归(Logistic Regression)

在这个视频中，我们会重温逻辑回归学习算法，该算法适用于二分类问题，本节将主要介绍逻辑回归的 **Hypothesis Function**（假设函数）。

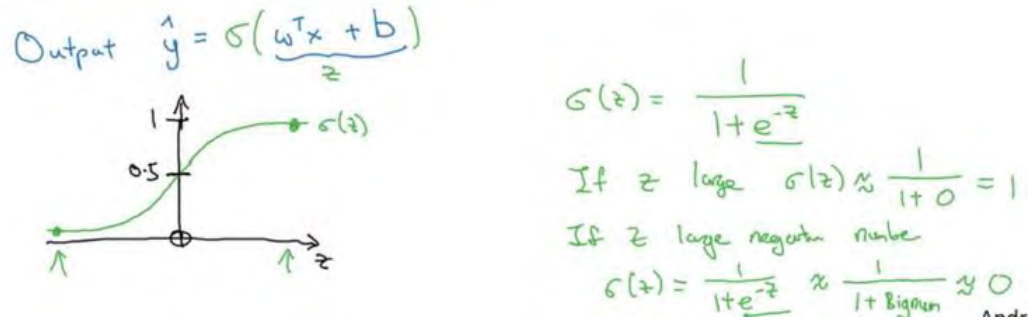
对于二元分类问题来讲，给定一个输入特征向量X，它可能对应一张图片，你想识别这张图片识别看它是否是一只猫或者不是一只猫的图片，你想要一个算法能够输出预测，你只能称之为$\hat{y}$，也就是你对实际值 y 的估计。更正式地来说，你想让 $\hat{y}$ 表示 y 等于 1 的一种可能性或者是机会，前提条件是给定了输入特征X。换句话来说，如果X是我们在上个视频看到的图片，你想让 $\hat{y}$ 来告诉你这是一只猫的图片的机率有多大。在之前的视频中所说的，X是一个n_x维的向量（相当于有n_x个特征的特征向量）。我们用w来表示逻辑回归的参数，这也是一个n_x维向量（因为w实际上是特征权重，维度与特征向量相同），参数里面还有b，这是一个实数（表示偏差）。所以给出输入x以及参数w和b之后，我们怎样产生输出预测值$\hat{y}$，一件你可以尝试却不可行的事是让$\hat{y} = w^T x + b$。



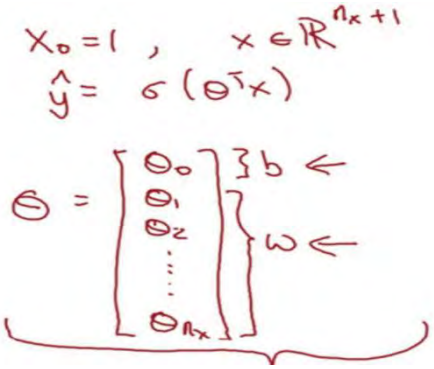
这时候我们得到的是一个关于输入x的线性函数，实际上这是你在做线性回归时所用到的，但是这对于二元分类问题来讲不是一个非常好的算法，因为你想让$\hat{y}$表示实际值y等于 1 的机率的话，$\hat{y}$ 应该在 0 到 1 之间。这是一个需要解决的问题，因为$w^T x + b$可能比 1 要大得多，或者甚至为一个负值。对于你想要的在 0 和 1 之间的概率来说它是没有意义的，因此在逻辑回归中，我们的输出应该是$\hat{y}$等于由上面得到的线性函数式子作为自变量的 **sigmoid** 函数中，公式如上图最下面所示，将线性函数转换为非线性函数。

下图是 **sigmoid** 函数的图像，如果我把水平轴作为z轴，那么关于z的 **sigmoid** 函数是这

样的，它是平滑地从 0 走向 1，让我在这里标记纵轴，这是 0，曲线与纵轴相交的截距是 0.5，这就是关于z的 **sigmoid** 函数的图像。我们通常都使用z来表示$w^T x + b$的值。



关于 **sigmoid** 函数的公式是这样的，$\sigma(z) = \frac{1}{1+e^{-z}}$,在这里$z$是一个实数，这里要说明一些要注意的事情，如果$z$非常大那么$e^{-z}$将会接近于 0，关于$z$的 **sigmoid** 函数将会近似等于 1 除以 1 加上某个非常接近于 0 的项，因为e 的指数如果是个绝对值很大的负数的话，这项将会接近于 0，所以如果z很大的话那么关于z的 **sigmoid** 函数会非常接近 1。相反地，如果z非常小或者说是一个绝对值很大的负数，那么关于e^{-z}这项会变成一个很大的数，你可以认为这是 1 除以 1 加上一个非常非常大的数，所以这个就接近于 0。实际上你看到当z变成一个绝对值很大的负数，关于z的 **sigmoid** 函数就会非常接近于 0，因此当你实现逻辑回归时，你的工作就是去让机器学习参数w以及b这样才使得$\hat{y}$成为对$y = 1$这一情况的概率的一个很好的估计。



在继续进行下一步之前，介绍一种符号惯例，可以让参数w和参数b分开。在符号上要注意的一点是当我们对神经网络进行编程时经常会让参数w和参数b分开，在这里参数b对应的是一种偏置。在之前的机器学习课程里，你可能已经见过处理这个问题时的其他符号表示。比如在某些例子里，你定义一个额外的特征称之为x_0，并且使它等于 1，那么现在X就是一个n_x加 1 维的变量，然后你定义$\hat{y} = \sigma(\theta^T x)$的 **sigmoid** 函数。在这个备选的符号惯例里，你有一个参数向量$\theta_0, \theta_1, \theta_2, \ldots, \theta_{n_x}$，这样$\theta_0$就充当了$b$，这是一个实数，而剩下的$\theta_1$ 直到θ_{n_x}充当了w，结果就是当你实现你的神经网络时，有一个比较简单的方法是保持b和w分开。但

是在这节课里我们不会使用任何这类符号惯例，所以不用去担心。 现在你已经知道逻辑回归模型是什么样子了，下一步要做的是训练参数w和参数b，你需要定义一个代价函数，让我们在下节课里对其进行解释。

2.3 逻辑回归的代价函数（Logistic Regression Cost Function）

在上个视频中，我们讲了逻辑回归模型，这个视频里，我们讲逻辑回归的代价函数（也翻译作成本函数）。

为什么需要代价函数：

为了训练逻辑回归模型的参数参数w和参数b我们，需要一个代价函数，通过训练代价函数来得到参数w和参数b。先看一下逻辑回归的输出函数：

$$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b), \text{ where } \sigma(z) = \frac{1}{1+e^{-z}}$$

$$\text{Given } \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}, \text{ want } \hat{y}^{(i)} \approx y^{(i)}.$$

为了让模型通过学习调整参数，你需要给予一个m样本的训练集，这会让你在训练集上找到参数w和参数b,，来得到你的输出。

对训练集的预测值，我们将它写成$\hat{y}$，我们更希望它会接近于训练集中的y值，为了对上面的公式更详细的介绍，我们需要说明上面的定义是对一个训练样本来说的，这种形式也使用于每个训练样本，我们使用这些带有圆括号的上标来区分索引和样本，训练样本i所对应的预测值是$y^{(i)}$,是用训练样本的$w^T x^{(i)} + b$然后通过 **sigmoid** 函数来得到，也可以把z定义为$z^{(i)} = w^T x^{(i)} + b$,我们将使用这个符号$(i)$注解，上标$(i)$来指明数据表示$x$或者$y$或者$z$或者其他数据的第$i$个训练样本，这就是上标$(i)$的含义。

损失函数：

损失函数又叫做误差函数，用来衡量算法的运行情况，**Loss function:**$L(\hat{y}, y)$.

我们通过这个L称为的损失函数，来衡量预测输出值和实际值有多接近。一般我们用预测值和实际值的平方差或者它们平方差的一半，但是通常在逻辑回归中我们不这么做，因为当我们在学习逻辑回归参数的时候，会发现我们的优化目标不是凸优化，只能找到多个局部最优值，梯度下降法很可能找不到全局最优值，虽然平方差是一个不错的损失函数，但是我们在逻辑回归模型中会定义另外一个损失函数。

我们在逻辑回归中用到的损失函数是：

$$L(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

为什么要用这个函数作为逻辑损失函数？当我们使用平方误差作为损失函数的时候，你会想要让这个误差尽可能地小，对于这个逻辑回归损失函数，我们也想让它尽可能地小，为

了更好地理解这个损失函数怎么起作用，我们举两个例子：

当$y=1$时损失函数$L=-\log(\hat{y})$，

如果想要损失函数L尽可能得小，那么$\hat{y}$就要尽可能大，因为 **sigmoid** 函数取值[0,1]，所以$\hat{y}$会无限接近于 1。

当$y=0$时损失函数$L=-\log(1-\hat{y})$，

如果想要损失函数L尽可能得小，那么$\hat{y}$就要尽可能小，因为 **sigmoid** 函数取值[0,1]，所以$\hat{y}$会无限接近于 0。

在这门课中有很多的函数效果和现在这个类似，就是如果y等于 1，我们就尽可能让$\hat{y}$变大，如果y等于 0，我们就尽可能让 $\hat{y}$ 变小。 损失函数是在单个训练样本中定义的，它衡量的是算法在单个训练样本中表现如何，为了衡量算法在全部训练样本上的表现如何，我们需要定义一个算法的代价函数，算法的代价函数是对m个样本的损失函数求和然后除以m:

$$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L\left(\hat{y}^{(i)},y^{(i)}\right)=\frac{1}{m}\sum_{i=1}^{m}\left(-y^{(i)}\log\hat{y}^{(i)}-(1-y^{(i)})\log(1-\hat{y}^{(i)})\right)$$

损失函数只适用于像这样的单个训练样本，而代价函数是参数的总代价，所以在训练逻辑回归模型时候，我们需要找到合适的w和b，来让代价函数 J 的总代价降到最低。 根据我们对逻辑回归算法的推导及对单个样本的损失函数的推导和针对算法所选用参数的总代价函数的推导，结果表明逻辑回归可以看做是一个非常小的神经网络，在下一个视频中，我们会看到神经网络会做什么。

2.4 梯度下降法（Gradient Descent）

梯度下降法可以做什么？

在你测试集上，通过最小化代价函数（成本函数）$J(w,b)$来训练的参数w和b，

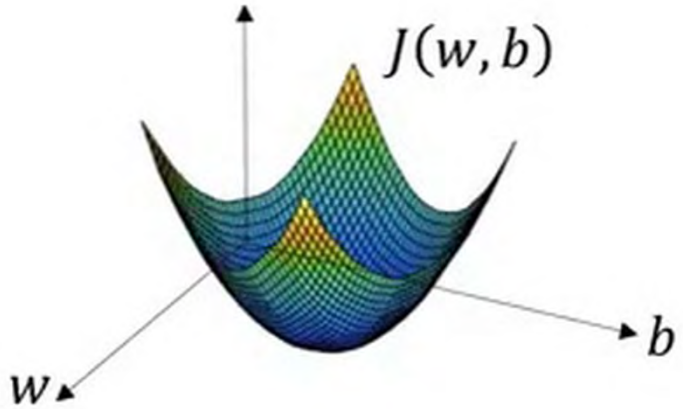
Gradient Descent

Recap: $\hat{y} = \sigma(w^T x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}}$

$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})$

如图，在第二行给出和之前一样的逻辑回归算法的代价函数（成本函数）

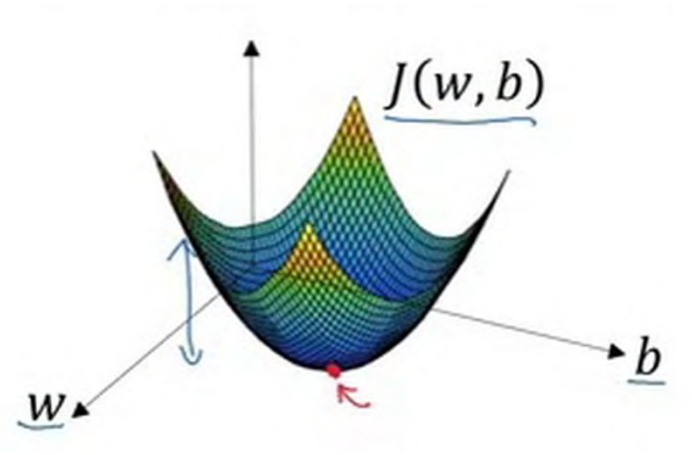
梯度下降法的形象化说明



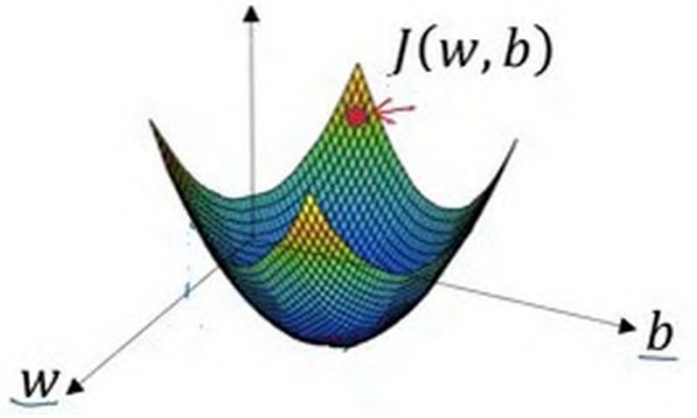
在这个图中，横轴表示你的空间参数w和b，在实践中，w可以是更高的维度，但是为了更好地绘图，我们定义w和b，都是单一实数，代价函数（成本函数）$J(w,b)$是在水平轴w和b上的曲面，因此曲面的高度就是$J(w,b)$在某一点的函数值。我们所做的就是找到使得代价函数（成本函数）$J(w,b)$函数值是最小值，对应的参数w和b。

如图，代价函数（成本函数）$J(w,b)$是一个凸函数(convex function)，像一个大碗一样。

如图，这就与刚才的图有些相反，因为它是非凸的并且有很多不同的局部最小值。由于逻辑回归的代价函数（成本函数）$J(w,b)$特性，我们必须定义代价函数（成本函数）$J(w,b)$为凸函数。 初始化w和b，

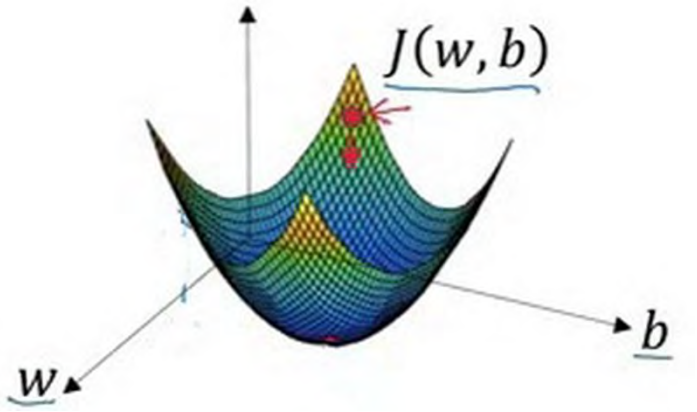


可以用如图那个小红点来初始化参数w和b，也可以采用随机初始化的方法，对于逻辑回归几乎所有的初始化方法都有效，因为函数是凸函数，无论在哪里初始化，应该达到同一点或大致相同的点。

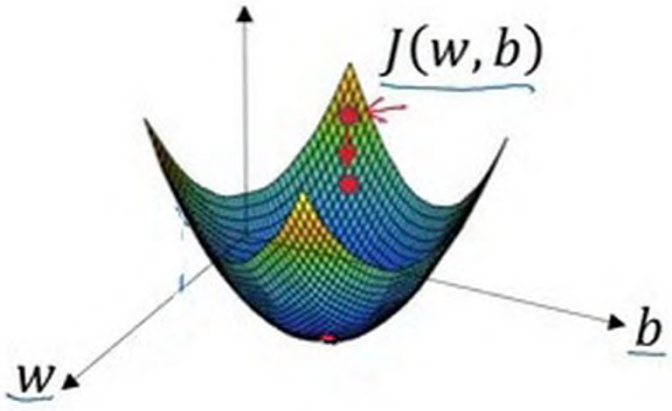


我们以如图的小红点的坐标来初始化参数w和b。

2. 朝最陡的下坡方向走一步，不断地迭代



我们朝最陡的下坡方向走一步，如图，走到了如图中第二个小红点处。

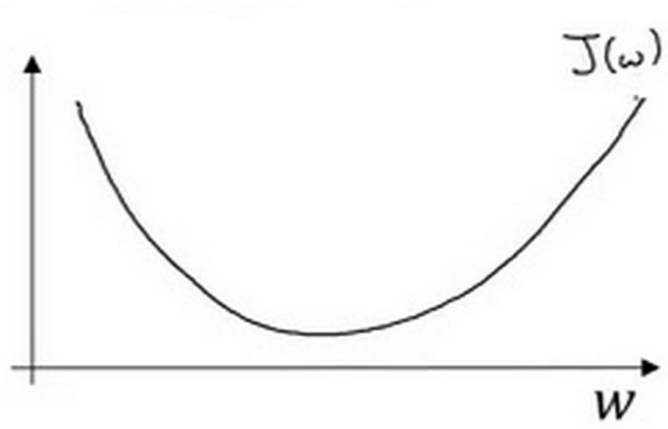


我们可能停在这里也有可能继续朝最陡的下坡方向再走一步，如图，经过两次迭代走到第三个小红点处。

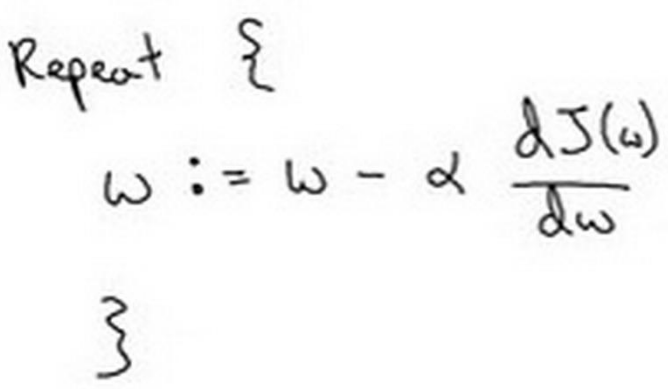
3.直到走到全局最优解或者接近全局最优解的地方

通过以上的三个步骤我们可以找到全局最优解，也就是代价函数（成本函数）$J(w,b)$这个凸函数的最小值点。

梯度下降法的细节化说明（仅有一个参数）



假定代价函数（成本函数）$J(w)$ 只有一个参数w，即用一维曲线代替多维曲线，这样可以更好画出图像。

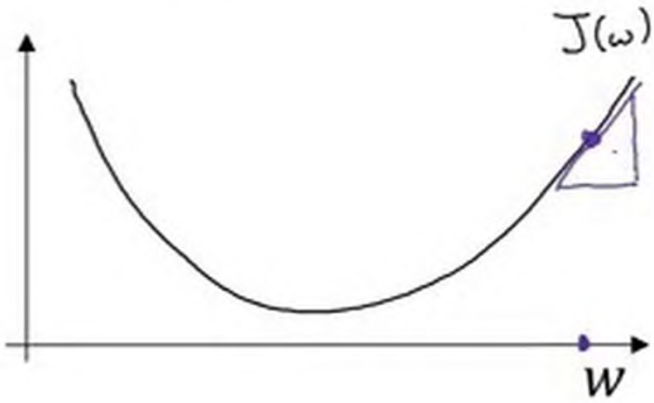


$$w := w - \alpha \frac{dJ(w)}{dw}$$

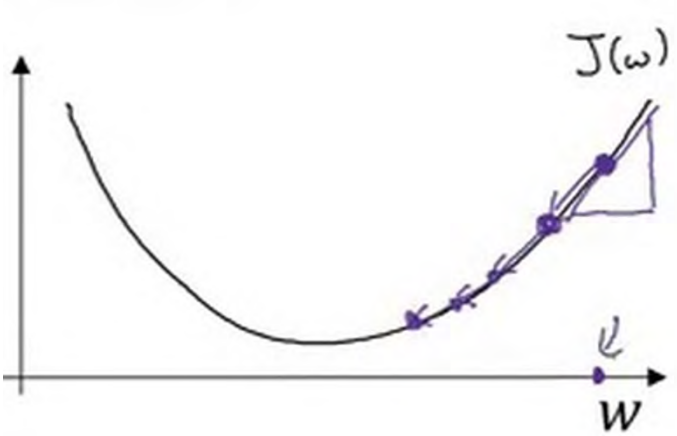
迭代就是不断重复做如图的公式:

:=表示更新参数,

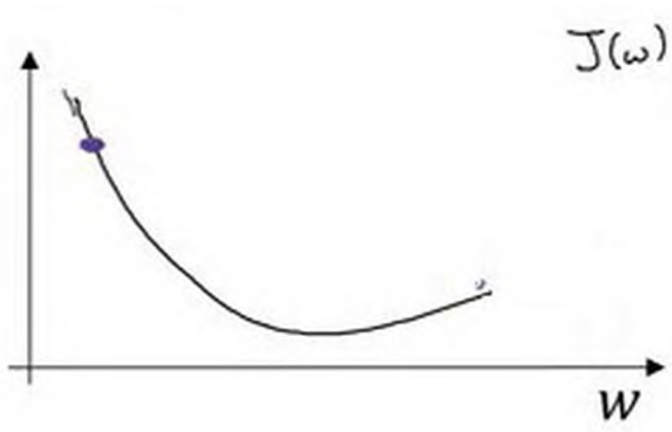
a 表示学习率（learning rate），用来控制步长（step），即向下走一步的长度$\frac{dJ(w)}{dw}$ 就是函数$J(w)$对w 求导（derivative），在代码中我们会使用dw表示这个结果



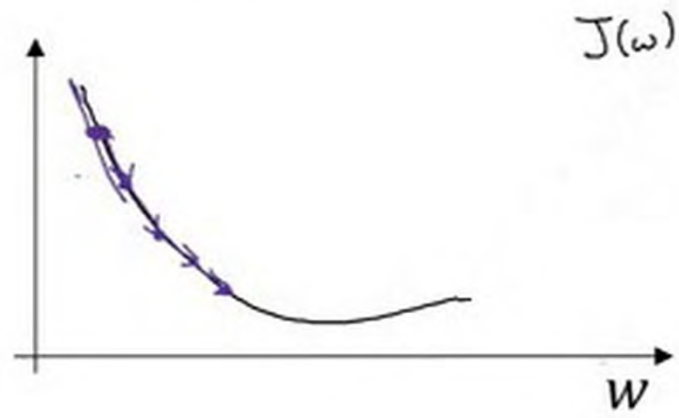
对于导数更加形象化的理解就是斜率（slope），如图该点的导数就是这个点相切于 $J(w)$ 的小三角形的高除宽。假设我们以如图点为初始化点，该点处的斜率的符号是正的，即$\frac{dJ(w)}{dw} > 0$，所以接下来会向左走一步。



整个梯度下降法的迭代过程就是不断地向左走，直至逼近最小值点。



假设我们以如图点为初始化点，该点处的斜率的符号是负的，即$\frac{dJ(w)}{dw}<0$，所以接下来会向右走一步。



整个梯度下降法的迭代过程就是不断地向右走，即朝着最小值点方向走。

梯度下降法的细节化说明（两个参数）

逻辑回归的代价函数（成本函数）$J(w,b)$是含有两个参数的。

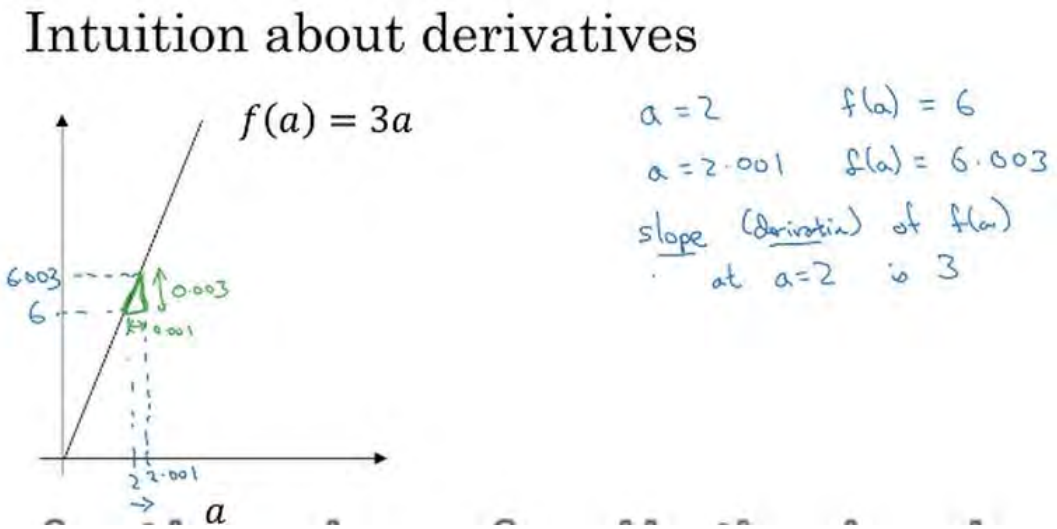
$$w := w - \alpha \frac{\partial J(w\ ,\ b)}{\partial w}$$

$$b := b - \alpha \frac{\partial J(w\ ,\ b)}{\partial b}$$

∂ 表示求偏导符号，可以读作 round， $\frac{\partial J(w,b)}{\partial w}$ 就是函数$J(w,b)$ 对w 求偏导，在代码中我们会使用dw 表示这个结果， $\frac{\partial J(w,b)}{\partial b}$ 就是函数$J(w,b)$对b 求偏导，在代码中我们会使用db 表示这个结果， 小写字母d 用在求导数（derivative），即函数只有一个参数， 偏导数符号∂ 用在求偏导（partial derivative），即函数含有两个以上的参数。

2.5 导数（Derivatives）

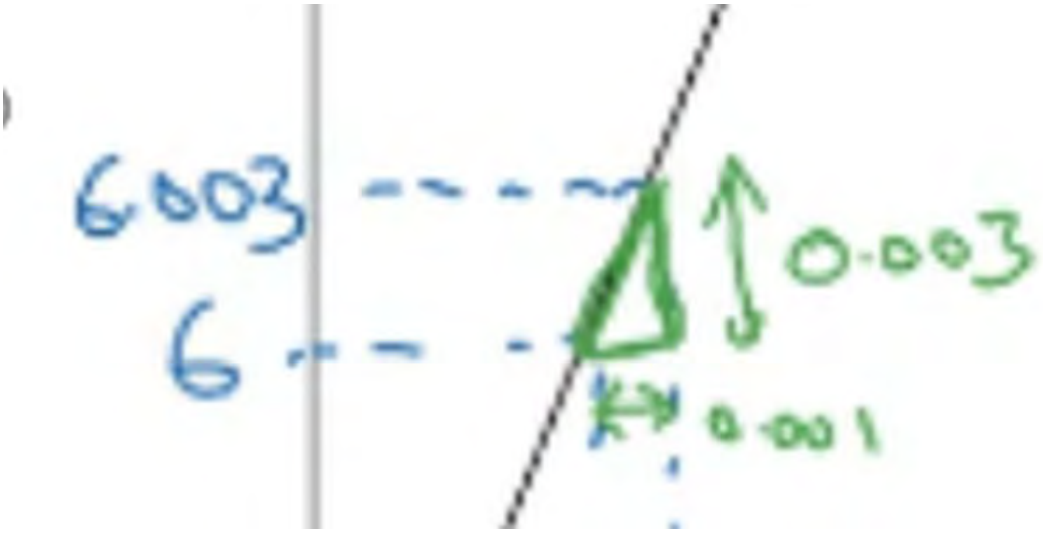
这个视频我主要是想帮你获得对微积分和导数直观的理解。或许你认为自从大学毕以后你再也没有接触微积分。这取决于你什么时候毕业，也许有一段时间了，如果你顾虑这点，请不要担心。为了高效应用神经网络和深度学习，你并不需要非常深入理解微积分。因此如果你观看这个视频或者以后的视频时心想：“哇哦，这些知识、这些运算对我来说很复杂。”我给你的建议是：坚持学习视频，最好下课后做作业，成功的完成编程作业，然后你就可以使用深度学习了。在第四周之后的学习中，你会看到定义的很多种类的函数，通过微积分他们能够帮助你把所有的知识结合起来，其中有的叫做前向函数和反向函数，因此你不需要了解所有你使用的那些微积分中的函数。所以你不用担心他们，除此之外在对深度学习的尝试中，这周我们要进一步深入了解微积分的细节。所有你只需要直观地认识微积分，用来构建和成功的应用这些算法。最后，如果你是精通微积分的那一小部分人群，你对微积分非常熟悉，你可以跳过这部分视频。其他同学让我们开始深入学习导数。



一个函数$f(a) = 3a$，它是一条直线。下面我们来简单理解下导数。让我们看看函数中几个点，假定$a = 2$，那么$f(a)$是a的 3 倍等于 6，也就是说如果$a = 2$，那么函数$f(a) = 6$。假定稍微改变一点点a的值，只增加一点，变为 2.001，这时a将向右做微小的移动。0.001 的差别实在是太小了，不能在图中显示出来，我们把它右移一点，现在$f(a)$等于a的 3 倍是 6.003，画在图里，比例不太符合。请看绿色高亮部分的这个小三角形，如果向右移动 0.001，那么$f(a)$增加 0.003，$f(a)$的值增加 3 倍于右移的a，因此我们说函数$f(a)$在$a = 2$，.是这个导数的斜率，或者说，当$a = 2$时，斜率是 3。导数这个概念意味着斜率，导数听起来是一个很可怕、很令人惊恐的词，但是斜率以一种很友好的方式来描述导数这个概念。所以提到导数，

我们把它当作函数的斜率就好了。更正式的斜率定义为在上图这个绿色的小三角形中，高除以宽。即斜率等于 0.003 除以 0.001，等于 3。或者说导数等于 3，这表示当你将a右移 0.001，$f(a)$的值增加 3 倍水平方向的量。

现在让我们从不同的角度理解这个函数。 假设$a = 5$ ，此时$f(a) = 3a = 15$。 把a右移一个很小的幅度，增加到 5.001，$f(a) = 15.003$。 即在$a = 5$ 时，斜率是 3，这就是表示，当微小改变变量a的值，$\frac{df(a)}{da} = 3$ 。一个等价的导数表达式可以这样写$\frac{d}{da}f(a)$ ，不管你是否将$f(a)$放在上面或者放在右边都没有关系。 在这个视频中，我讲解导数讨论的情况是我们将a偏移 0.001，如果你想知道导数的数学定义，导数是你右移很小的a值（不是 0.001，而是一个非常非常小的值）。通常导数的定义是你右移a(可度量的值)一个无限小的值，$f(a)$增加 3 倍（增加了一个非常非常小的值）。也就是这个三角形右边的高度。

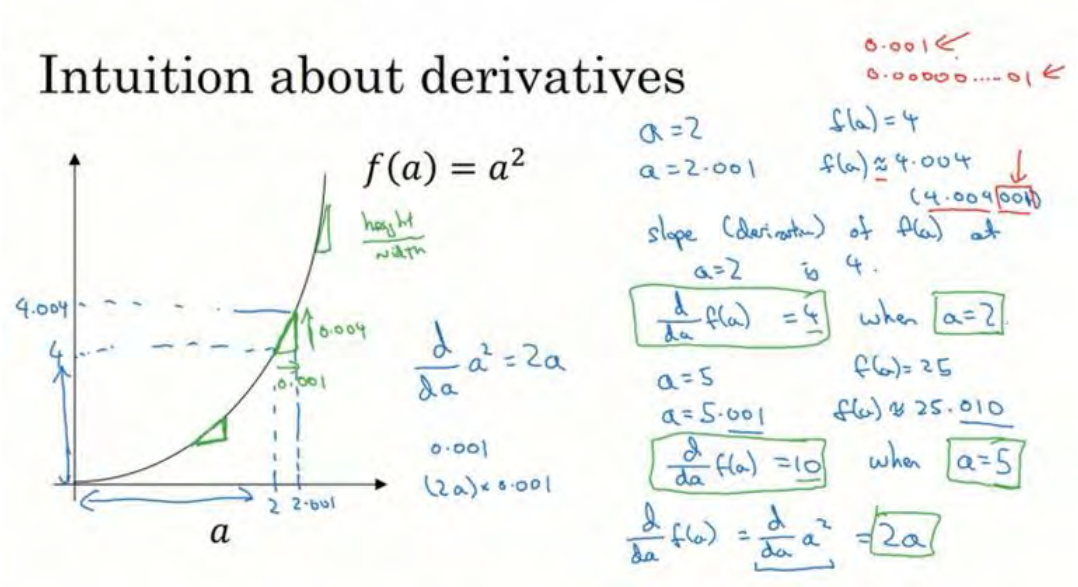


那就是导数的正式定义。但是为了直观的认识，我们将探讨右移$a = 0.001$ 这个值，即使 0.001 并不是无穷小的可测数据。导数的一个特性是：这个函数任何地方的斜率总是等于 3，不管$a = 2$或 $a = 5$，这个函数的斜率总等于 3，也就是说不管a的值如何变化，如果你增加 0.001，$f(a)$的值就增加 3 倍。这个函数在所有地方的斜率都相等。一种证明方式是无论你将小三角形画在哪里，它的高除以宽总是 3。

我希望带给你一种感觉：什么是斜率？什么是导函数？对于一条直线，在例子中函数的斜率，在任何地方都是 3。在下一个视频让我们看一个更复杂的例子，这个例子中函数在不同点的斜率是可变的。

2.6 更多的导数例子（More Derivative Examples）

在这个视频中我将给出一个更加复杂的例子，在这个例子中，函数在不同点处的斜率是不一样的，先来举个例子:

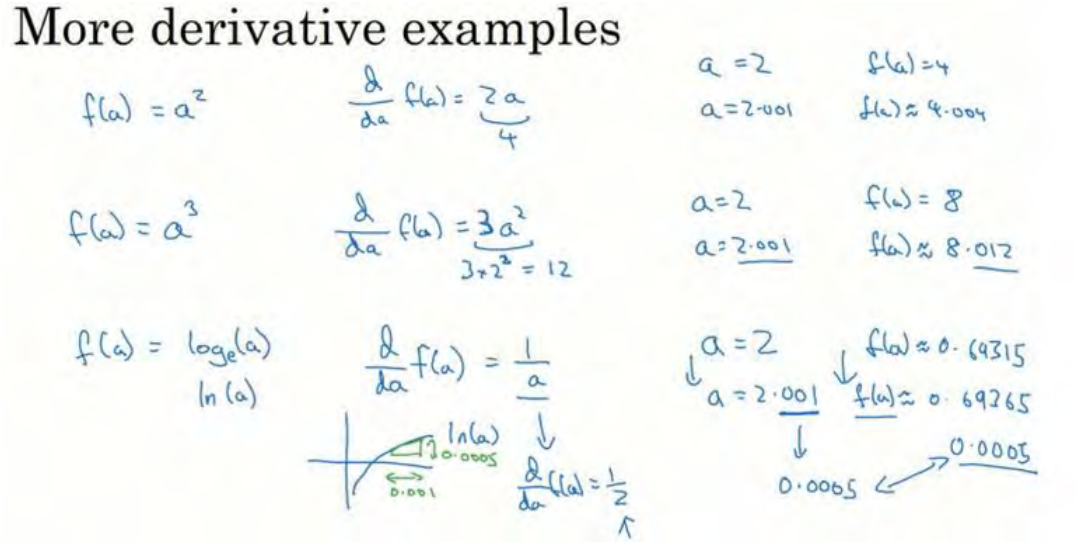


我在这里画一个函数，$f(a) = a^2$，如果$a = 2$ 的话，那么$f(a) = 4$。让我们稍稍往右推进一点点，现在$a = 2.001$ ，则$f(a) \approx 4.004$ (如果你用计算器算的话，这个准确的值应该为 4.004001 我只是为了简便起见，省略了后面的部分)，如果你在这儿画，一个小三角形 你就会发现，如果把a往右移动 0.001，那么$f(a)$将增大四倍，即增大 0.004。在微积分中我们把这个三角形斜边的斜率，称为$f(a)$在点$a = 2$ 处的导数(即为 4)，或者写成微积分的形式，当$a = 2$ 的时候， $\frac{d}{da}f(a) = 4$ 由此可知，函数$f(a) = a^2$，在a取不同值的时候，它的斜率是不同的，这和上个视频中的例子是不同的。

这里有种直观的方法可以解释，为什么一个点的斜率，在不同位置会不同如果你在曲线上，的不同位置画一些小小的三角形你就会发现，三角形高和宽的比值，在曲线上不同的地方，它们是不同的。所以当$a = 2$ 时，斜率为 4；而当$a = 5$时，斜率为 10 。如果你翻看微积分的课本，课本会告诉你，函数$f(a) = a^2$的斜率（即导数）为$2a$。这意味着任意给定一点a，如果你稍微将a，增大 0.001，那么你会看到$f(a)$将增大$2a$，即增大的值为点在a处斜率或导数，乘以你向右移动的距离。

现在有个小细节需要注意，导数增大的值，不是刚好等于导数公式算出来的值，而只是根据导数算出来的一个估计值。

为了总结这堂课所学的知识，我们再来看看几个例子：



假设$f(a)=a^3$ 如果你翻看导数公式表，你会发现这个函数的导数，等于$3a^2$。所以这是什么意思呢，同样地举一个例子：我们再次令$a=2$，所以$a^3=8$ ，如果我们又将a增大一点点，你会发现$f(a)\approx 8.012$，你可以自己检查一遍，如果我们取 8.012，你会发现2.001^3 ，和 8.012 很接近，事实上当$a=2$时，导数值为3×2^2，即$3\times 4=12$。所以导数公式，表明如果你将a向右移动 0.001 时，$f(a)$ 将会向右移动 12 倍，即 0.012。

来看最后一个例子，假设$f(a)=\log_e a$，有些可能会写作$\ln a$，函数$\log a$ 的斜率应该为$\frac{1}{a}$，所以我们可以解释如下：如果a取任何值，比如又取$a=2$，然后又把a向右边移动 0.001 那么$f(a)$将增大$\frac{1}{a}\times 0.001$，如果你借助计算器的话，你会发现当$a=2$时$f(a)\approx 0.69315$ ；而$a=2.001$时，$f(a)\approx 0.69365$。所以$f(a)$增大了 0.0005，如果你查看导数公式，当$a=2$的时候，导数值$\frac{d}{da}f(a)=\frac{1}{2}$。这表明如果你把 增大 0.001，$f(a)$将只会增大 0.001 的二分之一，即 0.0005。如果你画个小三角形你就会发现，如果x 轴增加了 0.001，那么y 轴上的函数$\log a$，将增大 0.001 的一半 即 0.0005。所以 $\frac{1}{a}$ ，当$a=2$时这里是 ，就是当$a=2$时这条线的斜率。这些就是有关，导数的一些知识。

在这个视频中，你只需要记住两点：

第一点，导数就是斜率，而函数的斜率，在不同的点是不同的。在第一个例子中$f(a)=3a$ ，这是一条直线，在任何点它的斜率都是相同的，均为 3。但是对于函数$f(a)=a^2$ ，或者$f(a)=\log a$，它们的斜率是变化的，所以它们的导数或者斜率，在曲线上不同的点处是不同的。

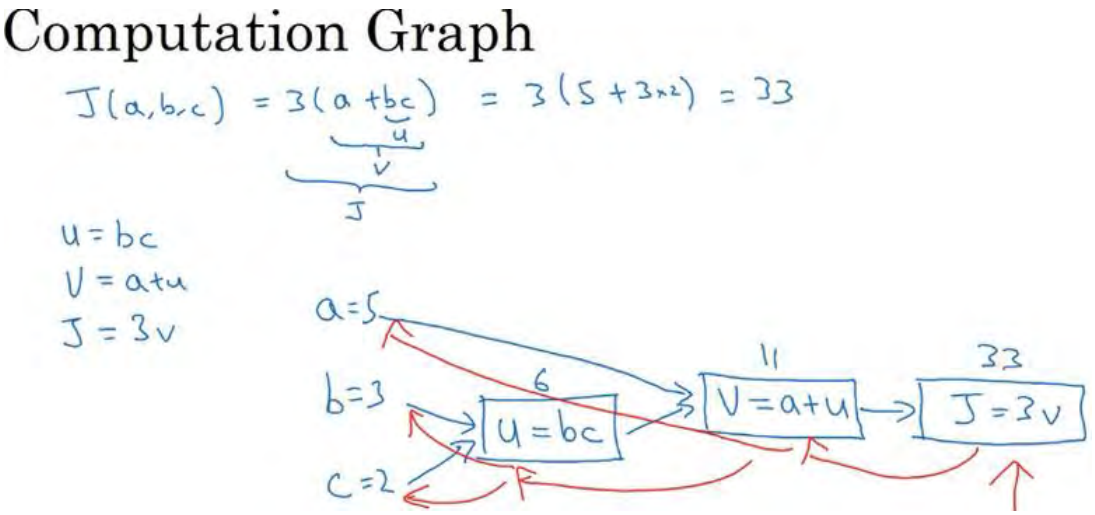
第二点，如果你想知道一个函数的导数，你可参考你的微积分课本或者维基百科，然后

你应该就能找到这些函数的导数公式。

最后我希望，你能通过我生动的讲解，掌握这些有关导数和斜率的知识，下一课我们将讲解计算图，以及如何用它来求更加复杂的函数的导数。

2.7 计算图（Computation Graph）

可以说，一个神经网络的计算，都是按照前向或反向传播过程组织的。首先我们计算出一个新的网络的输出（前向过程），紧接着进行一个反向传输操作。后者我们用来计算出对应的梯度或导数。计算图解释了为什么我们用这种方式组织这些计算过程。在这个视频中，我们将举一个例子说明计算图是什么。让我们举一个比逻辑回归更加简单的，或者说不那么正式的神经网络的例子。



我们尝试计算函数J，J是由三个变量a,b,c组成的函数，这个函数是3(a + bc) 。计算这个函数实际上有三个不同的步骤，首先是计算 b 乘以 c，我们把它储存在变量u中，因此$u = bc$； 然后计算$v = a + u$；最后输出$J = 3v$，这就是要计算的函数J。我们可以把这三步画成如下的计算图，我先在这画三个变量a,b,c，第一步就是计算$u = bc$，我在这周围放个矩形框，它的输入是b,c，接着第二步$v = a + u$，最后一步$J = 3v$。 举个例子: $a = 5, b = 3, c = 2$ ，$u = bc$就是 6， ，就是 5+6=11。J是 3 倍的 ，因此。即$3 \times (5 + 3 \times 2)$。如果你把它算出来，实际上得到 33 就是$J$的值。 当有不同的或者一些特殊的输出变量时，例如本例中的J和逻辑回归中你想优化的代价函数J，因此计算图用来处理这些计算会很方便。从这个小例子中我们可以看出，通过一个从左向右的过程，你可以计算出J的值。为了计算导数，从右到左（红色箭头，和蓝色箭头的过程相反）的过程是用于计算导数最自然的方式。 概括一下：计算图组织计算的形式是用蓝色箭头从左到右的计算，让我们看看下一个视频中如何进行反向红色箭头(也就是从右到左)的导数计算，让我们继续下一个视频的学习。

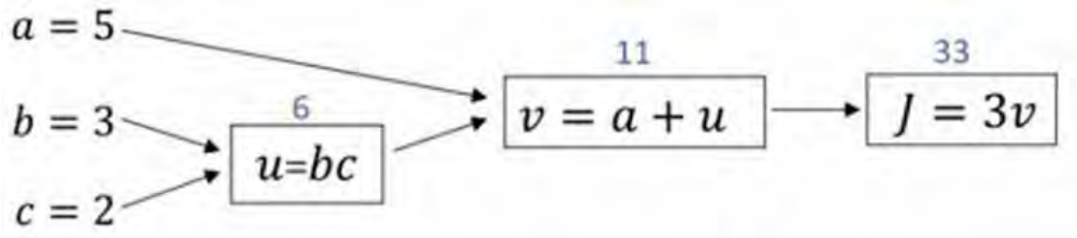
2.8 计算图的导数计算（Derivatives with a Computation Graph）

在上一个视频中，我们看了一个例子使用流程计算图来计算函数 J。现在我们清理一下流程图的描述，看看你如何利用它计算出函数J的导数。

下面用到的公式：

$\frac{dJ}{du}=\frac{dJ}{dv}\frac{dv}{db}$ ， $\frac{dJ}{db}=\frac{dJ}{du}\frac{du}{db}$ ， $\frac{dJ}{da}=\frac{dJ}{du}\frac{du}{da}$

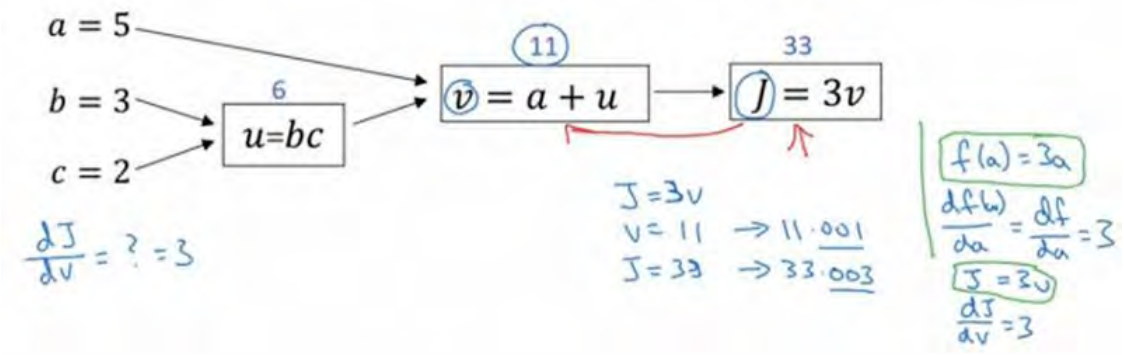
这是一个流程图：



假设你要计算$\frac{\mathrm{dJ}}{\mathrm{dv}}$，那要怎么算呢？好，比如说，我们要把这个$v$值拿过来，改变一下，那么$J$的值会怎么变呢？

所以定义上 $J=3v$，现在$v=11$，所以如果你让 v 增加一点点，比如到 11.001，那么 $J=3v=33.003$，所以我这里v增加了 0.001，然后最终结果是 J 上升到原来的 3 倍，所以 $\frac{\mathrm{dJ}}{\mathrm{dv}}=3$，因为对于任何 v 的增量J都会有 3 倍增量，而且这类似于我们在上一个视频中的例子，我们有$f(a)=3a$，然后我们推导出 $\frac{df(a)}{da}=3$，所以这里我们有$J=3v$，所以$\frac{dJ}{dv}=3$，这里J扮演了f的角色，在之前的视频里的例子。

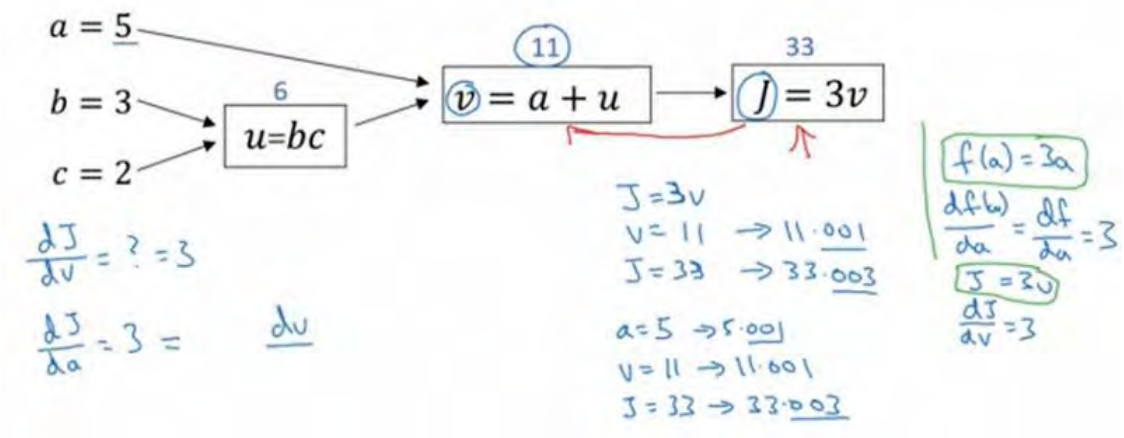
在反向传播算法中的术语，我们看到，如果你想计算最后输出变量的导数，使用你最关心的变量对v的导数，那么我们就做完了一步反向传播，在这个流程图中是一个反向步。



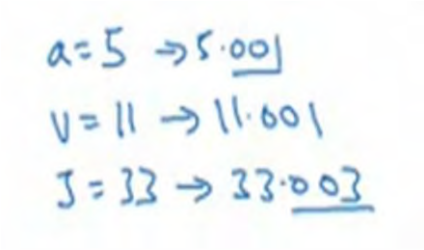
我们来看另一个例子，$\frac{\mathrm{dJ}}{da}$是多少呢？换句话说，如果我们提高 a 的数值，对 J 的数值

有什么影响？

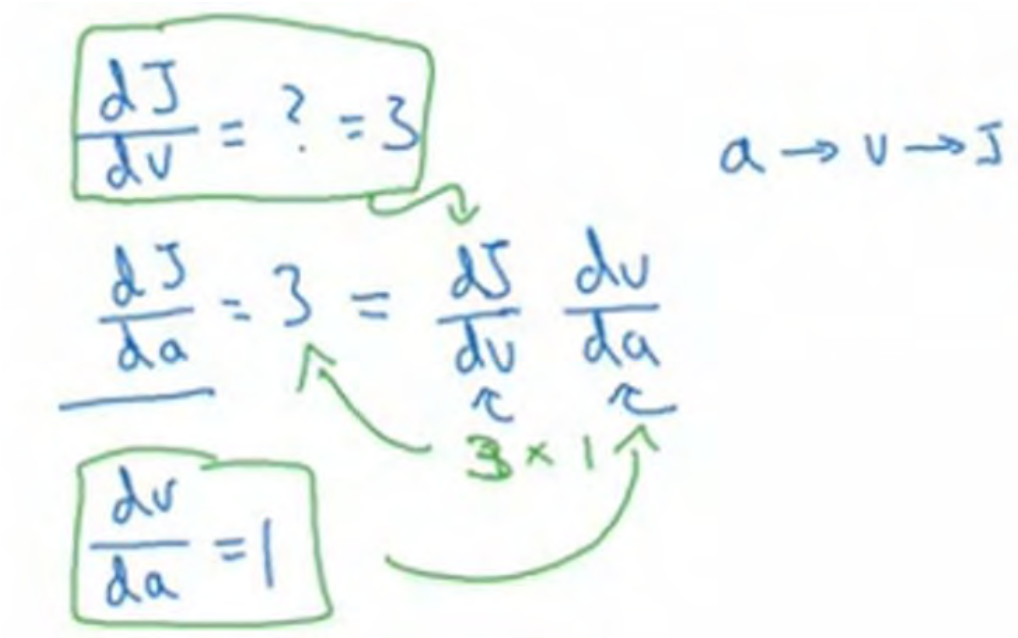
好，我们看看这个例子。变量$a = 5$，我们让它增加到 5.001，那么对v的影响就是$a + u$，之前$v = 11$，现在变成 11.001，我们从上面看到现在J 就变成 33.003 了，所以我们看到的是，如果你让a增加 0.001，J增加 0.003。那么增加a，我是说如果你把这个 5 换成某个新值，那么a的改变量就会传播到流程图的最右，所以J最后是 33.003。所以 J 的增量是 3 乘以a的增量，意味着这个导数是 3。



要解释这个计算过程，其中一种方式是：如果你改变了a，那么也会改变v，通过改变v，也会改变J，所以J值的净变化量，当你提升这个值（0.001），当你把a值提高一点点，这就是J的变化量（0.003）。

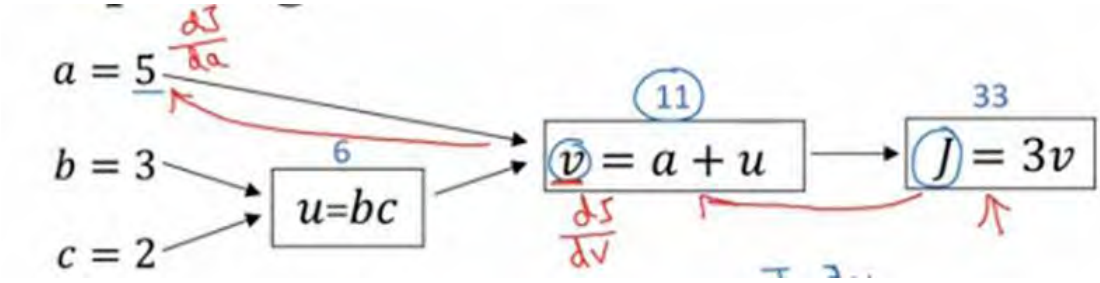


首先 a 增加了，v也会增加，v增加多少呢？这取决于$\frac{dv}{da}$，然后v的变化导致J也在增加，所以这在微积分里实际上叫链式法则，如果a影响到v，v影响到J，那么当你让a变大时，J 的变化量就是当你改变a时，v的变化量乘以改变v时J的变化量，在微积分里这叫链式法则。



我们从这个计算中看到，如果你让a增加 0.001，v也会变化相同的大小，所以$\frac{\mathrm{dv}}{da}=1$。事实上，如果你代入进去，我们之前算过$\frac{dJ}{dv}=3$，$\frac{dv}{da}=1$，所以这个乘积 3×1，实际上就给出了正确答案，$\frac{dJ}{da}=3$。

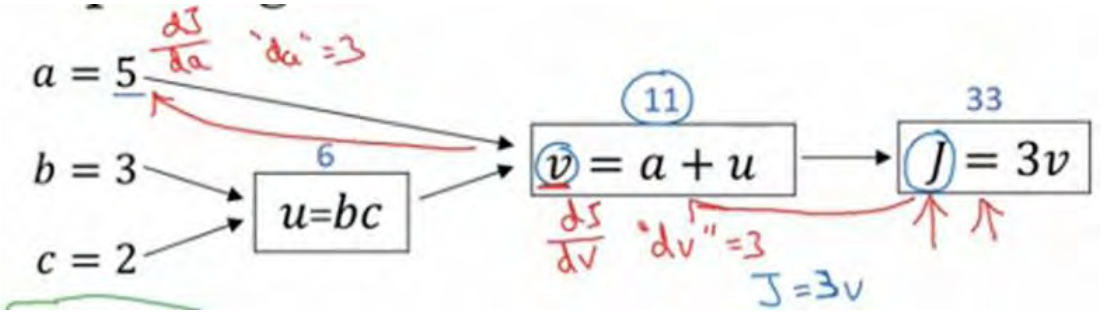
这张小图表示了如何计算，$\frac{dJ}{dv}$就是J对变量v的导数，它可以帮助你计算$\frac{dJ}{da}$，所以这是另一步反向传播计算。



现在我想介绍一个新的符号约定，当你编程实现反向传播时，通常会有一个最终输出值是你要关心的，最终的输出变量，你真正想要关心或者说优化的。在这种情况下最终的输出变量是 J，就是流程图里最后一个符号，所以有很多计算尝试计算输出变量的导数，所以输出变量对某个变量的导数，我们就用$dvar$命名，所以在很多计算中你需要计算最终输出结果的导数，在这个例子里是J，还有各种中间变量，比如a、b、c、u、v，当你在软件里实现的时候，变量名叫什么？你可以做的一件事是，在 **python** 中，你可以写一个很长的变量名，比如$dFinalOutputvar_dvar$，但这个变量名有点长，我们就用dJ_dvar，但因为你一直对dJ求导，对这个最终输出变量求导。我这里要介绍一个新符号，在程序里，当你编程的时候，

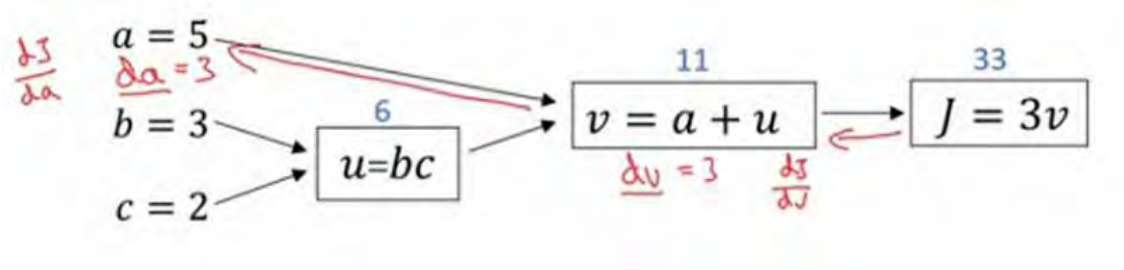
在代码里，我们就使用变量名$dvar$，来表示那个量。

好，所以在程序里是$dvar$表示导数，你关心的最终变量J的导数，有时最后是L，对代码中各种中间量的导数，所以代码里这个东西，你用dv表示这个值，所以$dv = 3$，你的代码表示就是$da = 3$。



好，所以我们通过这个流程图完成部分的后向传播算法。我们在下一张幻灯片看看这个例子剩下的部分。

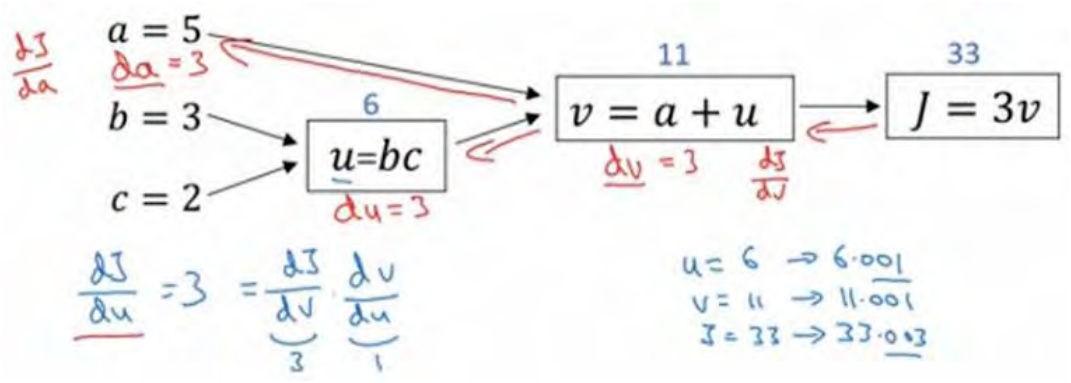
我们清理出一张新的流程图，我们回顾一下，到目前为止，我们一直在往回传播，并计算$dv = 3$，再次，dv是代码里的变量名，其真正的定义是$\frac{dJ}{dv}$。我发现$da = 3$，再次，da是代码里的变量名，其实代表$\frac{dJ}{da}$的值。



大概手算了一下，两条直线怎么计算反向传播。

好，我们继续计算导数，我们看看这个值u，那么$\frac{dJ}{du}$是多少呢？通过和之前类似的计算，现在我们从$u = 6$出发，如果你令 u 增加到 6.001，那么v之前是 11，现在变成 11.001 了，J就从 33 变成 33.003，所以J 增量是 3 倍，所以$\frac{\mathrm{dJ}}{du} = 3$。对$u$的分析很类似对 a 的分析，实际上这计算起来就是$\frac{dJ}{dv} \cdot \frac{dv}{du}$，有了这个，我们可以算出$\frac{\mathrm{dJ}}{dv} = 3$，$\frac{dv}{du} = 1$，最终算出结果是$3 \times 1 =$

3。

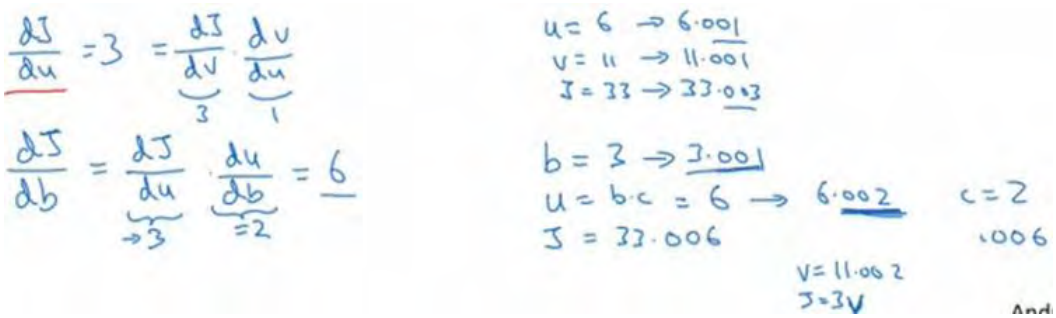


所以我们还有一步反向传播，我们最终计算出$du = 3$，这里的du当然了，就是$\frac{dJ}{du}$。

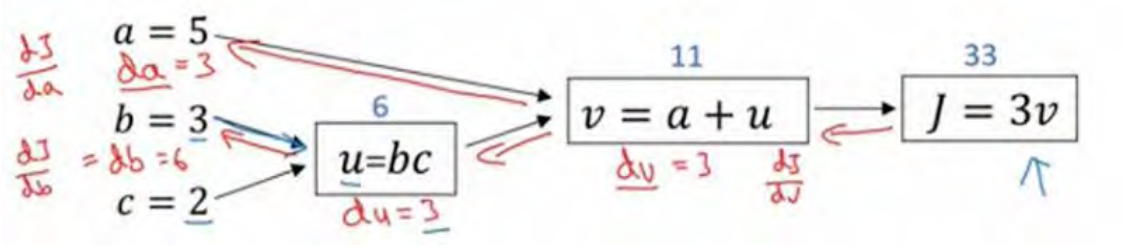
现在，我们仔细看看最后一个例子，那么$\frac{dJ}{db}$呢？想象一下，如果你改变了b的值，你想要然后变化一点，让 J 值到达最大或最小，那么导数是什么呢？这个J函数的斜率，当你稍微改变b值之后。事实上，使用微积分链式法则，这可以写成两者的乘积，就是$\frac{dJ}{du} \cdot \frac{du}{db}$，理由是，如果你改变$b$一点点，所以$b$变化比如说 3.001，它影响 J 的方式是，首先会影响 u，它对u的影响有多大？好，u的定义是$b \cdot c$，所以$b = 3$时这是 6，现在就变成 6.002 了，对吧，因为在我们的例子中$c = 2$，所以这告诉我们$\frac{du}{db} = 2$当你让 b 增加 0.001 时，u就增加两倍。所以$\frac{du}{db} = 2$，现在我想u的增加量已经是b的两倍，那么$\frac{\mathrm{dJ}}{du}$是多少呢？我们已经弄清楚了，这等于 3，所以让这两部分相乘，我们发现$\frac{dJ}{db} = 6$。

好，这就是第二部分的推导，其中我们想知道 u 增加 0.002，会对J 有什么影响。实际上$\frac{dJ}{du} = 3$，这告诉我们 u 增加 0.002 之后，J上升了 3 倍，那么J 应该上升 0.006，对吧。这可以从$\frac{dJ}{du} = 3$推导出来。

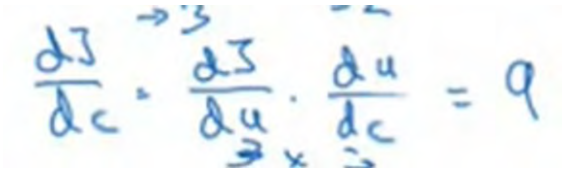
如果你仔细看看这些数学内容，你会发现，如果b变成 3.001，那么u就变成 6.002，v变成 11.002，然后$J = 3v = 33.006$，对吧？这就是如何得到$\frac{dJ}{db} = 6$。



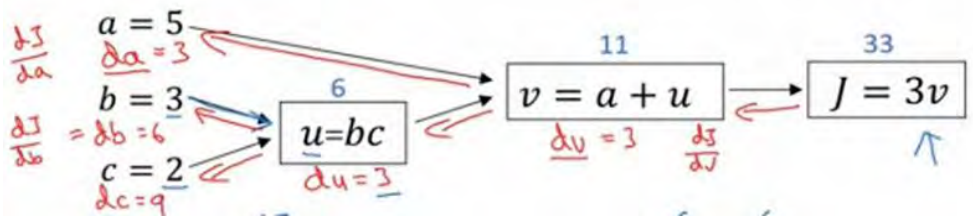
为了填进去，如果我们反向走的话，$db = 6$，而 db 其实是 **Python** 代码中的变量名，表示$\frac{dJ}{db}$。



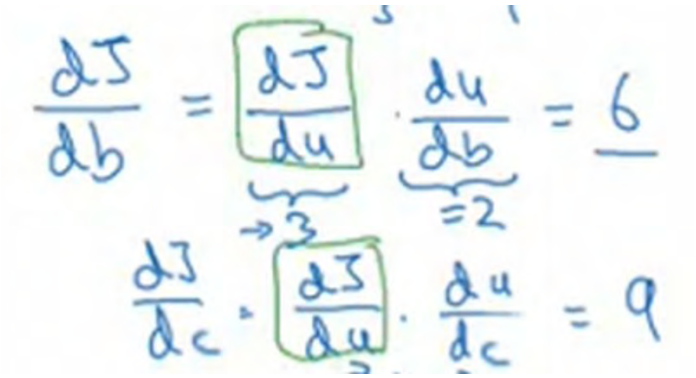
我不会很详细地介绍最后一个例子，但事实上，如果你计算$\frac{dJ}{dc} = \frac{dJ}{du} \cdot \frac{du}{dc} = 3 \times 3$，这个结果是 9。



我不会详细说明这个例子，在最后一步，我们可以推出$dc = 9$。



所以这个视频的要点是，对于那个例子，当计算所有这些导数时，最有效率的办法是从右到左计算，跟着这个红色箭头走。特别是当我们第一次计算对v的导数时，之后在计算对a导数就可以用到。然后对u的导数，比如说这个项和这里这个项：



可以帮助计算对b的导数，然后对c的导数。

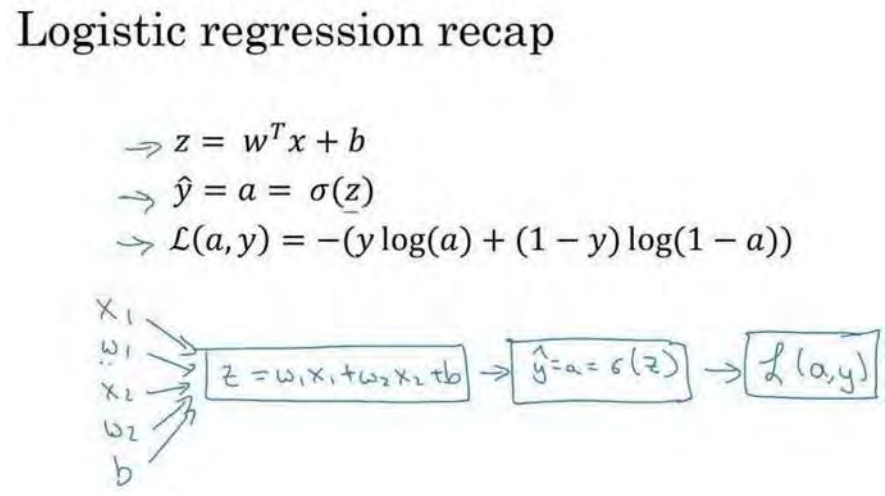
所以这是一个计算流程图，就是正向或者说从左到右的计算来计算成本函数 J，你可能需要优化的函数，然后反向从右到左计算导数。如果你不熟悉微积分或链式法则，我知道这里有些细节讲的很快，但如果你没有跟上所有细节，也不用怕。在下一个视频中，我会再过一遍。在逻辑回归的背景下过一遍，并给你介绍需要做什么才能编写代码，实现逻辑回归模型中的导数计算。

2.9 逻辑回归中的梯度下降（Logistic Regression Gradient Descent）

本节我们讨论怎样通过计算偏导数来实现逻辑回归的梯度下降算法。它的关键点是几个重要公式，其作用是用来实现逻辑回归中梯度下降算法。但是在本节视频中，我将使用计算图对梯度下降算法进行计算。我必须要承认的是，使用计算图来计算逻辑回归的梯度下降算法有点大材小用了。但是，我认为以这个例子作为开始来讲解，可以使你更好的理解背后的思想。从而在讨论神经网络时，你可以更深刻而全面地理解神经网络。接下来让我们开始学习逻辑回归的梯度下降算法。

假设样本只有两个特征x_1和x_2，为了计算z，我们需要输入参数w_1、w_2 和b，除此之外还有特征值x_1和x_2。因此z的计算公式为： $z = w_1x_1 + w_2x_2 + b$ 回想一下逻辑回归的公式定义如下： $\hat{y} = a = \sigma(z)$ 其中$z = w^Tx + b$ $\sigma(z) = \frac{1}{1+e^{-z}}$ 损失函数： $L(\hat{y}^{(i)}, y^{(i)}) = -y^{(i)}\log\hat{y}^{(i)} - (1-y^{(i)})\log(1-\hat{y}^{(i)})$ 代价函数：$J(w,b) = \frac{1}{m}\sum_i^m L(\hat{y}^{(i)}, y^{(i)})$ 假设现在只考虑单个样本的情况，单个样本的代价函数定义如下： $L(a,y) = -(y\log(a) + (1-y)\log(1-a))$ 其中a是逻辑回归的输出，y是样本的标签值。现在让我们画出表示这个计算的计算图。 这里先复习下梯度下降法，w和b的修正量可以表达如下：

$w := w - a\frac{\partial J(w,b)}{\partial w}$， $b := b - a\frac{\partial J(w,b)}{\partial b}$



如图：在这个公式的外侧画上长方形。然后计算： $\hat{y} = a = \sigma(z)$ 也就是计算图的下一步。最后计算损失函数$L(a,y)$。 有了计算图，我就不需要再写出公式了。因此，为了使得逻辑回归中最小化代价函数$L(a,y)$，我们需要做的仅仅是修改参数w和b的值。前面我们已

经讲解了如何在单个训练样本上计算代价函数的前向步骤。现在让我们来讨论通过反向计算出导数。 因为我们想要计算出的代价函数$L(a,y)$的导数，首先我们需要反向计算出代价函数$L(a,y)$关于a的导数，在编写代码时，你只需要用da 来表示$\frac{dL(a,y)}{da}$ 。 通过微积分得到：$\frac{dL(a,y)}{da}=-y/a+(1-y)/(1-a)$ 如果你不熟悉微积分，也不必太担心，我们会列出本课程涉及的所有求导公式。那么如果你非常熟悉微积分，我们鼓励你主动推导前面介绍的代价函数的求导公式，使用微积分直接求出$L(a,y)$关于变量a的导数。如果你不太了解微积分，也不用太担心。现在我们已经计算出da，也就是最终输出结果的导数。 现在可以再反向一步，在编写 Python 代码时，你只需要用dz来表示代价函数L关于z 的导数$\frac{dL}{dz}$，也可以写成$\frac{dL(a,y)}{dz}$，这两种写法都是正确的。$\frac{dL}{dz}=a-y$ 。因为$\frac{dL(a,y)}{dz}=\frac{dL}{dz}=(\frac{dL}{da})\cdot(\frac{da}{dz})$，并且$\frac{da}{dz}=a\cdot(1-a)$，而 $\frac{dL}{da}=(-\frac{y}{a}+\frac{(1-y)}{(1-a)})$， 因此将这两项相乘 $dz=\frac{dL(a,y)}{dz}=\frac{dL}{dz}=(\frac{dL}{da})\cdot(a(1-a)(-\frac{y}{a}+\frac{(1-y)}{(1-a)}))=a-y$ 视频中为了简化推导过程，假设n_x 这个推导的过程就是我之前提到过的链式法则。如果你对微积分熟悉，放心地去推导整个求导过程，如果不熟悉微积分，你只需要知道$dz=(a-y)$已经计算好了。

现在进行最后一步反向推导，也就是计算w和b变化对代价函数L的影响，特别地，可以用:

$dw_1=\frac{1}{m}\sum_i^m x_1^{(i)}(a^{(i)}-y^{(i)})$

$dw_2=\frac{1}{m}\sum_i^m x_2^{(i)}(a^{(i)}-y^{(i)})$

$db=\frac{1}{m}\sum_i^m(a^{(i)}-y^{(i)})$

视频中， dw_1 表示$\frac{\partial L}{\partial w_1}=x_1\cdot dz$， dw_2 表示$\frac{\partial L}{\partial w_2}=x_2\cdot dz$， $db=dz$。 因此，关于单个样本的梯度下降算法，你所需要做的就是如下的事情：

使用公式$dz=(a-y)$计算dz，

使用$dw_1=x_1\cdot dz$ 计算dw_1，

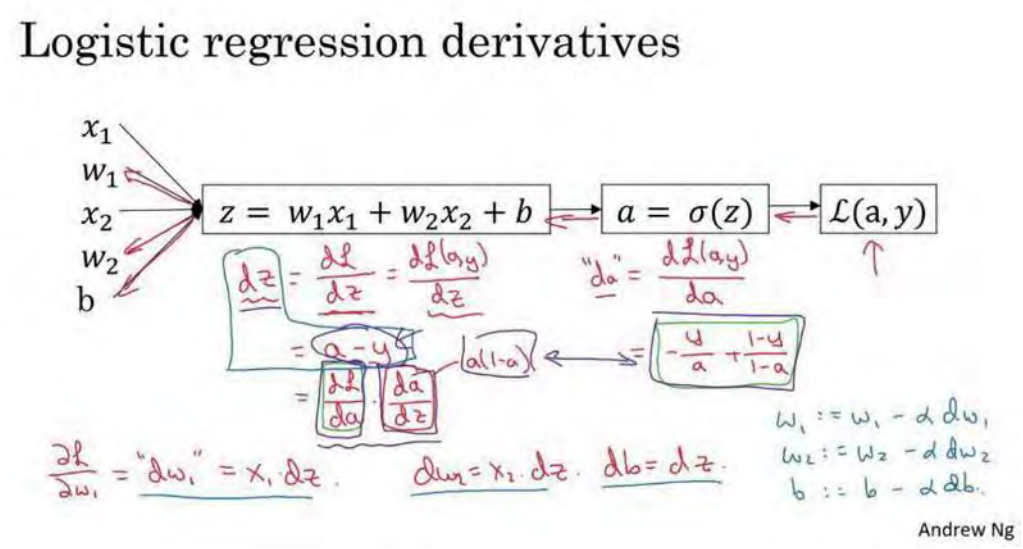
$dw_2=x_2\cdot dz$计算dw_2， $db=dz$ 来计算db，

然后: 更新$w_1=w_1-adw_1$，

更新$w_2=w_2-adw_2$，

更新$b=b-\alpha db$。

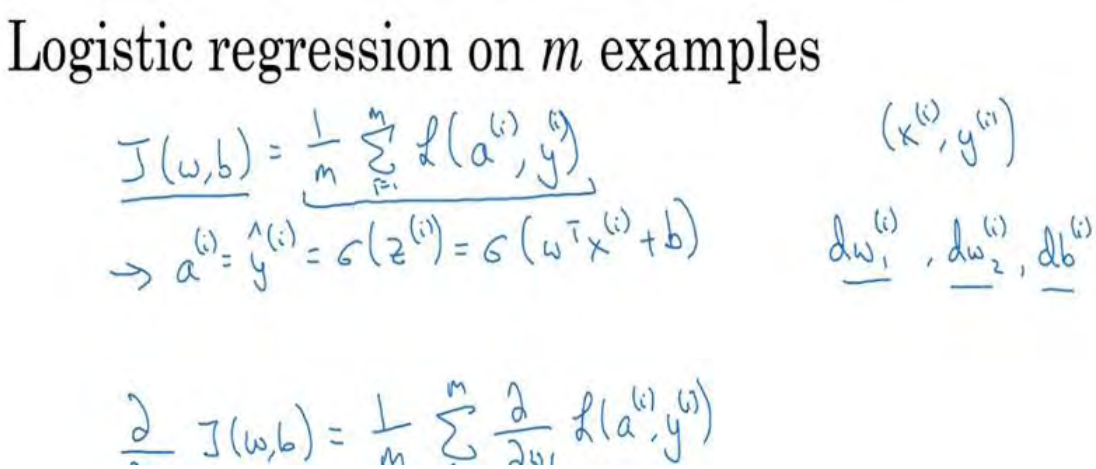
这就是关于单个样本实例的梯度下降算法中参数更新一次的步骤。



现在你已经知道了怎样计算导数，并且实现针对单个训练样本的逻辑回归的梯度下降算法。但是，训练逻辑回归模型不仅仅只有一个训练样本，而是有m个训练样本的整个训练集。因此在下一节视频中，我们将这些思想应用到整个训练样本集中，而不仅仅只是单个样本上。

2.10 m 个样本的梯度下降(Gradient Descent on m Examples)

在之前的视频中,你已经看到如何计算导数，以及应用梯度下降在逻辑回归的一个训练样本上。现在我们想要把它应用在m个训练样本上。



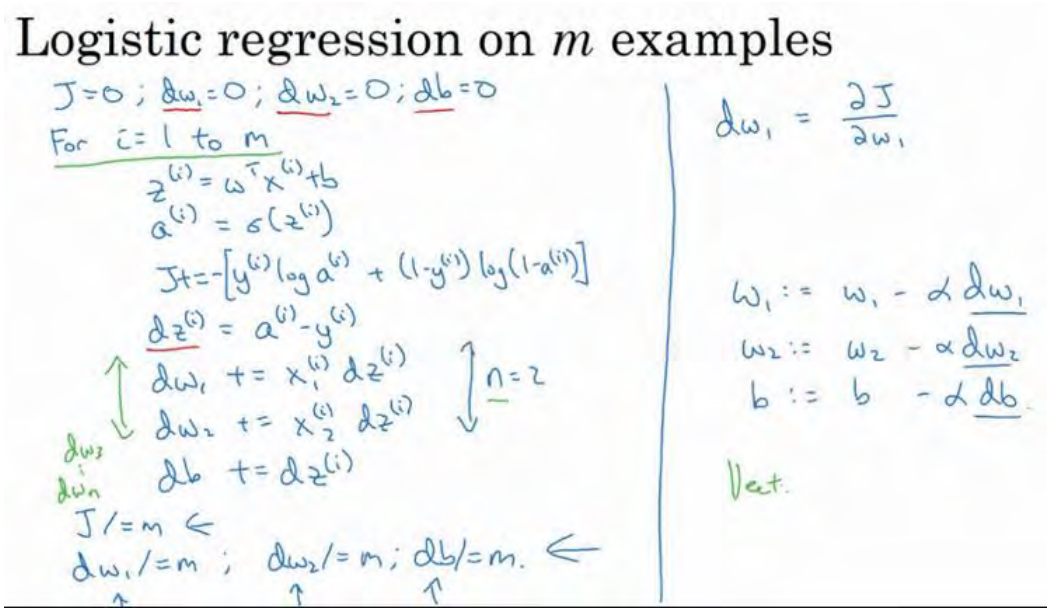
首先，让我们时刻记住有关于损失函数$J(w,b)$的定义。

$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L(a^{(i)},y^{(i)})$

当你的算法输出关于样本y的$a^{(i)}$，$a^{(i)}$是训练样本的预测值，即：

$\sigma(z^{(i)})=\sigma(w^Tx^{(i)}+b)$。

所以我们在前面的幻灯中展示的是对于任意单个训练样本，如何计算微分当你只有一个训练样本。因此dw_1，dw_2和db 添上上标i表示你求得的相应的值。如果你面对的是我们在之前的幻灯中演示的那种情况，但只使用了一个训练样本$(x^{(i)},y^{(i)})$。现在你知道带有求和的全局代价函数，实际上是 1 到m项各个损失的平均。 所以它表明全局代价函数对w_1的微分，对w_1的微分也同样是各项损失对w_1微分的平均。



但之前我们已经演示了如何计算这项，即之前幻灯中演示的如何对单个训练样本进行计算。所以你真正需要做的是计算这些微分，如我们在之前的训练样本上做的。并且求平均，这会给你全局梯度值，你能够把它直接应用到梯度下降算法中。

所以这里有很多细节，但让我们把这些装进一个具体的算法。同时你需要一起应用的就是逻辑回归和梯度下降。

我们初始化 $J=0, dw_1=0, dw_2=0, db=0$

代码流程：

```
J=0;dw1=0;dw2=0;db=0;
for i = 1 to m
    z(i) = wx(i)+b;
    a(i) = sigmoid(z(i));
    J += -[y(i)log(a(i))+(1-y(i)) log(1-a(i));
    dz(i) = a(i)-y(i);
    dw1 += x1(i)dz(i);
    dw2 += x2(i)dz(i);
    db += dz(i);
J/= m;
dw1/= m;
dw2/= m;
db/= m;
```

```
w=w-alpha*dw
b=b-alpha*db
```

幻灯片上只应用了一步梯度下降。因此你需要重复以上内容很多次，以应用多次梯度下降。看起来这些细节似乎很复杂，但目前不要担心太多。希望你明白，当你继续尝试并应用这些在编程作业里，所有这些会变的更加清楚。

但这种计算中有两个缺点，也就是说应用此方法在逻辑回归上你需要编写两个 **for** 循环。第一个 **for** 循环是一个小循环遍历m个训练样本，第二个 **for** 循环是一个遍历所有特征的 **for** 循环。这个例子中我们只有 2 个特征，所以n等于 2 并且n_x 等于 2。 但如果你有更多特征，你开始编写你的因此dw_1，dw_2，你有相似的计算从dw_3一直下去到dw_n。所以看来你需要一个 **for** 循环遍历所有n个特征。

当你应用深度学习算法，你会发现在代码中显式地使用 **for** 循环使你的算法很低效，同时在深度学习领域会有越来越大的数据集。所以能够应用你的算法且没有显式的 **for** 循环会是重要的，并且会帮助你适用于更大的数据集。所以这里有一些叫做向量化技术,它可以允许你的代码摆脱这些显式的 **for** 循环。

我想在先于深度学习的时代，也就是深度学习兴起之前，向量化是很棒的。可以使你有时候加速你的运算，但有时候也未必能够。但是在深度学习时代向量化，摆脱 for 循环已经变得相当重要。因为我们越来越多地训练非常大的数据集，因此你真的需要你的代码变得非常高效。所以在接下来的几个视频中，我们会谈到向量化，以及如何应用向量化而连一个 **for** 循环都不使用。所以学习了这些，我希望你有关于如何应用逻辑回归，或是用于逻辑回归的梯度下降，事情会变得更加清晰。当你进行编程练习，但在真正做编程练习之前让我们先谈谈向量化。然后你可以应用全部这些东西，应用一个梯度下降的迭代而不使用任何 **for** 循环。

2.11 向量化(Vectorization)

向量化是非常基础的去除代码中 **for** 循环的艺术，在深度学习安全领域、深度学习实践中，你会经常发现自己训练大数据集，因为深度学习算法处理大数据集效果很棒，所以你的代码运行速度非常重要，否则如果在大数据集上，你的代码可能花费很长时间去运行，你将要等待非常长的时间去得到结果。所以在深度学习领域，运行向量化是一个关键的技巧，让我们举个栗子说明什么是向量化。

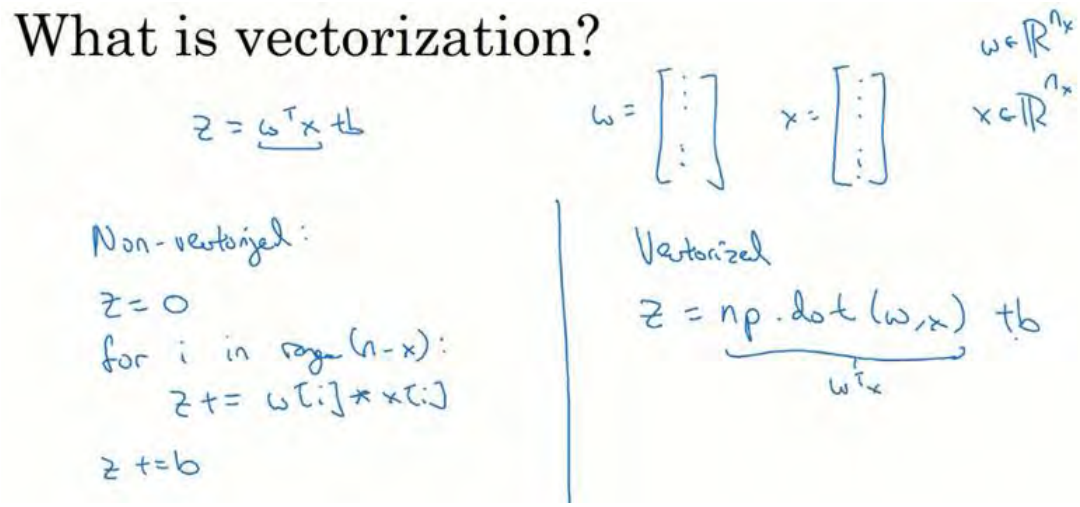
在逻辑回归中你需要去计算$z = w^T x + b$，w、x都是列向量。如果你有很多的特征那么就会有一个非常大的向量，所以$w \in \mathbb{R}^{n_x}$ ，$x \in \mathbb{R}^{n_x}$，所以如果你想使用非向量化方法去计算$w^T x$，你需要用如下方式（**python**）

```
z=0
for i in range(n_x)
z+=w[i]*x[i]
z+=b
```

这是一个非向量化的实现，你会发现这真的很慢，作为一个对比，向量化实现将会非常直接计算$w^T x$，代码如下：

```
z=np.dot(w,x)+b
```

这是向量化计算$w^T x$的方法，你将会发现这个非常快



让我们用一个小例子说明一下，在我的我将会写一些代码（以下为教授在他的 **Jupyter notebook** 上写的 **Python** 代码，）

```
import numpy as np #导入 numpy 库
```

```
a = np.array([1,2,3,4]) #创建一个数据 a
print(a)
# [1 2 3 4]
import time #导入时间库
a = np.random.rand(1000000)
b = np.random.rand(1000000) #通过 round 随机得到两个一百万维度的数组
tic = time.time() #现在测量一下当前时间
#向量化的版本
c = np.dot(a,b)
toc = time.time()
print("Vectorized version:" + str(1000*(toc-tic)) +"ms") #打印一下向量化的版本的时间

#继续增加非向量化的版本
c = 0
tic = time.time()
for i in range(1000000):
c += a[i]*b[i]
toc = time.time()
print(c)
print("For loop:" + str(1000*(toc-tic)) + "ms")#打印 for 循环的版本的时间
```

返回值见图。

在两个方法中，向量化和非向量化计算了相同的值，如你所见，向量化版本花费了 1.5 毫秒，非向量化版本的 **for** 循环花费了大约几乎 500 毫秒，非向量化版本多花费了 300 倍时间。所以在这个例子中，仅仅是向量化你的代码，就会运行 300 倍快。这意味着如果向量化方法需要花费一分钟去运行的数据，for 循环将会花费 5 个小时去运行。

一句话总结，以上都是再说和 for 循环相比，向量化可以快速得到结果。

你可能听过很多类似如下的话，“大规模的深度学习使用了 **GPU** 或者图像处理单元实现”，但是我做的所有的案例都是在 **jupyter notebook** 上面实现，这里只有 **CPU**，**CPU** 和 **GPU**

都有并行化的指令，他们有时候会叫做 **SIMD** 指令，这个代表了一个单独指令多维数据，这个的基础意义是，如果你使用了 **built-in** 函数,像 `np.function` 或者并不要求你实现循环的函数，它可以让 **python** 的充分利用并行化计算，这是事实在 **GPU** 和 **CPU** 上面计算，**GPU** 更加擅长 **SIMD** 计算，但是 **CPU** 事实上也不是太差，可能没有 **GPU** 那么擅长吧。接下来的视频中，你将看到向量化怎么能够加速你的代码，经验法则是，无论什么时候，避免使用明确的 **for** 循环。

以下代码及运行结果截图：

In [1]:
```
import numpy as np

a = np.array([1,2,3,4])
print(a)
```

```
[1 2 3 4]
```

In [13]:
```
import time

a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = np.dot(a,b)
toc = time.time()

print("Vectorized version:" + str(1000*(toc-tic)) +"ms")
```

```
import time

a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized version:" + str(1000*(toc-tic)) +"ms")

c = 0
tic = time.time()
for i in range(1000000):
    c += a[i]*b[i]
toc = time.time()

print(c)
print("For loop:" + str(1000*(toc-tic)) + "ms")
```

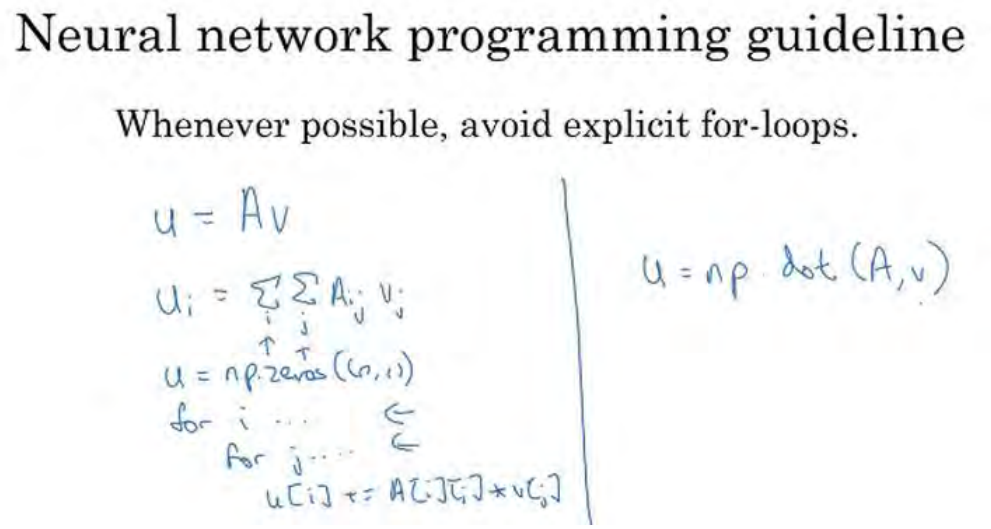
```
250286.989866
Vectorized version:1.5027523040771484ms
250286.989866
For loop:474.29513931274414ms
```

2.12 向量化的更多例子（More Examples of Vectorization）

从上节视频中，你知道了怎样通过 **numpy** 内置函数和避开显式的循环(**loop**)的方式进行向量化，从而有效提高代码速度。

经验提醒我，当我们在写神经网络程序时，或者在写逻辑(**logistic**)回归，或者其他神经网络模型时，应该避免写循环(**loop**)语句。虽然有时写循环(**loop**)是不可避免的，但是我们可以使用比如 **numpy** 的内置函数或者其他办法去计算。当你这样使用后，程序效率总是快于循环(**loop**)。

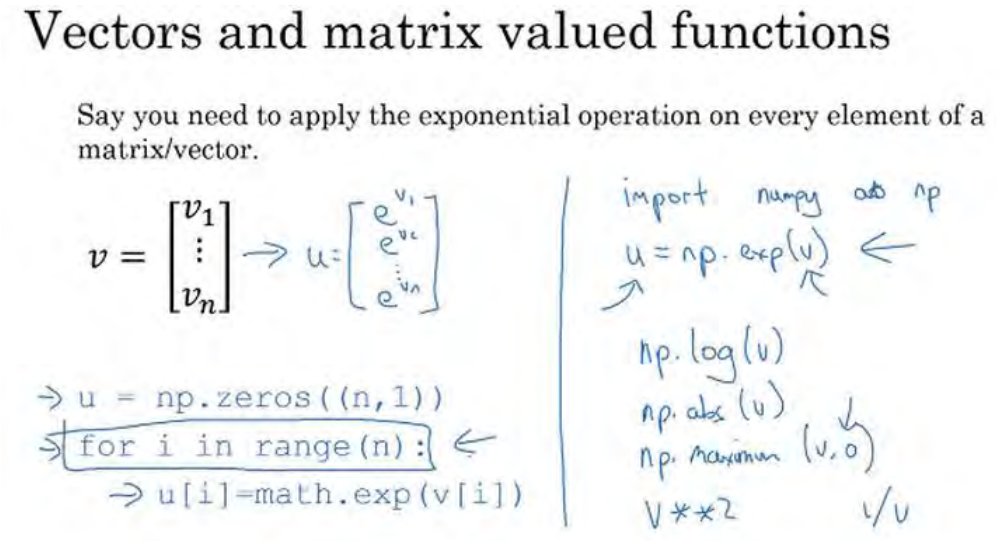
让我们看另外一个例子。如果你想计算向量$u=Av$，这时矩阵乘法定义为，矩阵乘法的定义就是：$u_i=\sum_j A_{ij}v_i$，这取决于你怎么定义u_i值。同样使用非向量化实现，$u=np.zeros(n,1)$，并且通过两层循环$for(i):for(j):$，得到$u[i]=u[i]+A[i][j]*v[j]$。现在就有了i 和 j 的两层循环，这就是非向量化。向量化方式就可以用$u=np.dot(A,v)$，右边这种向量化实现方式，消除了两层循环使得代码运行速度更快。



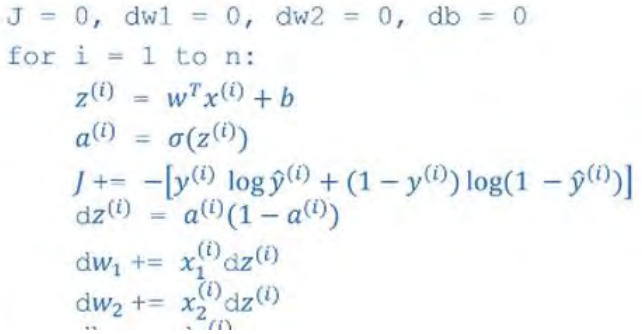
下面通过另一个例子继续了解向量化。如果你已经有一个向量v，并且想要对向量v的每个元素做指数操作，得到向量u等于e的v_1，e的v_2，一直到e的v_n次方。这里是非向量化的实现方式，首先你初始化了向量$u=np.zeros(n,1)$，并且通过循环依次计算每个元素。但事实证明可以通过 **python** 的 **numpy** 内置函数，帮助你计算这样的单个函数。所以我会引入 `import numpy as np`，执行 $u=np.exp(v)$ 命令。注意到，在之前有循环的代码中，这里仅用了一行代码，向量v作为输入，u作为输出。你已经知道为什么需要循环，并且通过右边代码实现，效率会明显的快于循环方式。

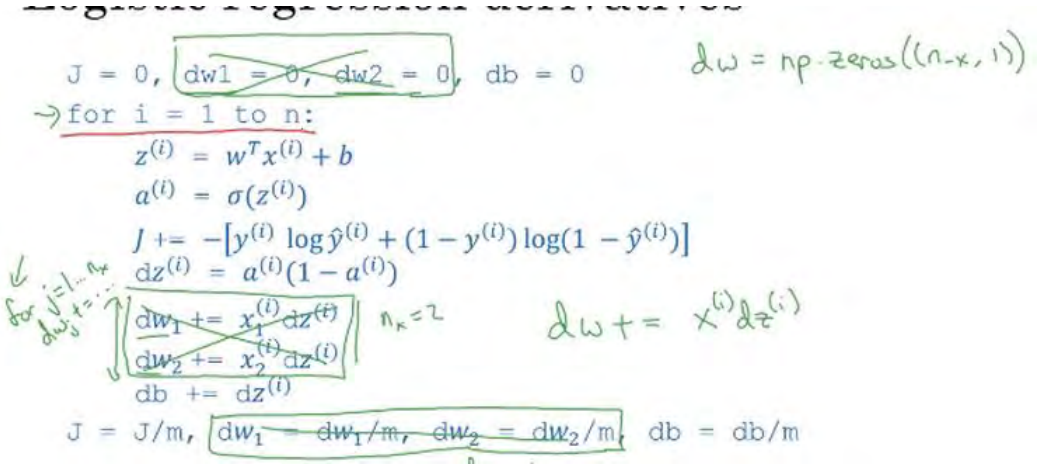
事实上，**numpy** 库有很多向量函数。比如 `u=np.log` 是计算对数函数(log)、`np.abs()`

是计算数据的绝对值、np.maximum() 计算元素 y 中的最大值，你也可以 np.maximum(v,0) 、$v**2$ 代表获得元素 y 每个值得平方、$\frac{1}{v}$ 获取元素 y 的逆等等。所以当你想写循环时候，检查 **numpy** 是否存在类似的内置函数，从而避免使用循环(**loop**)方式。



那么，将刚才所学到的内容，运用在逻辑回归的梯度下降上，看看我们是否能简化两个计算过程中的某一步。这是我们逻辑回归的求导代码，有两层循环。在这例子我们有n个特征值。如果你有超过两个特征时，需要循环 dw_1 、dw_2 、dw_3 等等。所以 j 的实际值是 1、2 和 x，就是你想要更新的值。所以我们想要消除第二循环，在这一行，这样我们就不用初始化 dw_1 ， dw_2 都等于 0。去掉这些，而是定义 dw 为一个向量，设置 $u = np.zeros(n(x),1)$。定义了一个x行的一维向量，从而替代循环。我们仅仅使用了一个向量操作 $dw = dw + x^{(i)}dz^{(i)}$ 。最后，我们得到 $dw = dw/m$ 。现在我们通过将两层循环转成一层循环，我们仍然还有这个循环训练样本。





希望这个视频给了你一点向量化感觉，减少一层循环使你代码更快，但事实证明我们能做得更好。所以在下个视频，我们将进一步的讲解逻辑回归，你将会看到更好的监督学习结果。在训练中不需要使用任何 **for** 循环，你也可以写出代码去运行整个训练集。到此为止一切都好，让我们看下一个视频。

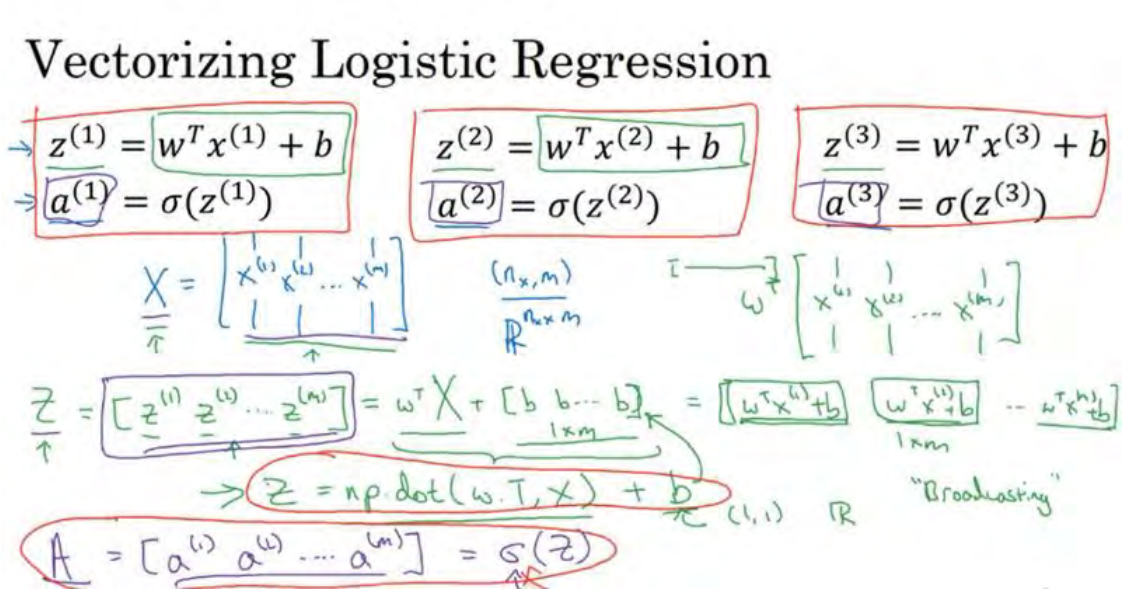
2.13 向量化逻辑回归(Vectorizing Logistic Regression)

我们已经讨论过向量化是如何显著加速你的代码，在本次视频中我们将讨论如何实现逻辑回归的向量化计算。这样就能处理整个数据集，甚至不会用一个明确的 for 循环就能实现对于整个数据集梯度下降算法的优化。我对这项技术感到非常激动，并且当我们后面谈到神经网络时同样也不会用到一个明确的 for 循环。

让我们开始吧，首先我们回顾一下逻辑回归的前向传播步骤。所以，如果你有 m 个训练样本，然后对第一个样本进行预测，你需要这样计算。计算 z，我正在使用这个熟悉的公式 $z^{(1)} = w^T x^{(1)} + b$ 。然后计算激活函数 $a^{(1)} = \sigma(z^{(1)})$ ，计算第一个样本的预测值 y 。

然后对第二个样本进行预测，你需要计算 $z^{(2)} = w^T x^{(2)} + b$ ， $a^{(2)} = \sigma(z^{(2)})$ 。然后对第三个样本进行预测，你需要计算 $z^{(3)} = w^T x^{(3)} + b$ ， $a^{(3)} = \sigma(z^{(3)})$ ，依次类推。如果你有 m 个训练样本，你可能需要这样做 m 次，可以看出，为了完成前向传播步骤，即对我们的 m 个样本都计算出预测值。有一个办法可以并且不需要任何一个明确的 **for** 循环。让我们来看一下你该怎样做。

首先，回忆一下我们曾经定义了一个矩阵 X 作为你的训练输入，(如下图中蓝色 X)像这样在不同的列中堆积在一起。这是一个 n_x 行 m 列的矩阵。我现在将它写为 **Python numpy** 的形式 (n_x, m) ，这只是表示 X 是一个 n_x 乘以 m 的矩阵 $R^{n_x \times m}$。



现在我首先想做的是告诉你该如何在一个步骤中计算 z_1、 z_2 、z_3 等等。实际上，只用了一行代码。所以，我打算先构建一个 $1 \times m$ 的矩阵，实际上它是一个行向量，同时我准备计算 $z^{(1)}$， $z^{(2)}$ ……一直到 $z^{(m)}$ ，所有值都是在同一时间内完成。结果发现它可以表

达为 w 的转置乘以大写矩阵 x 然后加上向量 $[bb\dots b]$ ， $([z^{(1)}z^{(2)}\dots z^{(m)}] = w^T + [bb\dots b])$ 。$[bb\dots b]$ 是一个 $1\times m$ 的向量或者 $1\times m$ 的矩阵或者是一个 m 维的行向量。所以希望你熟悉矩阵乘法，你会发现的 w 转置乘以 $x^{(1)}$ ， $x^{(2)}$ 一直到 $x^{(m)}$ 。所以 w 转置可以是一个行向量。所以第一项 w^TX 将计算 w 的转置乘以 $x^{(1)}$， w 转置乘以$x^{(2)}$ 等等。然后我们加上第二项 $[bb\dots b]$ ，你最终将 b 加到了每个元素上。所以你最终得到了另一个 $1\times m$ 的向量， $[z^{(1)}z^{(2)}\dots z^{(m)}] = w^TX + [bb\dots b] = [w^Tx^{(1)} + b, w^Tx^{(2)} + b\dots w^Tx^{(m)} + b]$ 。

$w^Tx^{(1)} + b$ 这是第一个元素，$w^Tx^{(2)} + b$ 这是第二个元素， $w^Tx^{(m)} + b$ 这是第 m 个元素。

如果你参照上面的定义，第一个元素恰好是 $z^{(1)}$ 的定义，第二个元素恰好是 $z^{(1)}$ 的定义，等等。所以，因为X是一次获得的，当你得到你的训练样本，一个一个横向堆积起来，这里我将 $[z^{(1)}z^{(2)}\dots z^{(m)}]$ 定义为大写的 Z ，你用小写 z 表示并将它们横向排在一起。所以当你将不同训练样本对应的小写 x 横向堆积在一起时得到大写变量 X 并且将小写变量也用相同方法处理，将它们横向堆积起来，你就得到大写变量 Z 。结果发现，为了计算 $W^TX + [bb\dots b]$ ，**numpy** 命令是$Z = np.dot(w.T, X) + b$。这里在 **Python** 中有一个巧妙的地方，这里 b 是一个实数，或者你可以说是一个 1×1 矩阵，只是一个普通的实数。但是当你将这个向量加上这个实数时，**Python** 自动把这个实数 b 扩展成一个 $1\times m$ 的行向量。所以这种情况下的操作似乎有点不可思议，它在 **Python** 中被称作广播**(brosdcasting)**，目前你不用对此感到顾虑，我们将在下一个视频中进行进一步的讲解。话说回来它只用一行代码，用这一行代码，你可以计算大写的 Z，而大写 Z 是一个包含所有小写$z^{(1)}$ 到 $z^{(m)}$ 的 $1\times m$ 的矩阵。这就是 Z 的内容，关于变量 a 又是如何呢？

我们接下来要做的就是找到一个同时计算 $[a^{(1)}a^{(2)}\dots a^{(m)}]$ 的方法。就像把小写 x 堆积起来得到大写 X 和横向堆积小写 z 得到大写 Z 一样，堆积小写变量 a 将形成一个新的变量，我们将它定义为大写 A。在编程作业中，你将看到怎样用一个向量在 **sigmoid** 函数中进行计算。所以 **sigmoid** 函数中输入大写 Z 作为变量并且非常高效地输出大写 A。你将在编程作业中看到它的细节。

总结一下，在这张幻灯片中我们已经看到，不需要 **for** 循环，利用 m 个训练样本一次性计算出小写 z 和小写 a，用一行代码即可完成。

```
Z = np.dot(w.T,X) + b
```

这一行代码：$A = [a^{(1)}a^{(2)}\dots a^{(m)}] = \sigma(Z)$ ，通过恰当地运用σ一次性计算所有 a。这

就是在同一时间内你如何完成一个所有 m 个训练样本的前向传播向量化计算。

概括一下，你刚刚看到如何利用向量化在同一时间内高效地计算所有的激活函数的所有a值。接下来，可以证明，你也可以利用向量化高效地计算反向传播并以此来计算梯度。让我们在下一个视频中看该如何实现。

2.14 向量化 logistic 回归的梯度输出（Vectorizing Logistic Regression's Gradient）

注：本节中大写字母代表向量，小写字母代表元素

如何向量化计算的同时，对整个训练集预测结果a，这是我们之前已经讨论过的内容。在本次视频中我们将学习如何向量化地计算m个训练数据的梯度，本次视频的重点是如何**同时**计算 m 个数据的梯度，并且实现一个非常高效的逻辑回归算法**(Logistic Regression)**。

之前我们在讲梯度计算的时候，列举过几个例子， $dz^{(1)} = a^{(1)} - y^{(1)}$，$dz^{(2)} = a^{(2)} - y^{(2)}$ ……等等一系列类似公式。现在，对 m个训练数据做同样的运算，我们可以定义一个新的变量 $dZ = [dz^{(1)}, dz^{(2)} \dots dz^{(m)}]$ ，所有的 dz 变量横向排列，因此，dZ 是一个 $1 \times m$ 的矩阵，或者说，一个 m 维行向量。在之前的幻灯片中，我们已经知道如何计算A，即$[a^{(1)}, a^{(2)} \dots a^{(m)}]$,我们需要找到这样的一个行向量 $Y = [y^{(1)} y^{(2)} \dots y^{(m)}]$ ，由此，我们可以这样计算 $dZ = A - Y = [a^{(1)} - y^{(1)} a^{(2)} - y^{(2)} \dots a^{(m)} - y^{(m)}]$，不难发现第一个元素就是$dz^{(1)}$，第二个元素就是 $dz^{(2)}$ ……所以我们现在仅需一行代码，就可以同时完成这所有的计算。

在之前的实现中，我们已经去掉了一个 **for** 循环，但我们仍有一个遍历训练集的循环，如下所示：

$$dw = 0$$

$$dw += x^{(1)} * dz^{(1)}$$

$$dw += x^{(2)} * dz^{(2)}$$

……

$$dw += x^{(m)} * dz^{(m)}$$

$$dw = \frac{dw}{m}$$

$$db = 0$$

$$db += dz^{(1)}$$

$$db += dz^{(2)}$$

……

$$db += dz^{(m)}$$

$$db = \frac{db}{m}$$

上述（伪）代码就是我们在之前实现中做的，我们已经去掉了一个 for 循环，但用上述方法计算 dw 仍然需要一个循环遍历训练集，我们现在要做的就是将其向量化！

首先我们来看 db，不难发现 $db = \frac{1}{m}\sum_{i=1}^{m} d\,z^{(i)}$ ，之前的讲解中，我们知道所有的$dz^{i)}$已经组成一个行向量 dZ了，所以在 Python 中，我们很容易地想到$db = \frac{1}{m} * np.sum(dZ)$；接下来看$dw$，我们先写出它的公式 $dw = \frac{1}{m} * X * dz^T$ 其中，X 是一个行向量。因此展开后 $dw = \frac{1}{m} * (x^{(1)}dz^{(1)} + x^{(2)}dz^{(2)} + \ldots + x^{m}dz^{m})$ 。因此我们可以仅用两行代码进行计算：$db = \frac{1}{m} * np.sum(dZ)$， $dw = \frac{1}{m} * X * dz^T$。这样，我们就避免了在训练集上使用 for 循环。

现在，让我们回顾一下，看看我们之前怎么实现的逻辑回归，可以发现，没有向量化是非常低效的，如下图所示代码：

$J = 0,\ dw_1 = 0,\ dw_2 = 0,\ db = 0$

for i = 1 to m:

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J\ += -\left[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})\right]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1\ += x_1^{(i)} dz^{(i)}$

$dw_2\ += x_2^{(i)} dz^{(i)}$

$db\ += dz^{(i)}$

$J = J/m,\ dw_1 = dw_1/m,\ dw_2 = dw_2/m$

$db = db/m$

我们的目标是不使用 **for** 循环，而是向量，我们可以这么做：

$Z = w^T X + b = np.dot(w.T, X) + b$

$A = \sigma(Z)$

$dZ = A - Y$

$$dw = \frac{1}{m} * X * dz^T$$

$$db = \frac{1}{m} * np.sum(dZ)$$

$w := w - a * dw$

$b := b - a * db$

现在我们利用前五个公式完成了前向和后向传播，也实现了对所有训练样本进行预测和求导，再利用后两个公式，梯度下降更新参数。我们的目的是不使用 **for** 循环，所以我们就通过一次迭代实现一次梯度下降，但如果你希望多次迭代进行梯度下降，那么仍然需要 **for** 循环，放在最外层。不过我们还是觉得一次迭代就进行一次梯度下降，避免使用任何循环比较舒服一些。

最后，我们得到了一个高度向量化的、非常高效的逻辑回归的梯度下降算法，我们将在下次视频中讨论 **Python** 中的 **Broadcasting** 技术。

2.15 Python 中的广播（Broadcasting in Python）

Calories from Carbs, Proteins, Fats in 100g of different foods:

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

这是一个不同食物(每 100g)中不同营养成分的卡路里含量表格，表格为 3 行 4 列，列表示不同的食物种类，从左至右依次为苹果，牛肉，鸡蛋，土豆。行表示不同的营养成分，从上到下依次为碳水化合物，蛋白质，脂肪。

那么，我们现在想要计算不同食物中不同营养成分中的卡路里百分比。

现在计算苹果中的碳水化合物卡路里百分比含量，首先计算苹果（100g）中三种营养成分卡路里总和 56+1.2+1.8 = 59，然后用 56/59 = 94.9%算出结果。

可以看出苹果中的卡路里大部分来自于碳水化合物，而牛肉则不同。

对于其他食物，计算方法类似。首先，按列求和，计算每种食物中（100g）三种营养成分总和，然后分别用不用营养成分的卡路里数量除以总和，计算百分比。

那么，能否不使用 for 循环完成这样的一个计算过程呢？

假设上图的表格是一个 4 行 3 列的矩阵A，记为 $A_{3\times4}$，接下来我们要使用 **Python** 的 **numpy** 库完成这样的计算。我们打算使用两行代码完成，第一行代码对每一列进行求和，第二行代码分别计算每种食物每种营养成分的百分比。

在 **jupyter notebook** 中输入如下代码，按 shift+Enter 运行，输出如下。

In [6]:
```
import numpy as np

A = np.array([[56.0, 0.0, 4.4, 68.0],
              [1.2, 104.0, 52.0, 8.0],
              [1.8, 135.0, 99.0, 0.9]])

print(A)
```

```
[[  56.     0.     4.4   68. ]
 [   1.2  104.    52.     8. ]
 [   1.8  135.    99.     0.9]]
```

下面使用如下代码计算每列的和，可以看到输出是每种食物(100g)的卡路里总和。

```
In [7]:  1 cal = A.sum(axis=0)
         2 print(cal)
```

```
[ 59.   239.   155.4   76.9]
```

其中 sum 的参数 axis=0 表示求和运算按列执行，之后会详细解释。

接下来计算百分比，这条指令将 3×4的矩阵A除以一个1×4的矩阵，得到了一个 3×4的结果矩阵，这个结果矩阵就是我们要求的百分比含量。

```
In [8]:  1 percentage = 100*A/cal.reshape(1,4)
         2 print(percentage)
```

```
[[ 94.91525424   0.          2.83140283  88.42652796]
 [  2.03389831  43.51464435  33.46203346  10.40312094]
 [  3.05084746  56.48535565  63.70656371   1.17035111]]
```

下面再来解释一下 A.sum(axis = 0)中的参数 axis。**axis 用来指明将要进行的运算是沿着哪个轴执行，在 numpy 中，0 轴是垂直的，也就是列，而 1 轴是水平的，也就是行。**

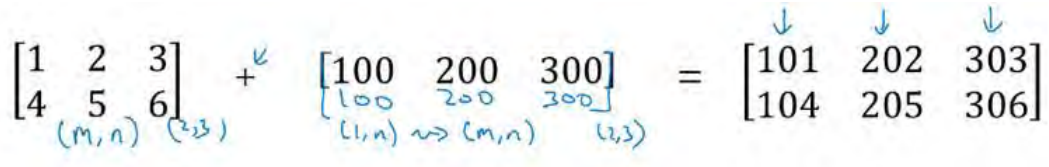
而第二个 A/cal.reshape(1,4)指令则调用了 **numpy** 中的广播机制。这里使用 3×4的矩阵A除以 1×4的矩阵cal。技术上来讲，其实并不需要再将矩阵cal reshape(重塑)成1×4，因为矩阵cal本身已经是 1×4了。但是当我们写代码时不确定矩阵维度的时候，通常会对矩阵进行重塑来确保得到我们想要的列向量或行向量。重塑操作 reshape 是一个常量时间的操作，时间复杂度是$O(1)$，它的调用代价极低。

那么一个 3×4 的矩阵是怎么和 1×4的矩阵做除法的呢？让我们来看一些更多的广播的例子。

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} 100 = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$$

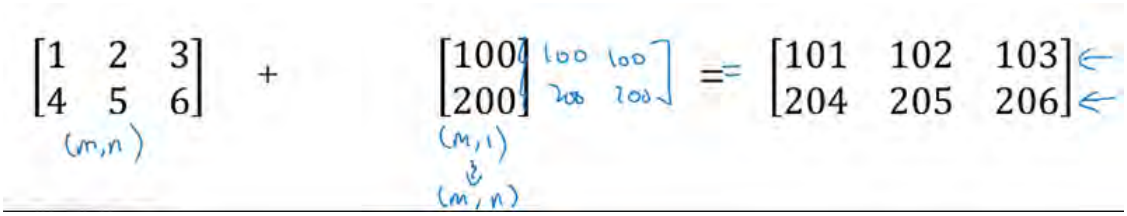
在 numpy 中，当一个 4×1的列向量与一个常数做加法时，实际上会将常数扩展为一个4×1的列向量，然后两者做逐元素加法。结果就是右边的这个向量。这种广播机制对于行向量和列向量均可以使用。

再看下一个例子。



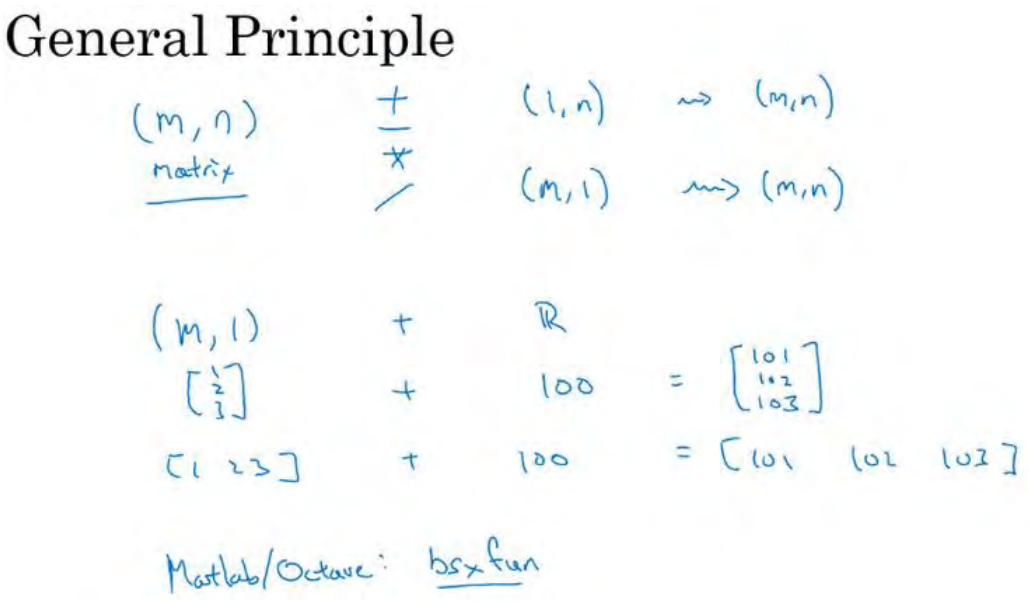
用一个 2×3的矩阵和一个 1×3 的矩阵相加，其泛化形式是 $m\times n$ 的矩阵和 $1\times n$ 的矩阵相加。在执行加法操作时，其实是将 $1\times n$ 的矩阵复制成为 $m\times n$ 的矩阵，然后两者做逐元素加法得到结果。针对这个具体例子，相当于在矩阵的第一列加 100，第二列加 200，第三列加 300。这就是在前一张幻灯片中计算卡路里百分比的广播机制，只不过这里是除法操作（广播机制与执行的运算种类无关）。

下面是最后一个例子



这里相当于是一个 $m\times n$ 的矩阵加上一个 $m\times1$ 的矩阵。在进行运算时，会先将 $m\times1$ 矩阵水平复制 n 次，变成一个 $m\times n$ 的矩阵，然后再执行逐元素加法。

广播机制的一般原则如下：



这里我先说一下我本人对 **numpy** 广播机制的理解，再解释上面这张 PPT。

首先是 **numpy** 广播机制

如果两个数组的后缘维度的轴长度相符或其中一方的轴长度为 1，则认为它们是广播兼

容的。广播会在缺失维度和轴长度为 1 的维度上进行。

后缘维度的轴长度：`A.shape[-1]` 即矩阵维度元组中的最后一个位置的值

对于视频中卡路里计算的例子，矩阵 $A_{3,4}$ 后缘维度的轴长度是 4，而矩阵 $cal_{1,4}$ 的后缘维度也是 4，则他们满足后缘维度轴长度相符，可以进行广播。广播会在轴长度为 1 的维度进行，轴长度为 1 的维度对应 `axis=0`，即垂直方向，矩阵 $\text{cal}_{1,4}$ 沿 `axis=0`(垂直方向)复制成为 $\text{cal\_temp}_{3,4}$ ，之后两者进行逐元素除法运算。

现在解释上图中的例子:

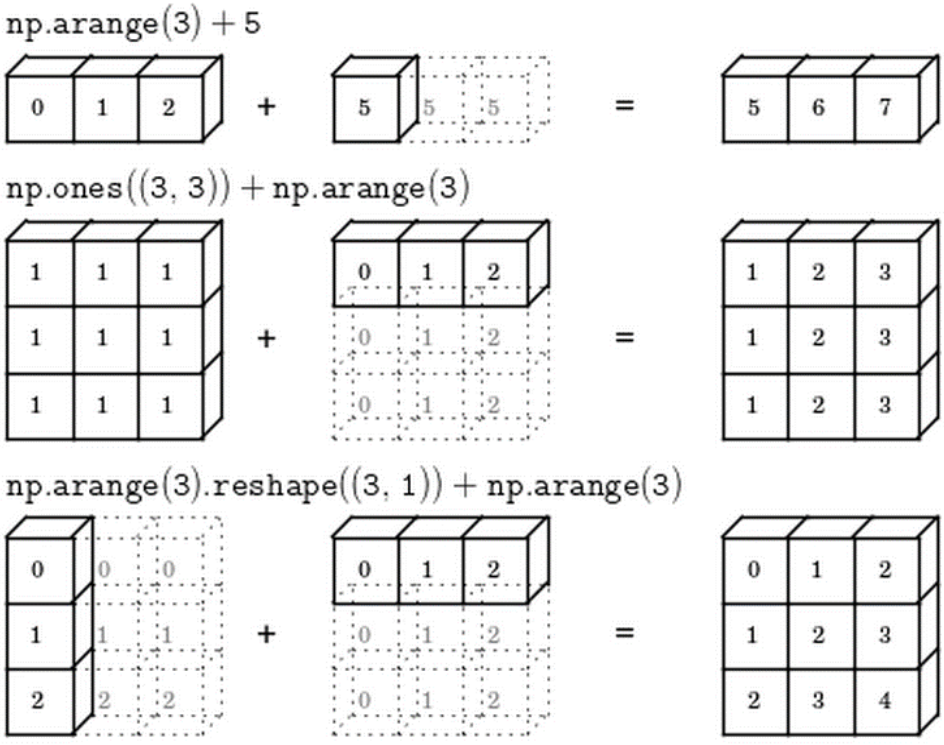
矩阵 $A_{m,n}$ 和矩阵 $B_{1,n}$ 进行四则运算，后缘维度轴长度相符，可以广播，广播沿着轴长度为 1 的轴进行，即 $B_{1,n}$ 广播成为 $B_{m,n}'$ ，之后做逐元素四则运算。

矩阵 $A_{m,n}$ 和矩阵 $B_{m,1}$ 进行四则运算，后缘维度轴长度不相符，但其中一方轴长度为 1，可以广播，广播沿着轴长度为 1 的轴进行，即 $B_{m,1}$ 广播成为 $B_{m,n}'$ ，之后做逐元素四则运算。

矩阵 $A_{m,1}$ 和常数R 进行四则运算，后缘维度轴长度不相符，但其中一方轴长度为 1，可以广播，广播沿着缺失维度和轴长度为 1 的轴进行，缺失维度就是 `axis=0`,轴长度为 1 的轴是 `axis=1`，即R广播成为 $B_{m,1}'$ ，之后做逐元素四则运算。

最后，对于 **Matlab/Octave** 有类似功能的函数 `bsxfun`。

总结一下 `broadcasting`，可以看看下面的图：



2.16 关于 python _ numpy 向量的说明（A note on python or numpy vectors）

本节主要讲 **Python** 中的 **numpy** 一维数组的特性，以及与行向量或列向量的区别。并介绍了老师在实际应用中的一些小技巧，去避免在 coding 中由于这些特性而导致的 bug。

Python 的特性允许你使用广播（**broadcasting**）功能，这是 **Python** 的 **numpy** 程序语言库中最灵活的地方。而我认为这是程序语言的优点，也是缺点。优点的原因在于它们创造出语言的表达性，**Python** 语言巨大的灵活性使得你仅仅通过一行代码就能做很多事情。但是这也是缺点，由于广播巨大的灵活性，有时候你对于广播的特点以及广播的工作原理这些细节不熟悉的话，你可能会产生很细微或者看起来很奇怪的 bug。例如，如果你将一个列向量添加到一个行向量中，你会以为它报出维度不匹配或类型错误之类的错误，但是实际上你会得到一个行向量和列向量的求和。

在 **Python** 的这些奇怪的影响之中，其实是有一个内在的逻辑关系的。但是如果对 Python 不熟悉的话，我就曾经见过的一些学生非常生硬、非常艰难地去寻找 bug。所以我在这里想做的就是分享给你们一些技巧，这些技巧对我非常有用，它们能消除或者简化我的代码中所有看起来很奇怪的 bug。同时我也希望通过这些技巧，你也能更容易地写没有 bug 的 **Python** 和 **numpy** 代码。

为了演示 Python-numpy 的一个容易被忽略的效果，特别是怎样在 Python-numpy 中构造向量，让我来做一个快速示范。首先设置$a = np.randomrandn(5)$，这样会生成存储在数组 a 中的 5 个高斯随机数变量。之后输出 a，从屏幕上可以得知，此时 a 的 **shape**（形状）是一个$(5,)$的结构。这在 **Python** 中被称作**一个一维数组**。它既不是一个行向量也不是一个列向量，这也导致它有一些不是很直观的效果。举个例子，如果我输出一个转置阵，最终结果它会和a看起来一样，所以a和a的转置阵最终结果看起来一样。而如果我输出a和a的转置阵的内积，你可能会想：a乘以a的转置返回给你的可能会是一个矩阵。但是如果我这样做，你只会得到一个数。

```
In [1]: import numpy as np
        a = np.random.randn(5)

In [2]: print(a)
        [ 0.50290632 -0.29691149  0.95429604 -0.82126861 -1.46269164]

In [3]: print(a.shape)
        (5,)

In [4]: print(a.T)
        [ 0.50290632 -0.29691149  0.95429604 -0.82126861 -1.46269164]

In [5]: print(np.dot(a,a.T))
        4.06570109321
```

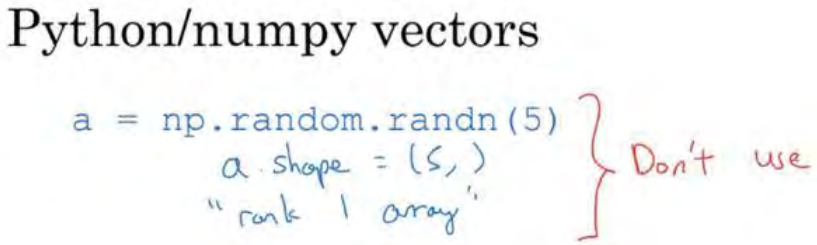
所以我建议当你编写神经网络时，不要在它的 **shape** 是$(5,)$还是$(n,)$或者一维数组时使用数据结构。相反，如果你设置 a 为$(5,1)$，那么这就将置于 5 行 1 列向量中。在先前的操作里 a 和 a 的转置看起来一样，而现在这样的 a 变成一个新的 a 的转置，并且它是一个行向量。请注意一个细微的差别，在这种数据结构中，当我们输出 a 的转置时有两对方括号，而之前只有一对方括号，所以这就是 1 行 5 列的矩阵和一维数组的差别。

```
In [4]: print(a.T)
        [ 0.50290632 -0.29691149  0.95429604 -0.82

In [5]: print(np.dot(a,a.T))
        4.06570109321

In [6]: a = np.random.randn(5,1)
        print(a)
        [[-0.0967311 ]
         [-2.38617377]
         [-0.3243588 ]
         [-0.96216349]
         [ 0.54410384]]

In [7]: print(a.T)
        [[-0.0967311  -2.38617377 -0.3243588  -0.9
```

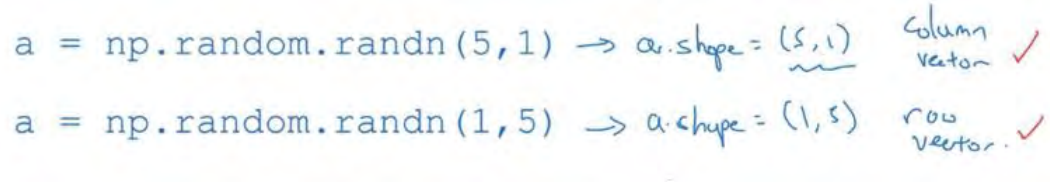
如果你输出 a 和 a 的转置的乘积，然后会返回给你一个向量的外积，是吧？所以这两个向量的外积返回给你的是一个矩阵。

```
[8]: print(np.dot(a,a.T))
[[ 0.00935691  0.23081721  0.03137558  0
 [ 0.23081721  5.69382526  0.77397645  2
 [ 0.03137558  0.77397645  0.10520863  0
 [ 0.09307113  2.29588928  0.31208619  0
 [-0.05263176 -1.2983263  -0.17648487 -0
```

就我们刚才看到的，再进一步说明。首先我们刚刚运行的命令是这个 $(a = np.random.randn(5))$，而且它生成了一个数据结构 $(a.shape)$，$a.shape$是(5,)，一个有趣的东西。这被称作 a 的一维数组，同时这也是一个非常有趣的数据结构。它不像行向量和列向量那样表现的很一致，这也让它的一些影响不那么明显。所以我建议，当你在编程练习或者在执行逻辑回归和神经网络时，你不需要使用这些一维数组。

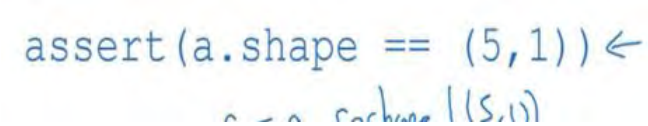


相反，如果你每次创建一个数组，你都得让它成为一个列向量，产生一个(5,1)向量或者你让它成为一个行向量，那么你的向量的行为可能会更容易被理解。所以在这种情况下，$a.shape$等同于(5,1)。这种表现很像 a，但是实际上却是一个列向量。同时这也是为什么当它是一个列向量的时候，你能认为这是矩阵(5,1)；同时这里 $a.shape$ 将要变成(1,5)，这就像行向量一样。所以当你需要一个向量时，我会说用这个或那个**(column vector or row vector)**，但绝不会是一维数组。



我写代码时还有一件经常做的事，那就是如果我不完全确定一个向量的维度**(dimension)**，我经常会扔进一个断言语句**(assertion statement)**。像这样，去确保在这种情况下是一个(5,1)向量，或者说是一个列向量。这些断言语句实际上是要去执行的，并且它们也会有助于为你的代码提供信息。所以不论你要做什么，不要犹豫直接插入断言语句。如果你不小心以一维数组来执行，你也能够重新改变数组维数 $a = reshape$，表明一个(5,1)数组或者一个(1,5)

数组，以致于它表现更像列向量或行向量。

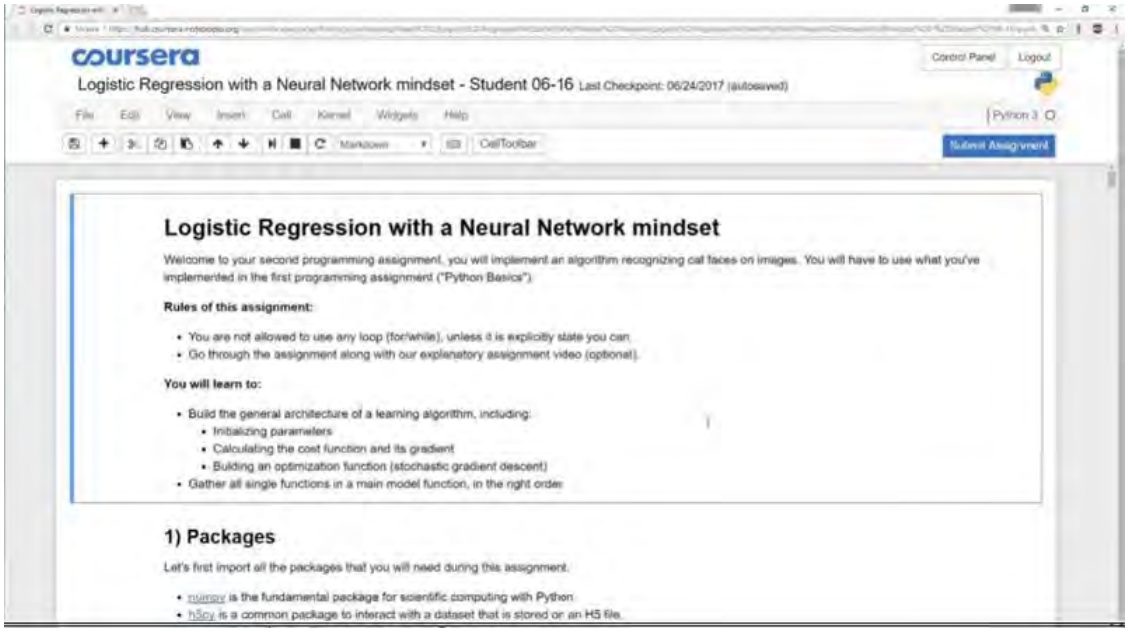


我有时候看见学生因为一维数组不直观的影响，难以定位 bug 而告终。通过在原先的代码里清除一维数组，我的代码变得更加简洁。而且实际上就我在代码中表现的事情而言，我从来不使用一维数组。因此，要去简化你的代码，而且不要使用一维数组。总是使用 $n \times 1$ 维矩阵（基本上是列向量），或者 $1 \times n$ 维矩阵（基本上是行向量），这样你可以减少很多 **assert** 语句来节省核矩阵和数组的维数的时间。另外，为了确保你的矩阵或向量所需要的维数时，不要羞于 **reshape** 操作。

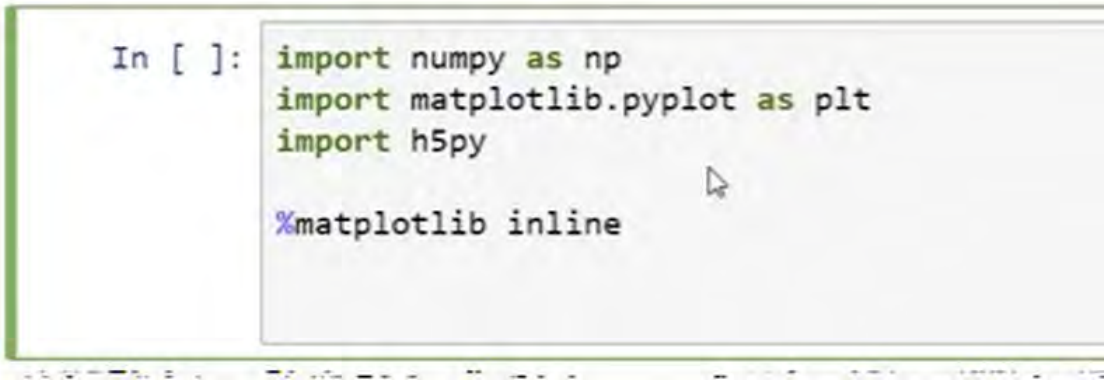
总之，我希望这些建议能帮助你解决一个 **Python** 中的 bug，从而使你更容易地完成练习。

2.17 Jupyter/iPython Notebooks 快速入门（Quick tour of Jupyter/iPython Notebooks）

学到现在，你即将要开始处理你的第一个编程作业。但在那之前，让我快速地给你介绍一下在 Coursera 上的 iPython Notebooks 工具。



这就是 **Jupyter iPython Notebooks** 的界面，你可以通过它连接到 **Coursera**。让我快速地讲解下它的一些特性。关于它的说明已经被写入这个 **Notebook** 中。



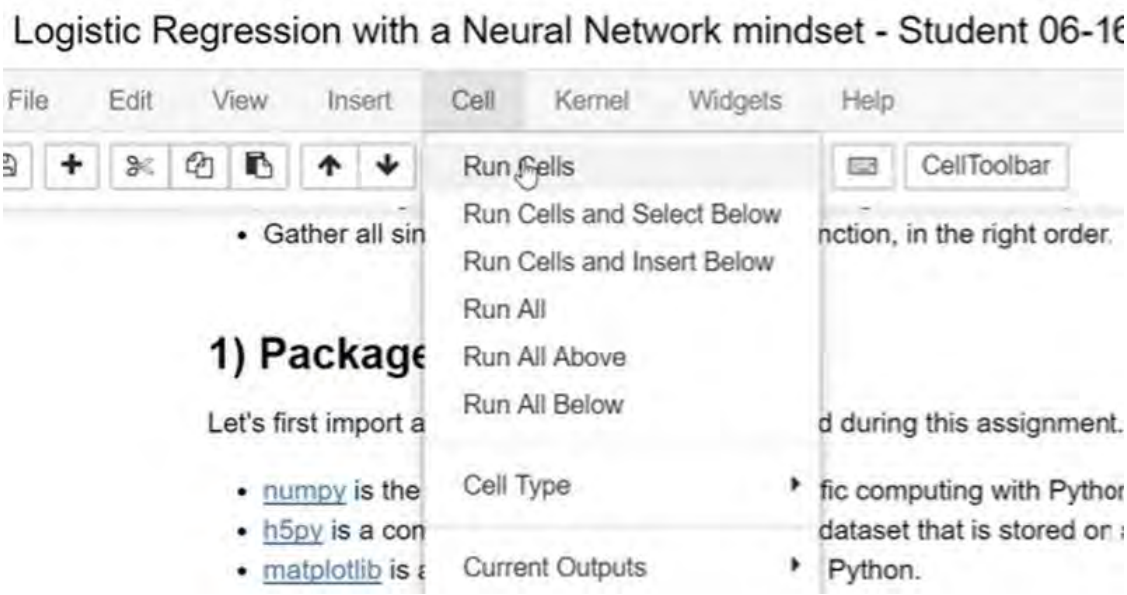
这里有一些空白区域的代码块，你可以在这里编写代码。有时，你也会看到一些函数块。而关于这些的说明都已经在 **iPython Notebook** 的文本中。在 **iPython Notebook** 中，在这些较长的灰色的区域就是代码块。

```
### START CODE HERE ### (≈ 3 lines of code)

### END CODE HERE ###
```

有时，你会看到代码块中有像这样的开始代码和结束代码。在进行编程练习时，请确保你的代码写在开始代码和结束代码之间。

```
In [5]: import numpy as np
        import matplotlib.pyplot as plt
        import h5py

        %matplotlib inline


        ### START CODE HERE ### (≈ 3 lines of code)
        print("Hello World")
        ### END CODE HERE ###

Hello World
```

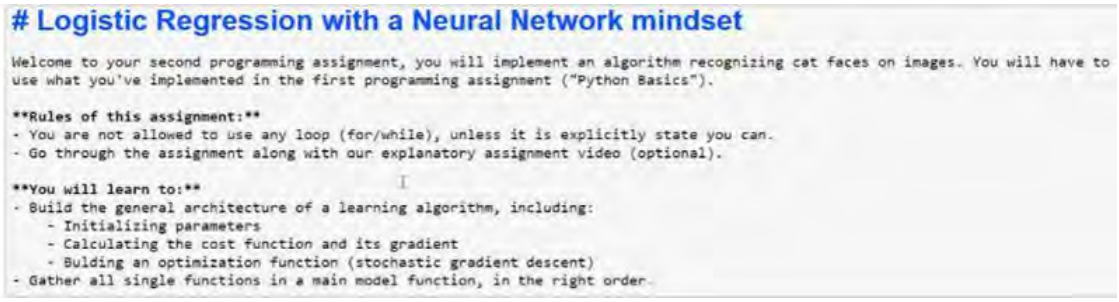
比如，编写打印输出 **Hello World** 的代码，然后执行这一代码块（你可以按 shift +enter 来执行这一代码块）。最终，它就会输出我们想要的 **Hello World**。



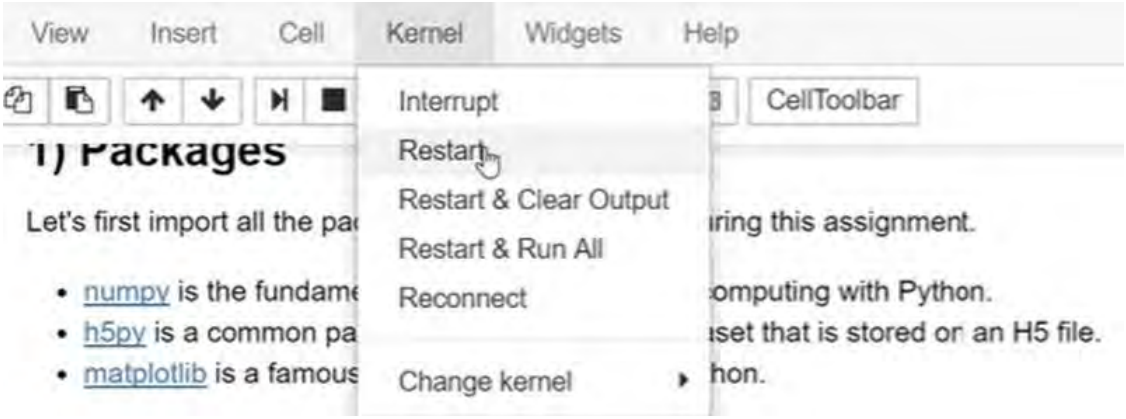
在运行一个单元格 **cell** 时，你也可以选择运行其中的一块代码区域。通过点击 **Cell** 菜单的 **Run Cells** 执行这部分代码。

也许，在你的计算机上，运行 **cell** 的键盘快捷方式可能并非是 **shift enter**。但是，Mac

应该和我的个人电脑一样，可以使用 **shift + enter** 来运行 **cell**。

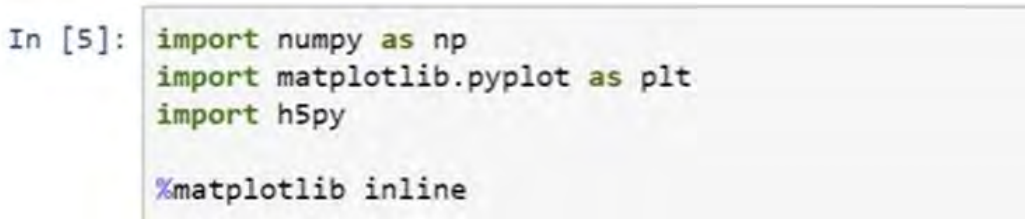


当你正在阅读指南时，如果不小心双击了它，点中的区域就会变成 **markdown** 语言形式。如果你不小心使其变成了这样的文本框，只要运行下单元格 **cell**，就可以回到原来的形式。所以，点击 **cell** 菜单的 **Run Cells** 或者使用 **shift + enter**，就可以使得它变回原样。



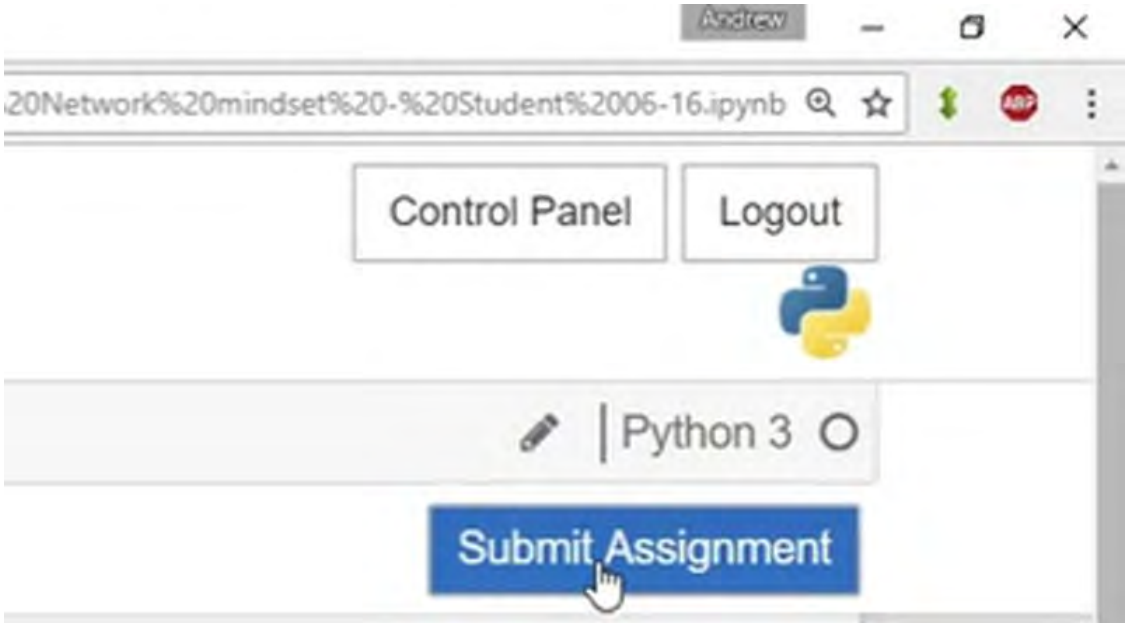
这里还有一些其他的小技巧。比如当你执行上面所使用的代码时，它实际上会使用一个内核在服务器上运行这段代码。如果你正在运行超负荷的进程，或者电脑运行了很长一段时间，或者在运行中出了错，又或者网络连接失败，这里依然有机会让 **Kernel** 重新工作。你只要点击 **Kernel**，选择 **Restart**，它会重新运行 **Kernel** 使程序继续工作。

所以，如果你只是运行相对较小的工作并且才刚刚启动你的 **ipad** 或笔记本电脑，这种情况应该是不会发生的。但是，如果你看见错误信息，比如 **Kernel** 已经中断或者其他信息，你可以试着重启 **Kernel**。



当我使用 **iPython Notebook** 时会有多个代码区域块。尽管我并没有在前面的代码块中添加自己的代码，但还是要确保先执行这块代码。因为在这个例子，它导入了 **numpy** 包并另命名为 **np** 等，并声明了一些你可能需要的变量。为了能顺利地执行下面的代码，就必须

确保先执行上面的代码，即使不要求你去写其他的代码。

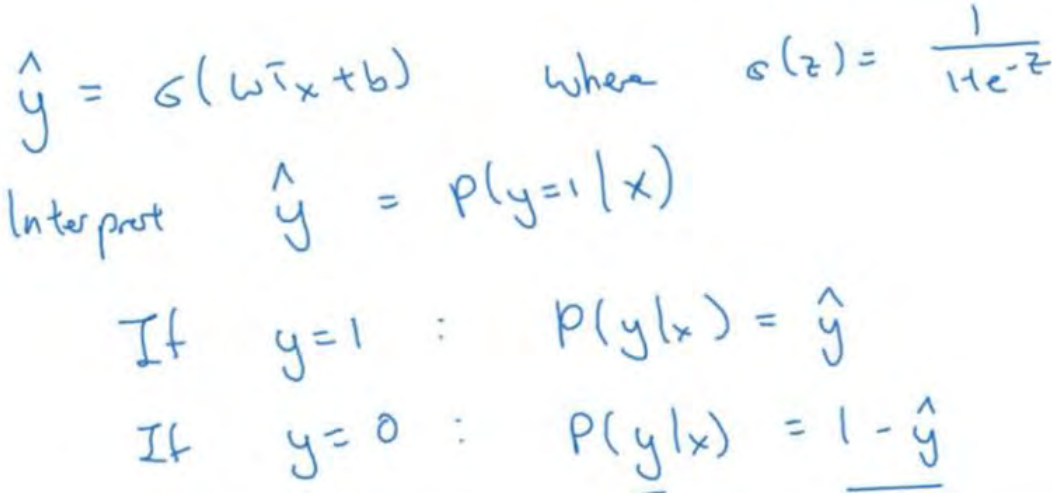


最后，当你完成作业后，可以通过点击右上方蓝色的 **Submit Assignment** 按钮提交你的作业。

我发现这种交互式的 **shell** 命令，在 **iPython Notebooks** 是非常有用的，能使你快速地实现代码并且查看输出结果，便于学习。所以我希望这些练习和 **Jupyter iPython Notebooks** 会帮助你更快地学习和实践，并且帮助你了解如何去实现这些学习算法。后面一个视频是一个选学视频，它主要是讲解逻辑回归中的代价函数。你可以选择是否观看。不管怎样，都祝愿你能通过这两次编程作业。我会在新一周的课程里等待着你。

2.18 （选修）logistic 损失函数的解释（Explanation of logistic regression cost function）

在前面的视频中，我们已经分析了逻辑回归的损失函数表达式，在这节选修视频中，我将给出一个简洁的证明来说明逻辑回归的损失函数为什么是这种形式。



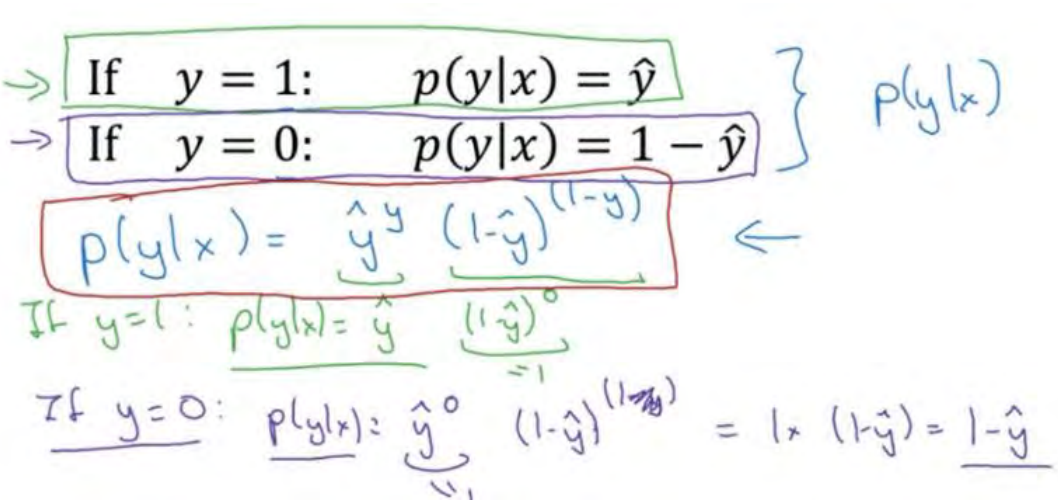
回想一下，在逻辑回归中，需要预测的结果$\hat{y}$,可以表示为$\hat{y} = \sigma(w^T x + b)$，$\sigma$是我们熟悉的$S$型函数 $\sigma(z) = \sigma(w^T x + b) = \frac{1}{1+e^{-z}}$ 。我们约定 $\hat{y} = p(y = 1|x)$ ，即算法的输出$\hat{y}$ 是给定训练样本 x 条件下 y 等于 1 的概率。换句话说，如果$y = 1$，在给定训练样本 x 条件下y等于$\hat{y}$；反过来说，如果$y = 0$，在给定训练样本x条件下 y 等于 1 减去$\hat{y}$ $(y = 1 - \hat{y})$，因此，如果 $\hat{y}$ 代表 $y = 1$ 的概率，那么$1 - \hat{y}$就是 $y = 0$的概率。接下来，我们就来分析这两个条件概率公式。

$$\text{If} \quad y = 1: \quad p(y|x) = \hat{y}$$
$$\text{If} \quad y = 0: \quad p(y|x) = 1 - \hat{y}$$

这两个条件概率公式定义形式为 $p(y|x)$并且代表了 $y = 0$ 或者 $y = 1$ 这两种情况，我们可以将这两个公式合并成一个公式。需要指出的是我们讨论的是二分类问题的损失函数，因此，y的取值只能是 0 或者 1。上述的两个条件概率公式可以合并成如下公式：

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{(1-y)}$$

接下来我会解释为什么可以合并成这种形式的表达式：$(1 - \hat{y})$的$(1 - y)$次方这行表达式包含了上面的两个条件概率公式，我来解释一下为什么。



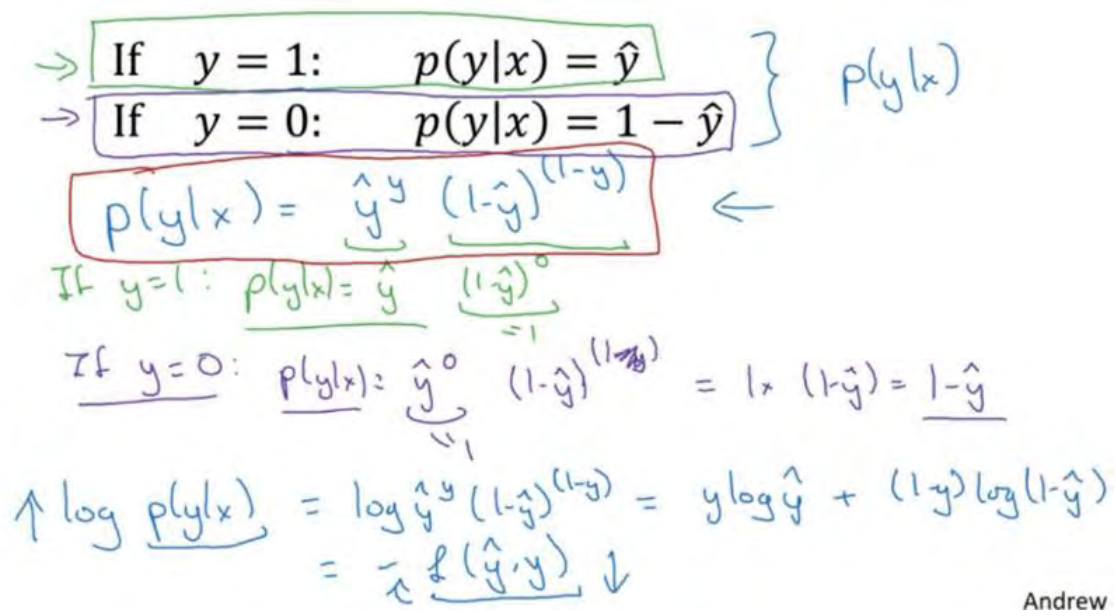
第一种情况，假设 $y=1$，由于$y=1$，那么$(\hat{y})^{y}=\hat{y}$，因为 $\hat{y}$的 1 次方等于$\hat{y}$，$1-(1-\hat{y})^{(1-y)}$的指数项$(1-y)$等于 0，由于任何数的 0 次方都是 1，$\hat{y}$乘以 1 等于$\hat{y}$。因此当$y=1$时 $p(y|x)=\hat{y}$（图中绿色部分）。

第二种情况，当 $y=0$ 时 $p(y|x)$ 等于多少呢? 假设$y=0$，$\hat{y}$的y次方就是 $\hat{y}$ 的 0 次方，任何数的 0 次方都等于 1，因此 $p(y|x)=1\times(1-\hat{y})^{1-y}$ ，前面假设 $y=0$ 因此$(1-y)$就等于 1，因此 $p(y|x)=1\times(1-\hat{y})$。因此在这里当$y=0$时，$p(y|x)=1-\hat{y}$。这就是这个公式(第二个公式，图中紫色字体部分)的结果。

因此，刚才的推导表明 $p(y|x)=\hat{y}^{(y)}(1-\hat{y})^{(1-y)}$，就是 $p(y|x)$ 的完整定义。由于 log 函数是严格单调递增的函数，最大化 $log(p(y|x))$ 等价于最大化 $p(y|x)$ 并且地计算 $p(y|x)$ 的 log 对数，就是计算 $log(\hat{y}^{(y)}(1-\hat{y})^{(1-y)})$ (其实就是将 $p(y|x)$ 代入)，通过对数函数化简为：

$$ylog\,\hat{y}+(1-y)log(1-\hat{y})$$

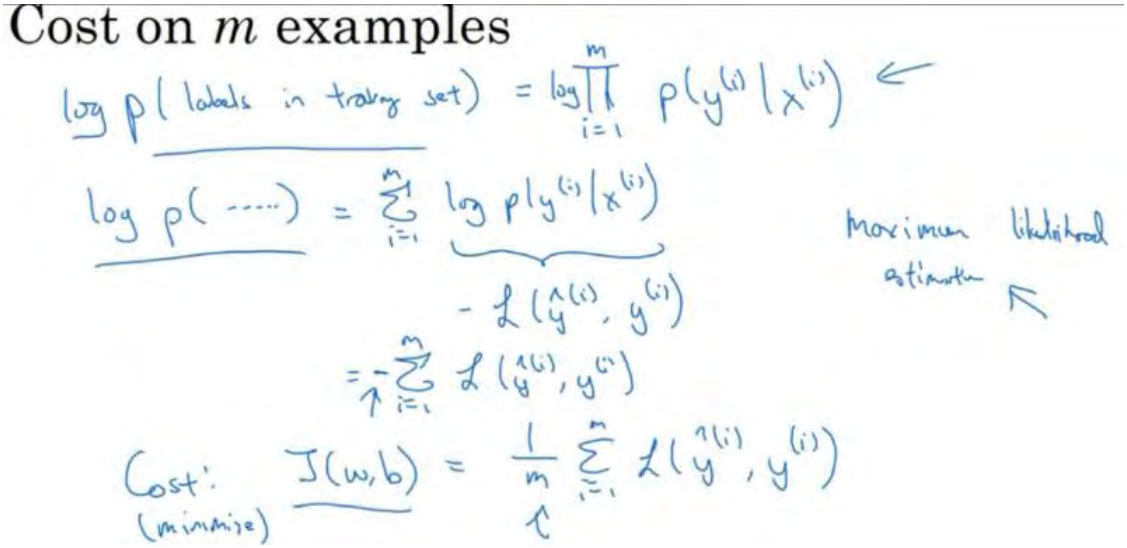
而这就是我们前面提到的损失函数的负数 $(-L(\hat{y},y))$ ，前面有一个负号的原因是当你训练学习算法时需要算法输出值的概率是最大的（以最大的概率预测这个值），然而在逻辑回归中我们需要最小化损失函数，因此最小化损失函数与最大化条件概率的对数 $log(p(y|x))$ 关联起来了，因此这就是单个训练样本的损失函数表达式。



在 m个训练样本的整个训练集中又该如何表示呢，让我们一起来探讨一下。

让我们一起来探讨一下，整个训练集中标签的概率，更正式地来写一下。假设所有的训练样本服从同一分布且相互独立，也即独立同分布的，所有这些样本的联合概率就是每个样本概率的乘积:

$P(\text{labels in training set}) = \prod_{i=1}^{m} P(y^{(i)}|x^{(i)})$。

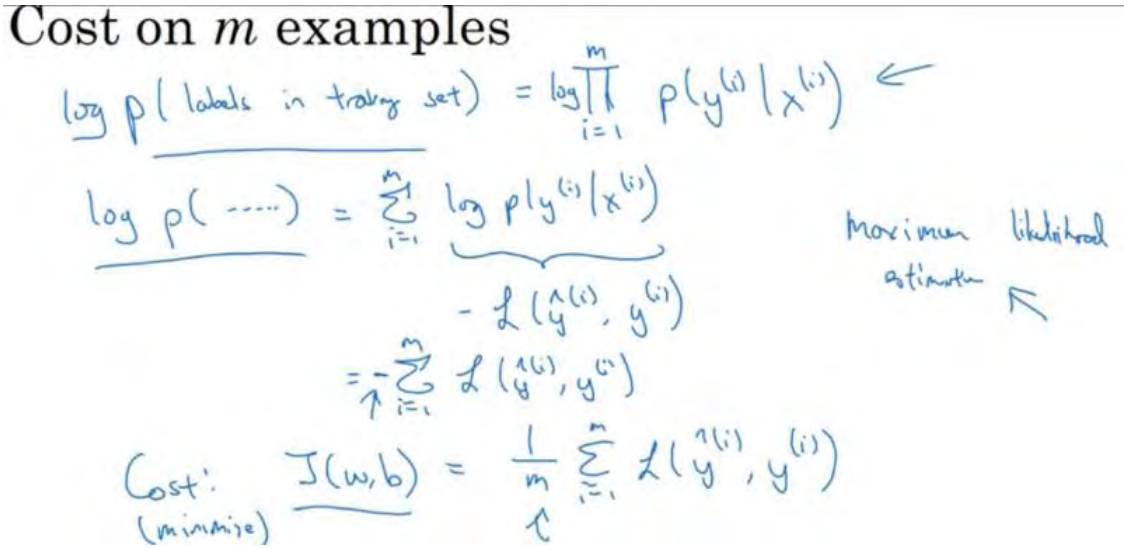


如果你想做最大似然估计，需要寻找一组参数，使得给定样本的观测值概率最大，但令这个概率最大化等价于令其对数最大化，在等式两边取对数：

$$logp(\text{labels in training set}) = log \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) = \sum_{i=1}^{m} logP(y^{(i)}|x^{(i)}) = \sum_{i=1}^{m} -L(\hat{y}^{(i)}, y^{(i)})$$

在统计学里面，有一个方法叫做最大似然估计，即求出一组参数，使这个式子取最大值，也就是说，使得这个式子取最大值，$\sum_{i=1}^{m} -L(\hat{y}^{(i)}, y^{(i)})$，可以将负号移到求和符号的外面，

$-\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$，这样我们就推导出了前面给出的 logistic 回归的成本函数$J(w,b) = \sum_{i=1}^{m} L(\hat{y}^{(i)}, \hat{y^{(i)}})$。



由于训练模型时，目标是让成本函数最小化，所以我们不是直接用最大似然概率，要去掉这里的负号，最后为了方便，可以对成本函数进行适当的缩放，我们就在前面加一个额外的常数因子$\frac{1}{m}$，即:$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$。

总结一下，为了最小化成本函数$J(w,b)$，我们从 **logistic** 回归模型的最大似然估计的角度出发，假设训练集中的样本都是独立同分布的条件下。尽管这节课是选修性质的，但还是感谢观看本节视频。我希望通过本节课您能更好地明白逻辑回归的损失函数，为什么是那种形式，明白了损失函数的原理，希望您能继续完成课后的练习，前面课程的练习以及本周的测验，在课后的小测验和编程练习中,祝您好运。

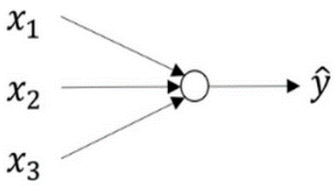
第三周：浅层神经网络(Shallow neural networks)

3.1 神经网络概述（Neural Network Overview）

本周你将学习如何实现一个神经网络。在我们深入学习具体技术之前，我希望快速的带你预览一下本周你将会学到的东西。如果这个视频中的某些细节你没有看懂你也不用担心，我们将在后面的几个视频中深入讨论技术细节。

现在我们开始快速浏览一下如何实现神经网络。上周我们讨论了逻辑回归，我们了解了这个模型(见图 3.1.1)如何与下面公式 3.1 建立联系。

图 3.1.1 :



公式 3.1：

$$\left.\begin{matrix} x \\ w \\ b \end{matrix}\right\} \Longrightarrow z = w^T x + b$$

如上所示，首先你需要输入特征x，参数w和b，通过这些你就可以计算出z，公式 3.2：

$$\left.\begin{matrix} x \\ w \\ b \end{matrix}\right\} \Longrightarrow z = w^T x + b \Longrightarrow \alpha = \sigma(z) \Longrightarrow L(a, y)$$

接下来使用z就可以计算出a。我们将的符号换为表示输出$\hat{y} \Longrightarrow a = \sigma(z)$,然后可以计算出 **loss function** $L(a, y)$

神经网络看起来是如下这个样子（图 3.1.2）。正如我之前已经提到过，你可以把许多 **sigmoid** 单元堆叠起来形成一个神经网络。对于图 3.1.1 中的节点，它包含了之前讲的计算的两个步骤：首先通过公式 3.1 计算出值z，然后通过$\sigma(z)$计算值a。

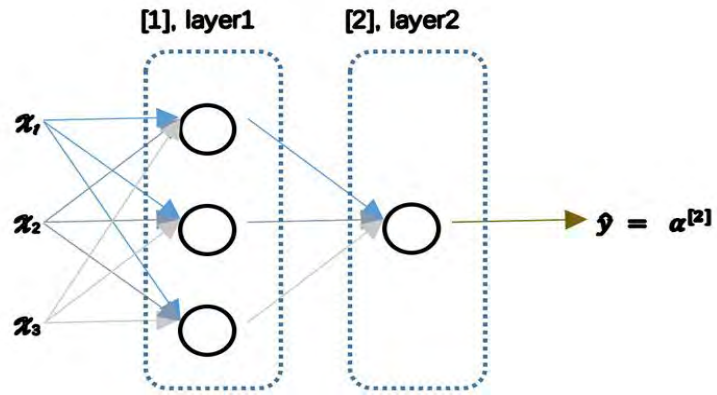


图 3.1.2

在这个神经网络（图 3.1.2）对应的 3 个节点，首先计算第一层网络中的各个节点相关的数$z^{[1]}$，接着计算$a^{[1]}$，在计算下一层网络同理； 我们会使用符号 $^{[m]}$表示第m层网络中节点相关的数，这些节点的集合被称为第m层网络。这样可以保证 $^{[m]}$不会和我们之前用来表示单个的训练样本的 $^{(i)}$(即我们使用表示第 i 个训练样本)混淆； 整个计算过程，公式如下: 公式 3.3：

$$\left.\begin{matrix} x \\ W^{[1]} \\ b^{[1]} \end{matrix}\right\} \Longrightarrow z^{[1]} = W^{[1]}x + b^{[1]} \Longrightarrow a^{[1]} = \sigma(z^{[1]})$$

公式 3.4：

$$\left.\begin{matrix} x \\ dW^{[1]} \\ db^{[1]} \end{matrix}\right\} \Longleftarrow dz^{[1]} = d(W^{[1]}x + b^{[1]}) \Longleftarrow d\alpha^{[1]} = d\sigma(z^{[1]})$$

类似逻辑回归，在计算后需要使用计算，接下来你需要使用另外一个线性方程对应的参数计算$z^{[2]}$， 计算$a^{[2]}$，此时$a^{[2]}$就是整个神经网络最终的输出，用 $\hat{y}$表示网络的输出。

公式 3.5：

$$\left.\begin{matrix} x \\ dW^{[1]} \\ db^{[1]} \end{matrix}\right\} \Longleftarrow dz^{[1]} = d(W^{[1]}x + b^{[1]}) \Longleftarrow d\alpha^{[1]} = d\sigma(z^{[1]})$$

公式 3.6：

$$\left.\begin{matrix} da^{[1]} = d\sigma(z^{[1]}) \\ dW^{[2]} \\ db^{[2]} \end{matrix}\right\} \Longleftarrow dz^{[2]} = d(W^{[2]}\alpha^{[1]} + b^{[2]}) \Longleftarrow da^{[2]} = d\sigma(z^{[2]})$$

$$\Longleftarrow dL\left(a^{[2]}, y\right)$$

我知道这其中有很多细节，其中有一点非常难以理解，即在逻辑回归中，通过直接计算z得到结果a。而这个神经网络中，我们反复的计算z和a，计算a和z，最后得到了最终的输出 **loss function**。

你应该记得逻辑回归中，有一些从后向前的计算用来计算导数da、dz。同样，在神经网络中我们也有从后向前的计算，看起来就像这样，最后会计算$da^{[2]}$ 、$dz^{[2]}$，计算出来之后，然后计算计算$dW^{[2]}$、$db^{[2]}$ 等，按公式 3.5、3.6 箭头表示的那样，从右到左反向计算。

现在你大概了解了一下什么是神经网络，基于逻辑回归重复使用了两次该模型得到上述例子的神经网络。我清楚这里面多了很多新符号和细节，如果没有理解也不用担心，在接下来的视频中我们会仔细讨论具体细节。

那么，下一个视频讲述神经网络的表示。

3.2 神经网络的表示（Neural Network Representation）

先回顾一下我在上一个视频画几张神经网络的图片，在这次课中我们将讨论这些图片的具体含义，也就是我们画的这些神经网络到底代表什么。

我们首先关注一个例子，本例中的神经网络只包含一个隐藏层（图 3.2.1）。这是一张神经网络的图片，让我们给此图的不同部分取一些名字。

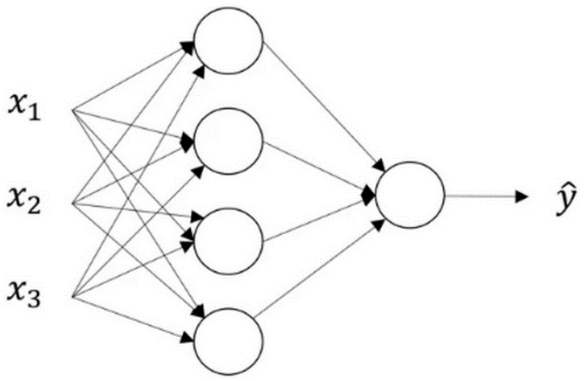


图 3.2.1

我们有输入特征x_1、x_2、x_3，它们被竖直地堆叠起来，这叫做神经网络的**输入层**。它包含了神经网络的输入；然后这里有另外一层我们称之为**隐藏层**（图 3.2.1 的四个结点）。待会儿我会回过头来讲解术语"隐藏"的意义；在本例中最后一层只由一个结点构成，而这个只有一个结点的层被称为**输出层**，它负责产生预测值。解释隐藏层的含义：在一个神经网络中，当你使用监督学习训练它的时候，训练集包含了输入x也包含了目标输出y，所以术语隐藏层的含义是在训练集中，这些中间结点的准确值我们是不知道到的，也就是说你看不见它们在训练集中应具有的值。你能看见输入的值，你也能看见输出的值，但是隐藏层中的东西，在训练集中你是无法看到的。所以这也解释了词语隐藏层，只是表示你无法在训练集中看到他们。

现在我们再引入几个符号，就像我们之前用向量x表示输入特征。这里有个可代替的记号$a^{[0]}$可以用来表示输入特征。a表示激活的意思，它意味着网络中不同层的值会传递到它们后面的层中，输入层将x传递给隐藏层，所以我们将输入层的激活值称为$a^{[0]}$；下一层即隐藏层也同样会产生一些激活值，那么我将其记作$a^{[1]}$，所以具体地，这里的第一个单元或结点我们将其表示为$a_1^{[1]}$，第二个结点的值我们记为$a_2^{[1]}$以此类推。所以这里的是一个四维的向量如果写成 Python 代码，那么它是一个规模为 4x1 的矩阵或一个大小为 4 的列向量，如下公式，它是四维的，因为在本例中，我们有四个结点或者单元，或者称为四个隐藏层单元； 公

式 3.7

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

最后输出层将产生某个数值a，它只是一个单独的实数，所以的$\hat{y}$值将取为$a^{[2]}$。这与逻辑回归很相似，在逻辑回归中，我们有$\hat{y}$直接等于a，在逻辑回归中我们只有一个输出层，所以我们没有用带方括号的上标。但是在神经网络中，我们将使用这种带上标的形式来明确地指出这些值来自于哪一层，有趣的是在约定俗成的符号传统中，在这里你所看到的这个例子，只能叫做一个两层的神经网络（图 3.2.2）。原因是当我们计算网络的层数时，输入层是不算入总层数内，所以隐藏层是第一层，输出层是第二层。第二个惯例是我们将输入层称为第零层，所以在技术上，这仍然是一个三层的神经网络，因为这里有输入层、隐藏层，还有输出层。但是在传统的符号使用中，如果你阅读研究论文或者在这门课中，你会看到人们将这个神经网络称为一个两层的神经网络，因为我们不将输入层看作一个标准的层。

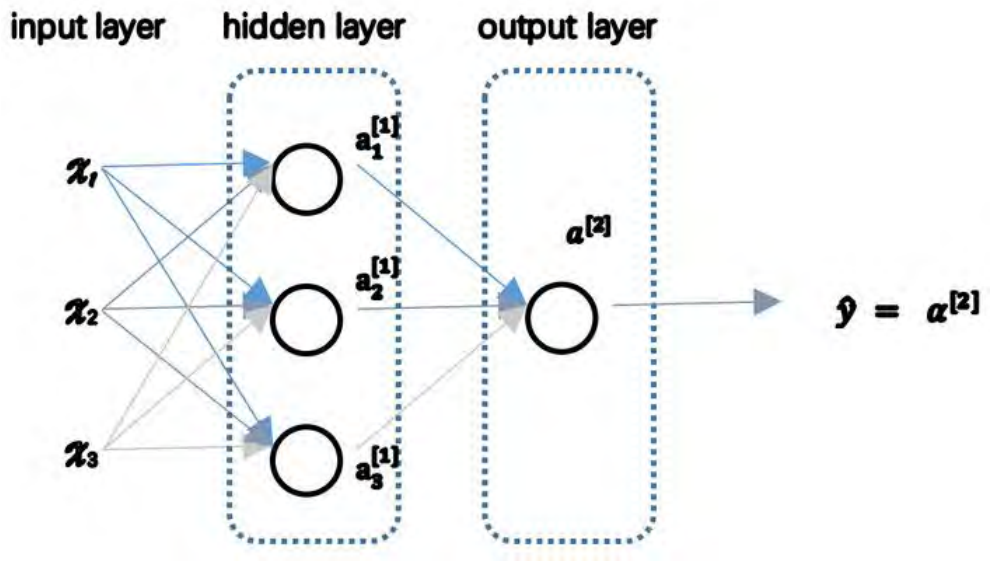


图 3.2.2

最后，我们要看到的隐藏层以及最后的输出层是带有参数的，这里的隐藏层将拥有两个参数W和b，我将给它们加上上标 $^{[1]}$($w^{[1]}$,$b^{[1]}$)，表示这些参数是和第一层这个隐藏层有关系的。之后在这个例子中我们会看到w是一个 4x3 的矩阵，而b是一个 4x1 的向量，第一个数字 4 源自于我们有四个结点或隐藏层单元，然后数字 3 源自于这里有三个输入特征，我们之后会更加详细地讨论这些矩阵的维数，到那时你可能就更加清楚了。相似的输出层也有一些与之关联的参数$W^{[2]}$以及$b^{[2]}$。从维数上来看，它们的规模分别是 1x4 以及 1x1。1x4 是因

为隐藏层有四个隐藏层单元而输出层只有一个单元，之后我们会对这些矩阵和向量的维度做出更加深入的解释，所以现在你已经知道一个两层的神经网络什么样的了，即它是一个只有一个隐藏层的神经网络。

在下一个视频中。我们将更深入地了解这个神经网络是如何进行计算的，也就是这个神经网络是怎么输入x，然后又是怎么得到$\hat{y}$。

3.3 计算一个神经网络的输出（Computing a Neural Network's output）

在上一节的视频中，我们介绍只有一个隐藏层的神经网络的结构与符号表示。在这节的视频中让我们了解神经网络的输出究竟是如何计算出来的。

首先，回顾下只有一个隐藏层的简单两层**神经网络结构**：

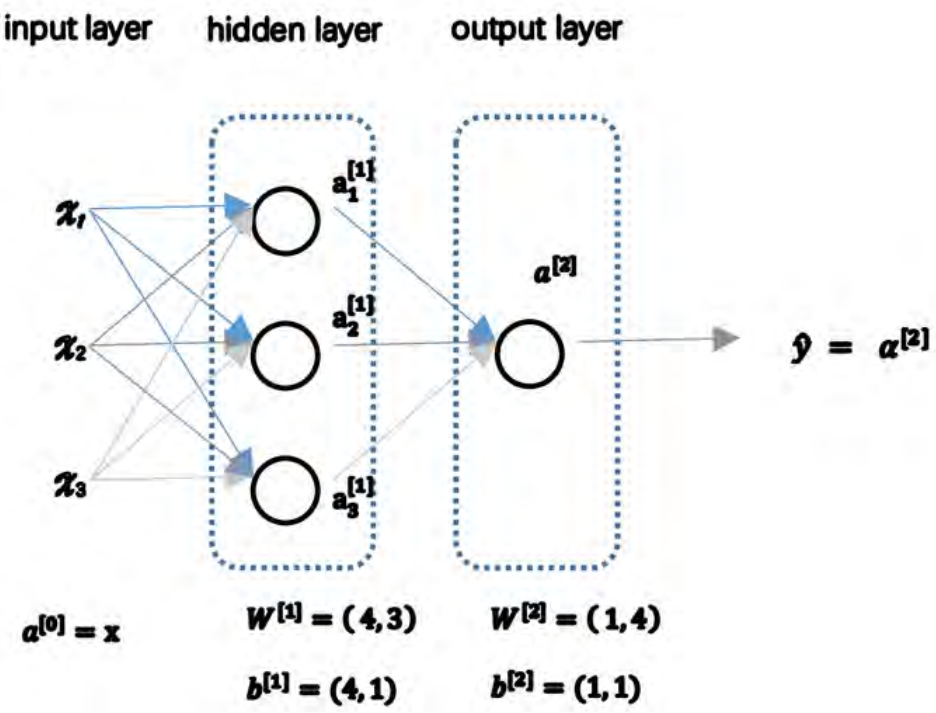
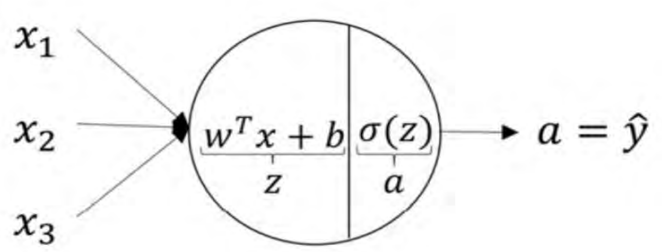


图 3.3.1

其中，x表示输入特征，a表示每个神经元的输出，W表示特征的权重，上标表示神经网络的层数（隐藏层为 1），下标表示该层的第几个神经元。这是神经网络的**符号惯例**，下同。

神经网络的计算

关于神经网络是怎么计算的，从我们之前提及的逻辑回归开始，如下图所示。用圆圈表示神经网络的计算单元，逻辑回归的计算有两个步骤，首先你按步骤计算出z，然后在第二步中你以 **sigmoid** 函数为激活函数计算z（得出a），一个神经网络只是这样子做了好多次重复计算。



$$z = w^T x + b$$

$$a = \sigma(z)$$

图 3.3.2

回到两层的神经网络，我们从隐藏层的第一个神经元开始计算，如上图第一个最上面的箭头所指。从上图可以看出，输入与逻辑回归相似，这个神经元的计算与逻辑回归一样分为两步，小圆圈代表了计算的两个步骤。

第一步，计算$z_1^{[1]}, z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$。

第二步，通过激活函数计算$a_1^{[1]}, a_1^{[1]} = \sigma(z_1^{[1]})$。

隐藏层的第二个以及后面两个神经元的计算过程一样，只是注意符号表示不同，最终分别得到$a_2^{[1]}$、$a_3^{[1]}$、$a_4^{[1]}$，详细结果见下:

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, a_4^{[1]} = \sigma(z_4^{[1]})$$

向量化计算 如果你执行神经网络的程序，用 for 循环来做这些看起来真的很低效。所以接下来我们要做的就是把这四个等式向量化。向量化的过程是将神经网络中的一层神经元参数纵向堆积起来，例如隐藏层中的w纵向堆积起来变成一个$(4,3)$的矩阵，用符号$W^{[1]}$表示。另一个看待这个的方法是我们有四个逻辑回归单元，且每一个逻辑回归单元都有相对应的参数——向量w，把这四个向量堆积在一起，你会得出这 4×3 的矩阵。因此， 公式 3.8： $z^{[n]} = w^{[n]} x + b^{[n]}$

公式 3.9：

$a^{[n]} = \sigma(z^{[n]})$

详细过程见下：公式 3.10：

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

公式 3.11：

$$\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = \overbrace{\begin{bmatrix} \dots W_1^{[1]T} \dots \\ \dots W_2^{[1]T} \dots \\ \dots W_3^{[1]T} \dots \\ \dots W_4^{[1]T} \dots \end{bmatrix}}^{W^{[1]}} * \overbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}^{input} + \overbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}^{b^{[1]}}$$

对于神经网络的第一层，给予一个输入x，得到$a^{[1]}$，x可以表示为$a^{[0]}$。通过相似的衍生你会发现，后一层的表示同样可以写成类似的形式，得到$a^{[2]}$，$\hat{y} = a^{[2]}$，具体过程见公式 3.8、3.9。

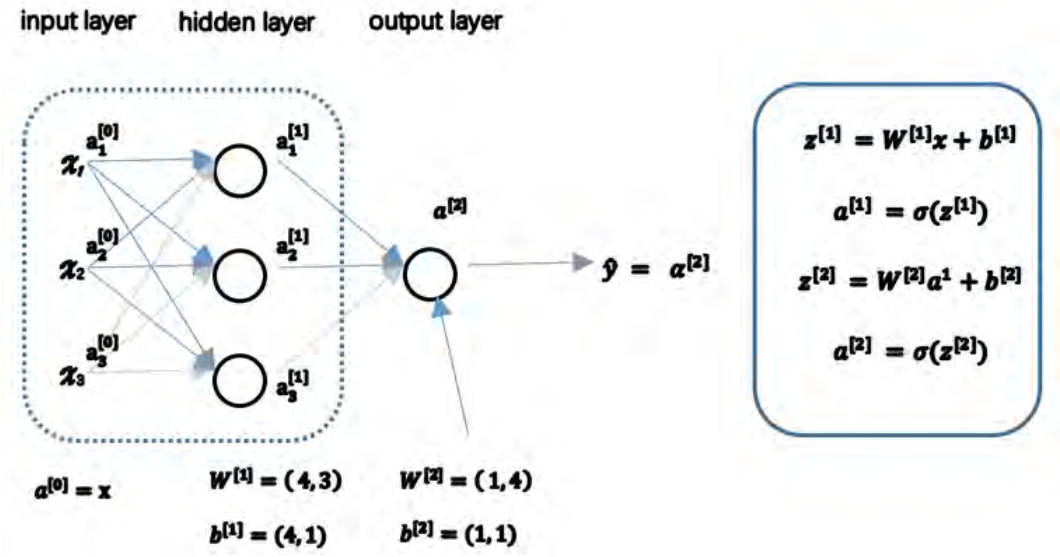


图 3.3.3

如上图左半部分所示为神经网络，把网络左边部分盖住先忽略，那么最后的输出单元就相当于一个逻辑回归的计算单元。当你有一个包含一层隐藏层的神经网络，你需要去实现以计算得到输出的是右边的四个等式，并且可以看成是一个向量化的计算过程，计算出隐藏层的四个逻辑回归单元和整个隐藏层的输出结果，如果编程实现需要的也只是这四行代码。

总结 通过本视频，你能够根据给出的一个单独的输入特征向量，运用四行代码计算出一个简单神经网络的输出。接下来你将了解的是如何一次能够计算出不止一个样本的神经网络输出，而是能一次性计算整个训练集的输出。

3.4 多样本向量化（Vectorizing across multiple examples）

在上一个视频，了解到如何针对于单一的训练样本，在神经网络上计算出预测值。

在这个视频，将会了解到如何向量化多个训练样本，并计算出结果。该过程与你在逻辑回归中所做类似。

逻辑回归是将各个训练样本组合成矩阵，对矩阵的各列进行计算。神经网络是通过对逻辑回归中的等式简单的变形，让神经网络计算出输出值。这种计算是所有的训练样本同时进行的，以下是实现它具体的步骤：

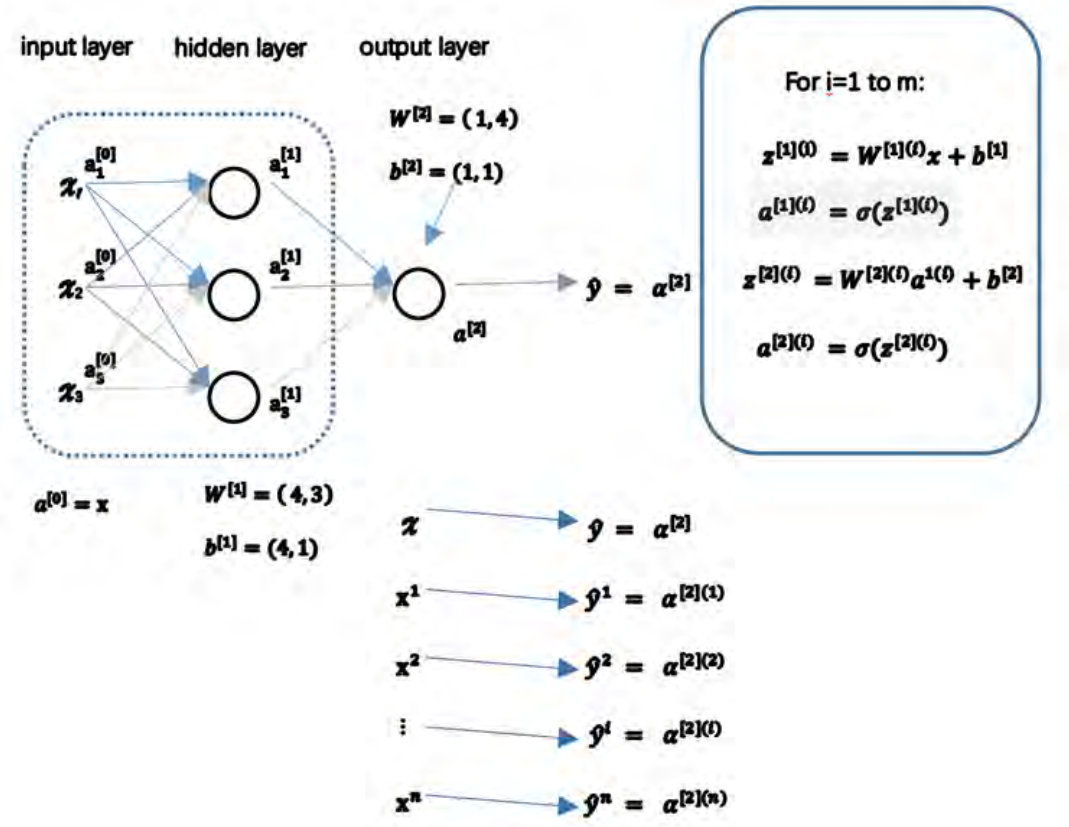


图 3.4.1

上一节视频中得到的四个等式。它们给出如何计算出$z^{[1]}$，$a^{[1]}$，$z^{[2]}$，$a^{[2]}$。

对于一个给定的输入特征向量X，这四个等式可以计算出$a^{[2]}$等于$\hat{y}$。这是针对于单一的训练样本。如果有m个训练样本,那么就需要重复这个过程。

用第一个训练样本$x^{[1]}$来计算出预测值$\hat{y}^{[1]}$，就是第一个训练样本上得出的结果。

然后，用$x^{[2]}$来计算出预测值$\hat{y}^{[2]}$，循环往复，直至用$x^{[m]}$计算出$\hat{y}^{[m]}$。

用激活函数表示法，如上图左下所示，它写成$a^{[2](1)}$、$a^{[2](2)}$和$a^{[2](m)}$。

【注】：$a^{[2](i)}$，(i)是指第i个训练样本而$[2]$是指第二层。

如果有一个非向量化形式的实现，而且要计算出它的预测值，对于所有训练样本，需要让i从 1 到m实现这四个等式：

$$z^{[1](i)} = W^{[1](i)}x^{(i)} + b^{[1](i)}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2](i)}a^{[1](i)} + b^{[2](i)}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

对于上面的这个方程中的 $^{(i)}$，是所有依赖于训练样本的变量，即将(i)添加到x，z和a。如果想计算m个训练样本上的所有输出，就应该向量化整个计算，以简化这列。

本课程需要使用很多线性代数的内容，重要的是能够正确地实现这一点，尤其是在深度学习的错误中。实际上本课程认真地选择了运算符号，这些符号只是针对于这个课程的，并且能使这些向量化容易一些。

所以，希望通过这个细节可以更快地正确实现这些算法。接下来讲讲如何向量化这些：公式 3.12：

$$x = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

公式 3.13：

$$Z^{[1]} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

公式 3.14：

$$A^{[1]} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ \alpha^{[1](1)} & \alpha^{[1](2)} & \cdots & \alpha^{[1](m)} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

公式 3.15：

$$\left.\begin{aligned} z^{[1](i)} &= W^{[1](i)}x^{(i)} + b^{[1]} \\ \alpha^{[1](i)} &= \sigma(z^{[1](i)}) \\ z^{[2](i)} &= W^{[2](i)}\alpha^{[1](i)} + b^{[2]} \\ \alpha^{[2](i)} &= \sigma(z^{[2](i)}) \end{aligned}\right\} \Longrightarrow \begin{cases} A^{[1]} = \sigma(z^{[1]}) \\ z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} = \sigma(z^{[2]}) \end{cases}$$

前一张幻灯片中的 **for** 循环是来遍历所有个训练样本。 定义矩阵X等于训练样本，将它们组合成矩阵的各列，形成一个n维或n乘以m维矩阵。接下来计算见公式 3.15：

以此类推，从小写的向量x到这个大写的矩阵X，只是通过组合x向量在矩阵的各列中。

同理，$z^{[1](1)}$，$z^{[1](2)}$等等都是$z^{[1](m)}$的列向量，将所有m都组合在各列中，就的到矩阵

$Z^{[1]}$。

同理，$a^{[1](1)}$，$a^{[1](2)}$，……，$a^{[1](m)}$将其组合在矩阵各列中，如同从向量x到矩阵X，以及从向量z到矩阵Z一样，就能得到矩阵$A^{[1]}$。

同样的，对于$Z^{[2]}$和$A^{[2]}$，也是这样得到。

这种符号其中一个作用就是，可以通过训练样本来进行索引。这就是水平索引对应于不同的训练样本的原因，这些训练样本是从左到右扫描训练集而得到的。

在垂直方向，这个垂直索引对应于神经网络中的不同节点。例如，这个节点，该值位于矩阵的最左上角对应于激活单元，它是位于第一个训练样本上的第一个隐藏单元。它的下一个值对应于第二个隐藏单元的激活值。它是位于第一个训练样本上的，以及第一个训练示例中第三个隐藏单元，等等。

当垂直扫描，是索引到隐藏单位的数字。当水平扫描，将从第一个训练示例中从第一个隐藏的单元到第二个训练样本，第三个训练样本……直到节点对应于第一个隐藏单元的激活值，且这个隐藏单元是位于这 m 个训练样本中的最终训练样本。

从水平上看，矩阵 A 代表了各个训练样本。从竖直上看，矩阵 A 的不同的索引对应于不同的隐藏单元。

对于矩阵Z，X情况也类似，水平方向上，对应于不同的训练样本；竖直方向上，对应不同的输入特征，而这就是神经网络输入层中各个节点。

神经网络上通过在多样本情况下的向量化来使用这些等式。

在下一个视频中，将证明为什么这是一种正确向量化的实现。这种证明将会与逻辑回归中的证明类似。

3.5 向量化实现的解释（Justification for vectorized implementation）

在上一个视频中，我们学习到如何将多个训练样本横向堆叠成一个矩阵X，然后就可以推导出神经网络中前向传播（forward propagation）部分的向量化实现。

在这个视频中，我们将会继续了解到，为什么上一节中写下的公式就是将多个样本向量化的正确实现。

我们先手动对几个样本计算一下前向传播，看看有什么规律： 公式 3.16： $z^{[1](1)} = W^{[1]}x^{(1)} + b^{[1]}$

$$z^{[1](2)} = W^{[1]}x^{(2)} + b^{[1]}$$

$$z^{[1](3)} = W^{[1]}x^{(3)} + b^{[1]}$$

这里，为了描述的简便，我们先忽略掉 $b^{[1]}$后面你将会看到利用 **Python** 的广播机制，可以很容易的将$b^{[1]}$ 加进来。

现在 $W^{[1]}$ 是一个矩阵，$x^{(1)}, x^{(2)}, x^{(3)}$都是列向量，矩阵乘以列向量得到列向量，下面将它们用图形直观的表示出来: 公式 3.17：

$$W^{[1]}x = \begin{bmatrix} \cdots \\ \cdots \\ \cdots \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ x^{(1)} & x^{(2)} & x^{(3)} & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ w^{(1)}x^{(1)} & w^{(1)}x^{(2)} & w^{(1)}x^{(3)} & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} =$$

$$\begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} = Z^{[1]}$$

视频中，吴恩达老师很细心的用不同的颜色表示不同的样本向量，及其对应的输出。所以从图中可以看出，当加入更多样本时，只需向矩阵X中加入更多列。

所以从这里我们也可以了解到，为什么之前我们对单个样本的计算要写成 $z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$ 这种形式，因为当有不同的训练样本时，将它们堆到矩阵X的各列中，那么它们的输出也就会相应的堆叠到矩阵 $Z^{[1]}$ 的各列中。现在我们就可以直接计算矩阵 $Z^{[1]}$加上$b^{[1]}$，因为列向量 $b^{[1]}$ 和矩阵 $Z^{[1]}$的列向量有着相同的尺寸，而 **Python** 的广播机制对于这种矩阵与向量直接相加的处理方式是，将向量与矩阵的每一列相加。 所以这一节只是说明了为什么公式 $Z^{[1]} = W^{[1]}X + b^{[1]}$是前向传播的第一步计算的正确向量化实现，但事实

证明，类似的分析可以发现，前向传播的其它步也可以使用非常相似的逻辑，即如果将输入按列向量横向堆叠进矩阵，那么通过公式计算之后，也能得到成列堆叠的输出。

最后，对这一段视频的内容做一个总结:

由公式 3.12、公式 3.13、公式 3.14、公式 3.15 可以看出，使用向量化的方法，可以不需要显示循环，而直接通过矩阵运算从X就可以计算出 $A^{[1]}$，实际上X可以记为 $A^{[0]}$，使用同样的方法就可以由神经网络中的每一层的输入 $A^{[i-1]}$ 计算输出 $A^{[i]}$。其实这些方程有一定对称性，其中第一个方程也可以写成$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]}$，你看这对方程，还有这对方程形式其实很类似，只不过这里所有指标加了 1。所以这样就显示出神经网络的不同层次，你知道大概每一步做的都是一样的，或者只不过同样的计算不断重复而已。这里我们有一个双层神经网络，我们在下周视频里会讲深得多的神经网络，你看到随着网络的深度变大，基本上也还是重复这两步运算，只不过是比这里你看到的重复次数更多。在下周的视频中将会讲解更深层次的神经网络，随着层数的加深，基本上也还是重复同样的运算。

以上就是对神经网络向量化实现的正确性的解释，到目前为止，我们仅使用 **sigmoid** 函数作为激活函数，事实上这并非最好的选择，在下一个视频中，将会继续深入的讲解如何使用更多不同种类的激活函数。

3.6 激活函数（Activation functions）

使用一个神经网络时，需要决定使用哪种激活函数用隐藏层上，哪种用在输出节点上。到目前为止，之前的视频只用过 **sigmoid** 激活函数，但是，有时其他的激活函数效果会更好。

在神经网路的前向传播中，的$a^{[1]} = \sigma(z^{[1]})$和$a^{[2]} = \sigma(z^{[2]})$这两步会使用到 **sigmoid** 函数。**sigmoid** 函数在这里被称为激活函数。 公式 3.18： $a = \sigma(z) = \frac{1}{1+e^{-z}}$

更通常的情况下，使用不同的函数$g(z^{[1]})$，g可以是除了 **sigmoid** 函数意外的非线性函数。**tanh** 函数或者双曲正切函数是总体上都优于 **sigmoid** 函数的激活函数。

如图，$a = tan(z)$的值域是位于+1 和-1 之间。 公式 3.19： $a = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

事实上，**tanh** 函数是 **sigmoid** 的向下平移和伸缩后的结果。对它进行了变形后，穿过了$(0,0)$点，并且值域介于+1 和-1 之间。

结果表明，如果在隐藏层上使用函数 公式 3.20： $g(z^{[1]}) = tanh(z^{[1]})$ 效果总是优于 **sigmoid** 函数。因为函数值域在-1 和+1 的激活函数，其均值是更接近零均值的。在训练一个算法模型时，如果使用 **tanh** 函数代替 **sigmoid** 函数中心化数据，使得数据的平均值更接近 0 而不是 0.5.

这会使下一层学习简单一点，在第二门课中会详细讲解。

在讨论优化算法时，有一点要说明：我基本已经不用 **sigmoid** 激活函数了，**tanh** 函数在所有场合都优于 **sigmoid** 函数。

但有一个例外：在二分类的问题中，对于输出层，因为y的值是 0 或 1，所以想让$\hat{y}$的数值介于 0 和 1 之间，而不是在-1 和+1 之间。所以需要使用 **sigmoid** 激活函数。这里的 公式 3.21： $g(z^{[2]}) = \sigma(z^{[2]})$ 在这个例子里看到的是，对隐藏层使用 **tanh** 激活函数，输出层使用 **sigmoid** 函数。

所以，在不同的神经网络层中，激活函数可以不同。为了表示不同的激活函数，在不同的层中，使用方括号上标来指出g上标为[1]的激活函数，可能会跟g上标为[2]不同。方括号上标[1]代表隐藏层，方括号上标[2]表示输出层。

sigmoid 函数和 **tanh** 函数两者共同的缺点是，在z特别大或者特别小的情况下，导数的梯度或者函数的斜率会变得特别小，最后就会接近于 0，导致降低梯度下降的速度。

在机器学习另一个很流行的函数是：修正线性单元的函数（**ReLu**），**ReLu** 函数图像是如下图。 公式 3.22： $a = max(0, z)$ 所以，只要z是正值的情况下，导数恒等于 1，当z是负

值的时候，导数恒等于 0。从实际上来说，当使用z的导数时，z=0 的导数是没有定义的。但是当编程实现的时候，z的取值刚好等于 0.0000000，这个值相当小，所以，在实践中，不需要担心这个值，z是等于 0 的时候，假设一个导数是 1 或者 0 效果都可以。

这有一些选择激活函数的经验法则：

如果输出是 0、1 值（二分类问题），则输出层选择 **sigmoid** 函数，然后其它的所有单元都选择 **Relu** 函数。

这是很多激活函数的默认选择，如果在隐藏层上不确定使用哪个激活函数，那么通常会使用 Relu 激活函数。有时，也会使用 **tanh** 激活函数，但 **Relu** 的一个优点是：当z是负值的时候，导数等于 0。

这里也有另一个版本的 **Relu** 被称为 **Leaky Relu**。

当z是负值时，这个函数的值不是等于 0，而是轻微的倾斜，如图。

这个函数通常比 **Relu** 激活函数效果要好，尽管在实际中 **Leaky ReLu** 使用的并不多。

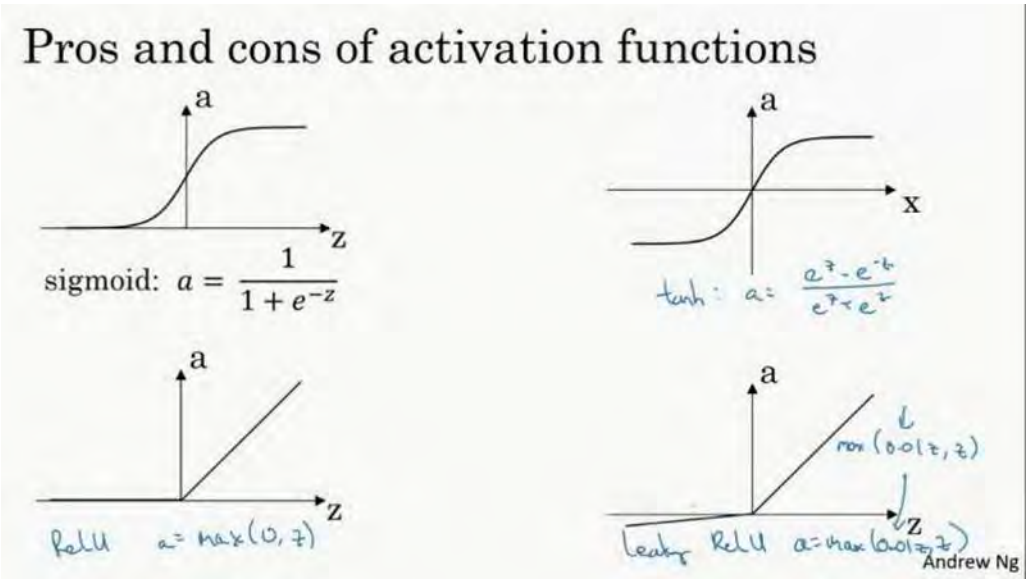


图 3.6.1

两者的优点是：

第一，在z的区间变动很大的情况下，激活函数的导数或者激活函数的斜率都会远大于 0，在程序实现就是一个 **if-else** 语句，而 **sigmoid** 函数需要进行浮点四则运算，在实践中，使用 **ReLu** 激活函数神经网络通常会比使用 **sigmoid** 或者 **tanh** 激活函数学习的更快。

第二，**sigmoid** 和 **tanh** 函数的导数在正负饱和区的梯度都会接近于 0，这会造成梯度弥散，而 **Relu** 和 **Leaky ReLu** 函数大于 0 部分都为常熟，不会产生梯度弥散现象。(同时应该注意到的是，**Relu** 进入负半区的时候，梯度为 0，神经元此时不会训练，产生所谓的稀疏性，

而 **Leaky ReLu** 不会有这问题)

z在 **ReLu** 的梯度一半都是 0，但是，有足够的隐藏层使得 z 值大于 0，所以对大多数的训练数据来说学习过程仍然可以很快。

快速概括一下不同激活函数的过程和结论。

sigmoid 激活函数：除了输出层是一个二分类问题基本不会用它。

tanh 激活函数：**tanh** 是非常优秀的，几乎适合所有场合。

ReLu 激活函数：最常用的默认函数，，如果不确定用哪个激活函数，就使用 **ReLu** 或者 **Leaky ReLu**。公式 3.23： $a = max(0.01z, z)$ 为什么常数是 0.01？当然，可以为学习算法选择不同的参数。

在选择自己神经网络的激活函数时，有一定的直观感受，在深度学习中的经常遇到一个问题：在编写神经网络的时候，会有很多选择：隐藏层单元的个数、激活函数的选择、初始化权值……这些选择想得到一个对比较好的指导原则是挺困难的。

鉴于以上三个原因，以及在工业界的见闻，提供一种直观的感受，哪一种工业界用的多，哪一种用的少。但是，自己的神经网络的应用，以及其特殊性，是很难提前知道选择哪些效果更好。所以通常的建议是：如果不确定哪一个激活函数效果更好，可以把它们都试试，然后在验证集或者发展集上进行评价。然后看哪一种表现的更好，就去使用它。

为自己的神经网络的应用测试这些不同的选择，会在以后检验自己的神经网络或者评估算法的时候，看到不同的效果。如果仅仅遵守使用默认的 **ReLu** 激活函数，而不要用其他的激励函数，那就可能在近期或者往后，每次解决问题的时候都使用相同的办法。

3.7 为什么需要非线性激活函数？（why need a nonlinear activation function?）

为什么神经网络需要非线性激活函数？事实证明：要让你的神经网络能够计算出有趣的函数，你必须使用非线性激活函数，证明如下：

这是神经网络正向传播的方程，现在我们去掉函数g，然后令$a^{[1]}=z^{[1]}$，或者我们也可以令$g(z)=z$，这个有时被叫做线性激活函数（更学术点的名字是恒等激励函数，因为它们就是把输入值输出）。为了说明问题我们把$a^{[2]}=z^{[2]}$，那么这个模型的输出y或仅仅只是输入特征x的线性组合。

如果我们改变前面的式子，令： (1) $a^{[1]}=z^{[1]}=W^{[1]}x+b^{[1]}$

(2) $a^{[1]}=z^{[2]}=W^{[2]}a^{[1]}+b^{[2]}$ 将式子(1)代入式子(2)中，则：$a^{[2]}=z^{[2]}=W^{[2]}(W^{[1]}x+b^{[1]})+b^{[2]}$

(3) $a^{[2]}=z^{[2]}=W^{[2]}W^{[1]}x+W^{[2]}b^{[1]}+b^{[2]}$ 简化多项式得 $a^{[2]}=z^{[2]}=W'x+b'$ 如果你是用线性激活函数或者叫恒等激励函数，那么神经网络只是把输入线性组合再输出。

我们稍后会谈到深度网络，有很多层的神经网络，很多隐藏层。事实证明，如果你使用线性激活函数或者没有使用一个激活函数，那么无论你的神经网络有多少层一直在做的只是计算线性函数，所以不如直接去掉全部隐藏层。在我们的简明案例中，事实证明如果你在隐藏层用线性激活函数，在输出层用 **sigmoid** 函数，那么这个模型的复杂度和没有任何隐藏层的标准 **Logistic** 回归是一样的，如果你愿意的话，可以证明一下。

在这里线性隐层一点用也没有，因为这两个线性函数的组合本身就是线性函数，所以除非你引入非线性，否则你无法计算更有趣的函数，即使你的网络层数再多也不行；只有一个地方可以使用线性激活函数------$g(z)=z$，就是你在做机器学习中的回归问题。y 是一个实数，举个例子，比如你想预测房地产价格，y 就不是二分类任务 0 或 1，而是一个实数，从 0 到正无穷。如果y 是个实数，那么在输出层用线性激活函数也许可行，你的输出也是一个实数，从负无穷到正无穷。

总而言之，不能在隐藏层用线性激活函数，可以用 **ReLU** 或者 **tanh** 或者 **leaky ReLU** 或者其他的非线性激活函数，唯一可以用线性激活函数的通常就是输出层；除了这种情况，会在隐层用线性函数的，除了一些特殊情况，比如与压缩有关的，那方面在这里将不深入讨论。在这之外，在隐层使用线性激活函数非常少见。因为房价都是非负数，所以我们也可以在输

出层使用 **ReLU** 函数这样你的$\hat{y}$都大于等于 0。

理解为什么使用非线性激活函数对于神经网络十分关键，接下来我们讨论梯度下降，并在下一个视频中开始讨论梯度下降的基础——激活函数的导数。

3.8 激活函数的导数（Derivatives of activation functions）

在神经网络中使用反向传播的时候，你真的需要计算激活函数的斜率或者导数。针对以下四种激活，求其导数如下：

1）**sigmoid activation function**

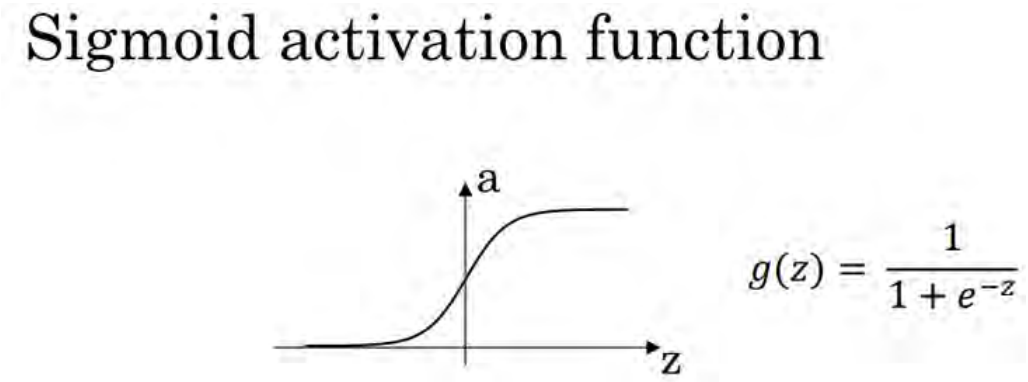


图 3.8.1

其具体的求导如下： 公式 3.25： $\frac{d}{dz}g(z) = \frac{1}{1+e^{-z}}(1 - \frac{1}{1+e^{-z}}) = g(z)(1-g(z))$

注：

当z = 10 或$z = -10$ $\frac{d}{dz}g(z) \approx 0$

当z= 0 $\frac{d}{dz}g(z) = \mathrm{g(z)(1-g(z))} = 1/4$

在神经网络中$a = g(z)$; $g(z)' = \frac{d}{dz}g(z) = a(1-a)$

2) **Tanh activation function**

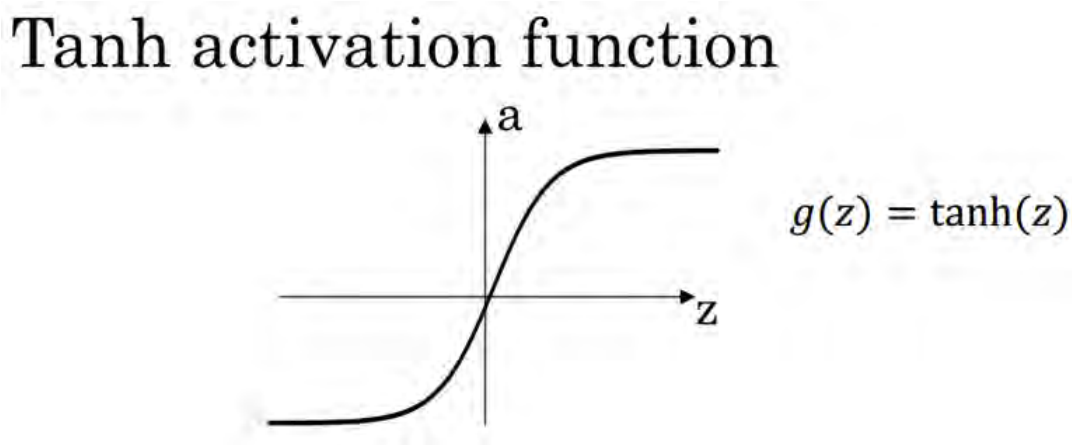


图 3.8.2

其具体的求导如下： 公式 3.26： $g(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

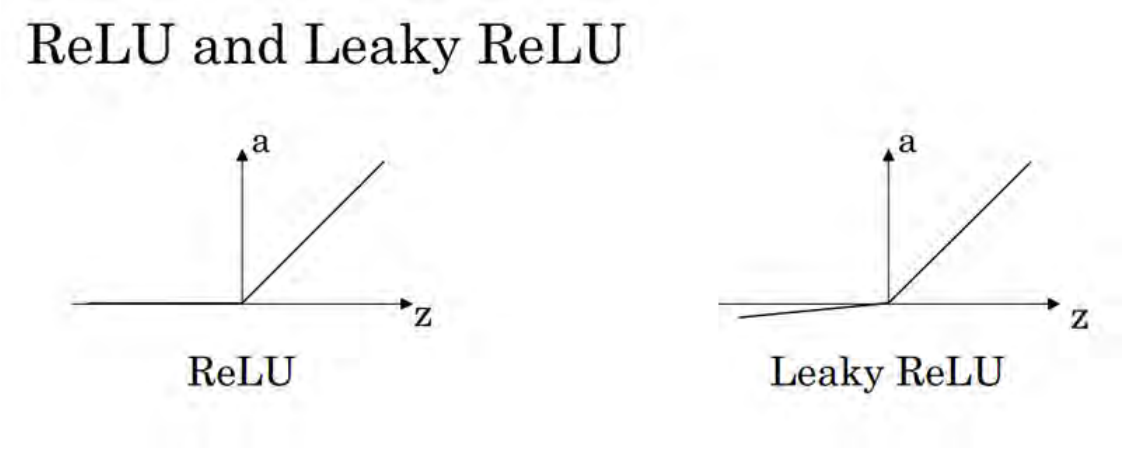
公式 3.27： $\frac{d}{\mathrm{d}z}g(z)=1-(tanh(z))^2$ 注：

当z =10 或$z=-10$ $\frac{d}{dz}g(z)\approx 0$

当z =0 $\frac{d}{dz}g(z)=1-(0)=1$

在神经网络中;

3）**Rectified Linear Unit (ReLU)**



$g(z)=max(0,z)$

$$g(z)'=\begin{cases}0 & \text{if z} < 0\\ 1 & \text{if z} > 0\\ undefined & \text{if z} = 0\end{cases}$$

注：通常在z= 0 的时候给定其导数 1,0；当然z=0 的情况很少

4）**Leaky linear unit (Leaky ReLU)**

与 **ReLU** 类似

$$g(z)=\max(0.01,z)$$

$$g(z)'=\begin{cases}0.01z & \text{if z} < 0\\ 1 & \text{if z} > 0\\ undefined & \text{if z} = 0\end{cases}$$

注：通常在$z=0$的时候给定其导数 1,0.01；当然$z=0$的情况很少

3.9 神经网络的梯度下降（Gradient descent for neural networks）

在这个视频中，我会给你实现反向传播或者说梯度下降算法的方程组，在下一个视频我们会介绍为什么这几个特定的方程是针对你的神经网络实现梯度下降的正确方程。

你的单隐层神经网络会有$W^{[1]}$，$b^{[1]}$，$W^{[2]}$，$b^{[2]}$这些参数，还有个n_x表示输入特征的个数，$n^{[1]}$表示隐藏单元个数，$n^{[2]}$表示输出单元个数。

在我们的例子中，我们只介绍过的这种情况，那么参数:

矩阵$W^{[1]}$的维度就是$(n^{[1]}, n^{[0]})$，$b^{[1]}$就是$n^{[1]}$维向量，可以写成$(n^{[1]}, 1)$，就是一个的列向量。 矩阵$W^{[2]}$的维度就是$(n^{[2]}, n^{[1]})$，$b^{[2]}$的维度就是$(n^{[2]}, 1)$维度。

你还有一个神经网络的成本函数，假设你在做二分类任务，那么你的成本函数等于：

Cost function: 公式 3.28： $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{n} L(\hat{y}, y)$

loss function 和之前做 **logistic** 回归完全一样。

训练参数需要做梯度下降，在训练神经网络的时候，随机初始化参数很重要，而不是初始化成全零。当你参数初始化成某些值后，每次梯度下降都会循环计算以下预测值：

$\hat{y}^{(i)}, (i = 1,2, \ldots, m)$

公式 3.28： $dW^{[1]} = \frac{dJ}{dW^{[1]}}, db^{[1]} = \frac{dJ}{db^{[1]}}$

公式 3.29： $dW^{[2]} = \frac{dJ}{dW^{[2]}}, db^{[2]} = \frac{dJ}{db^{[2]}}$

其中

公式 3.30： $W^{[1]} \Longrightarrow W^{[1]} - adW^{[1]}, b^{[1]} \Longrightarrow b^{[1]} - adb^{[1]}$

公式 3.31： $W^{[2]} \Rightarrow W^{[2]} - adW^{[2]}, b^{[2]} \Rightarrow b^{[2]} - adb^{[2]}$

正向传播方程如下（之前讲过）：

forward propagation：

(1) $z^{[1]} = W^{[1]}x + b^{[1]}$

(2) $a^{[1]} = \sigma(z^{[1]})$

(3) $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

(4) $a^{[2]} = g^{[2]}(z^{[z]}) = \sigma(z^{[2]})$

反向传播方程如下:

back propagation：

公式 3.32： $dz^{[1]} = A^{[2]} - Y, Y = [y^{[1]} \quad y^{[2]} \quad \cdots \quad y^{[m]}]$

公式 3.33： $dW^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$

公式 3.34： $db^{[2]} = \frac{1}{m} np.sum(dz^{[2]}, axis = 1, keepdims = True)$

公式 3.35：

$$dz^{[1]} = \underbrace{W^{[2]T} \mathrm{d}z^{[2]}}_{(n^{[1]},m)} * \underbrace{g^{[1]'}}_{activation\ function\ of\ hidden\ layer} * \underbrace{(z^{[1]})}_{(n^{[1]},m)}$$

公式 3.36： $dW^{[1]} = \frac{1}{m} dz^{[1]} x^T$

公式 3.37： $\underset{(n^{[1]},1)}{db^{[1]}} = \frac{1}{m} np.sum(dz^{[1]}, axis = 1, keepdims = True)$

上述是反向传播的步骤，注：这些都是针对所有样本进行过向量化，Y是$1 \times m$的矩阵；这里 `np.sum` 是 python 的 numpy 命令，`axis=1` 表示水平相加求和，`keepdims` 是防止 python 输出那些古怪的秩数$(n,)$，加上这个确保阵矩阵$db^{[2]}$这个向量输出的维度为$(n,1)$这样标准的形式。

目前为止，我们计算的都和 **Logistic** 回归十分相似，但当你开始计算反向传播时，你需要计算，是隐藏层函数的导数，输出在使用 **sigmoid** 函数进行二元分类。这里是进行逐个元素乘积，因为$W^{[2]T} dz^{[2]}$和$(z^{[1]})$这两个都为$(n^{[1]}, m)$矩阵；

还有一种防止 **python** 输出奇怪的秩数，需要显式地调用 `reshape` 把 `np.sum` 输出结果写成矩阵形式。

以上就是正向传播的 4 个方程和反向传播的 6 个方程，这里我是直接给出的，在下个视频中，我会讲如何导出反向传播的这 6 个式子的。如果你要实现这些算法，你必须正确执行正向和反向传播运算，你必须能计算所有需要的导数，用梯度下降来学习神经网络的参数；你也可以许多成功的深度学习从业者一样直接实现这个算法，不去了解其中的知识。

3.10（选修）直观理解反向传播（Backpropagation intuition）

这个视频主要是推导反向传播。

下图是逻辑回归的推导：

回想一下逻辑回归的公式(参考公式 3.2、公式 3.5、公式 3.6、公式 3.15) 公式 3.38：

$$\left.\begin{matrix}x\\w\\b\end{matrix}\right\}\Longrightarrow z=w^Tx+b\Longrightarrow a=\sigma(z)\Longrightarrow L(a,y)$$

所以回想当时我们讨论逻辑回归的时候，我们有这个正向传播步骤，其中我们计算z，然后a，然后损失函数L。

公式 3.39：

$$\left.\begin{matrix}x\\w\\b\end{matrix}\right\}\quad\Leftarrow\quad z=w^Tx+b$$

$$dw=dz\cdot x,db=dz \qquad dz=da\cdot g'(z),g(z)=\sigma(z),\frac{dL}{dz}=\frac{dL}{da}\cdot\frac{da}{dz},\frac{d}{dz}g(z)=g'(z)$$

$$\Leftarrow\quad a=\sigma(z)\Leftarrow L(a,y)$$

$$da=\frac{d}{da}L(a,y)=(-y\log a-(1-y)\log(1-a))'=-\frac{y}{a}+\frac{1-y}{1-a}$$

神经网络的计算中，与逻辑回归十分类似，但中间会有多层的计算。下图是一个双层神经网络，有一个输入层，一个隐藏层和一个输出层。

前向传播：

计算$z^{[1]}$，$a^{[1]}$，再计算$z^{[2]}$，$a^{[2]}$，最后得到 **loss function**。

反向传播：

向后推算出$da^{[2]}$，然后推算出$dz^{[2]}$，接着推算出$da^{[1]}$，然后推算出$dz^{[1]}$。我们不需要对x求导，因为x是固定的，我们也不是想优化x。向后推算出$da^{[2]}$，然后推算出$dz^{[2]}$的步骤可以合为一步： 公式 3.40： $dz^{[2]}=a^{[2]}-y$ ， $dW^{[2]}=dz^{[2]}a^{[1]^T}$ (注意：逻辑回归中；为什么$a^{[1]T}$多了个转置：dw中的W(视频里是$W_i^{[2]}$)是一个列向量，而$W^{[2]}$是个行向量，故需要加个转置); 公式 3.41： $db^{[2]}=dz^{[2]}$ 公式 3.42： $dz^{[1]}=W^{[2]T}dz^{[2]}*g[1]'(z^{[1]})$ 注意：这里的矩阵：$W^{[2]}$的维度是：$(n^{[2]},n^{[1]})$。

$z^{[2]}$ ， $dz^{[2]}$的维度都是：$(n^{[2]},1)$，如果是二分类，那维度就是$(1,1)$。

$z^{[1]}$，$dz^{[1]}$的维度都是：$(n^{[1]},1)$。

证明过程： 见公式 3.42，

其中$W^{[2]T}dz^{[2]}$维度为：$(n^{[1]},n^{[2]})$、$(n^{[2]},1)$相乘得到$(n^{[1]},1)$，和$z^{[1]}$维度相同，$g[1]'(z^{[1]})$的维度为$(n^{[1]},1)$，这就变成了两个都是$(n^{[1]},1)$向量逐元素乘积。

实现后向传播有个技巧，就是要保证矩阵的维度相互匹配。最后得到$dW^{[1]}$和$db^{[1]}$，公式 3.43： $dW^{[1]} = dz^{[1]}x^T, db^{[1]} = dz^{[1]}$

可以看出$dW^{[1]}$ 和$dW^{[2]}$ 非常相似，其中x扮演了$a^{[0]}$的角色，x^T 等同于$a^{[0]T}$。

由： $Z^{[1]} = W^{[1]}x + b^{[1]}$ ， $a^{[1]} = g^{[1]}(Z^{[1]})$ 得到： $Z^{[1]} = W^{[1]}x + b^{[1]}, A^{[1]} = g^{[1]}(Z^{[1]})$

$$Z^{[1]} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ z^{[1](1)} & z^{[1](2)} & \vdots & z^{[1](m)} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

注意：大写的$Z^{[1]}$表示$z^{[1](1)}, z^{[1](2)}, z^{[1](3)} \ldots z^{[1](m)}$的列向量堆叠成的矩阵，以下类同。

下图写了主要的推导过程：

公式 3.44： $dZ^{[2]} = A^{[2]} - Y$ ， $dW^{[2]} = \frac{1}{m}dZ^{[2]}A^{[1]^T}$

公式 3.45： $L = \frac{1}{m}\sum_i^n L(\hat{y}, y)$

公式 3.46： $db^{[2]} = \frac{1}{m}np.sum(dZ^{[2]}, axis = 1, keepdims = True)$

公式 3.47： $\underset{(n^{[1]},m)}{dZ^{[1]}} = \underset{(n^{[1]},m)}{W^{[2]T}dZ^{[2]}} * \underset{(n^{[1]},m)}{g[1]'(Z^{[1]})}$

公式 3.48： $dW^{[1]} = \frac{1}{m}dZ^{[1]}x^T$

公式 3.49： $db^{[1]} = \frac{1}{m}np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

吴恩达老师认为反向传播的推导是机器学习领域最难的数学推导之一，矩阵的导数要用链式法则来求，如果这章内容掌握不了也没大的关系，只要有这种直觉就可以了。还有一点，就是初始化你的神经网络的权重，不要都是 0，而是随机初始化，下一章将详细介绍原因。

3.11 随机初始化（Random+Initialization）

当你训练神经网络时，权重随机初始化是很重要的。对于逻辑回归，把权重初始化为 0 当然也是可以的。但是对于一个神经网络，如果你把权重或者参数都初始化为 0，那么梯度下降将不会起作用。

让我们看看这是为什么。有两个输入特征，$n^{[0]}=2$，2 个隐藏层单元$n^{[1]}$就等于 2。 因此与一个隐藏层相关的矩阵，或者说$W^{[1]}$是 2*2 的矩阵，假设把它初始化为 0 的 2*2 矩阵，$b^{[1]}$也等于 $[0\ 0]^T$，把偏置项b初始化为 0 是合理的，但是把w初始化为 0 就有问题了。 那这个问题如果按照这样初始化的话，你总是会发现$a_1^{[1]}$ 和 $a_2^{[1]}$相等，这个激活单元和这个激活单元就会一样。因为两个隐含单元计算同样的函数，当你做反向传播计算时，这会导致 $\mathrm{dz}_1^{[1]}$ 和 $\mathrm{dz}_2^{[1]}$也会一样，对称这些隐含单元会初始化得一样，这样输出的权值也会一模一样，由此$W^{[2]}$等于$[0\ 0]$；

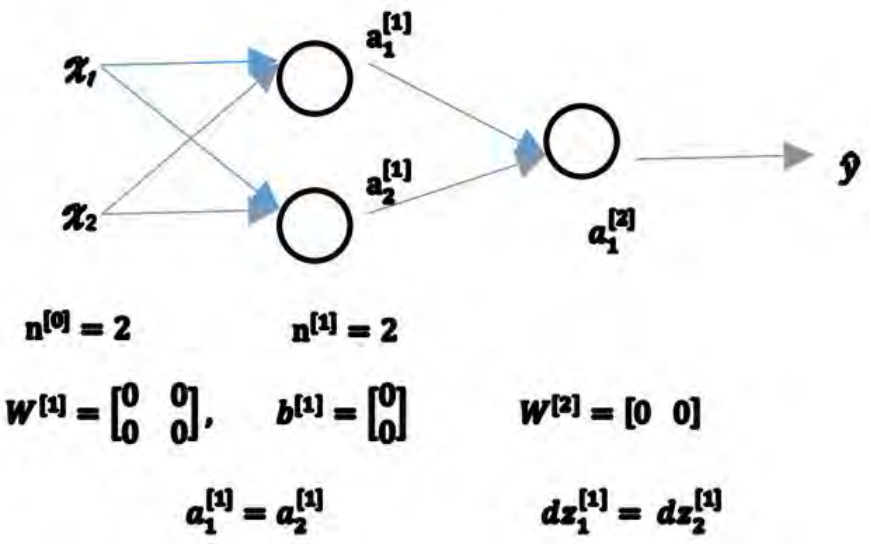


图 3.11.1 但是如果你这样初始化这个神经网络，那么这两个隐含单元就会完全一样，因此他们完全对称，也就意味着计算同样的函数，并且肯定的是最终经过每次训练的迭代，这两个隐含单元仍然是同一个函数，令人困惑。dW会是一个这样的矩阵，每一行有同样的值因此我们做权重更新把权重$W^{[1]}\Longrightarrow W^{[1]}-adW$每次迭代后的$W^{[1]}$，第一行等于第二行。

由此可以推导，如果你把权重都初始化为 0，那么由于隐含单元开始计算同一个函数，所有的隐含单元就会对输出单元有同样的影响。一次迭代后同样的表达式结果仍然是相同的，即隐含单元仍是对称的。通过推导，两次、三次、无论多少次迭代，不管你训练网络多长时间，隐含单元仍然计算的是同样的函数。因此这种情况下超过 1 个隐含单元也没什么意

义，因为他们计算同样的东西。当然更大的网络，比如你有 3 个特征，还有相当多的隐含单元。

如果你要初始化成 0，由于所有的隐含单元都是对称的，无论你运行梯度下降多久，他们一直计算同样的函数。这没有任何帮助，因为你想要两个不同的隐含单元计算不同的函数，这个问题的解决方法就是随机初始化参数。你应该这么做：把 $W^{[1]}$ 设为 `np.random.randn(2,2)`(生成高斯分布)，通常再乘上一个小的数，比如 0.01，这样把它初始化为很小的随机数。然后b没有这个对称的问题（叫做 **symmetry breaking problem**），所以可以把 b 初始化为 0，因为只要随机初始化W你就有不同的隐含单元计算不同的东西，因此不会有 **symmetry breaking** 问题了。相似的，对于$W^{[2]}$你可以随机初始化，$b^{[2]}$可以初始化为 0。

$W^{[1]} = np.random.randn(2,2) * 0.01$,

$b^{[1]} = np.zeros((2,1))$

$W^{[2]} = np.random.randn(2,2) * 0.01$, $b^{[2]} = 0$

你也许会疑惑，这个常数从哪里来，为什么是 0.01，而不是 100 或者 1000。我们通常倾向于初始化为很小的随机数。因为如果你用 **tanh** 或者 **sigmoid** 激活函数，或者说只在输出层有一个 **Sigmoid**，如果（数值）波动太大，当你计算激活值时$z^{[1]} = W^{[1]}x + b^{[1]}$, $a^{[1]} = \sigma(z^{[1]}) = g^{[1]}(z^{[1]})$如果$W$很大，$z$就会很大。$z$的一些值$a$就会很大或者很小，因此这种情况下你很可能停在 **tanh/sigmoid** 函数的平坦的地方(见图 3.8.2)，这些地方梯度很小也就意味着梯度下降会很慢，因此学习也就很慢。

回顾一下：如果w很大，那么你很可能最终停在（甚至在训练刚刚开始的时候）z很大的值，这会造成 **tanh/Sigmoid** 激活函数饱和在龟速的学习上，如果你没有 **sigmoid/tanh** 激活函数在你整个的神经网络里，就不成问题。但如果你做二分类并且你的输出单元是 **Sigmoid** 函数，那么你不会想让初始参数太大，因此这就是为什么乘上 0.01 或者其他一些小数是合理的尝试。对于$w^{[2]}$一样，就是 `np.random.randn((1,2))`，我猜会是乘以 0.01。

事实上有时有比 0.01 更好的常数，当你训练一个只有一层隐藏层的网络时（这是相对浅的神经网络，没有太多的隐藏层），设为 0.01 可能也可以。但当你训练一个非常非常深的神经网络，你可能会选择一个不同于的常数而不是 0.01。下一节课我们会讨论怎么并且何时去选择一个不同于 0.01 的常数，但是无论如何它通常都会是个相对小的数。

好了，这就是这周的视频。你现在已经知道如何建立一个一层的神经网络了，初始化参数，用前向传播预测，还有计算导数，结合反向传播用在梯度下降中。

第四周：深层神经网络(Deep Neural Networks)

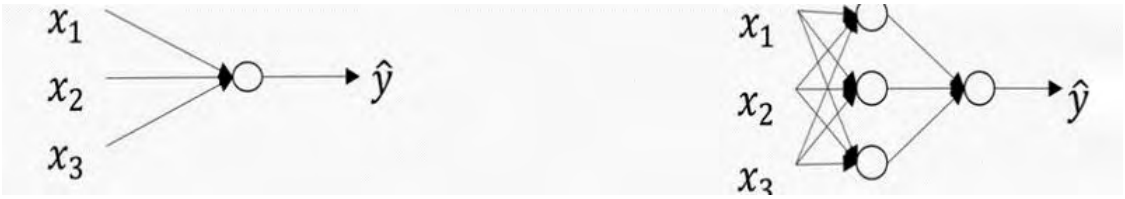
4.1 深层神经网络（Deep L-layer neural network）

目前为止我们学习了只有一个单独隐藏层的神经网络的正向传播和反向传播，还有逻辑回归，并且你还学到了向量化，这在随机初始化权重时是很重要。

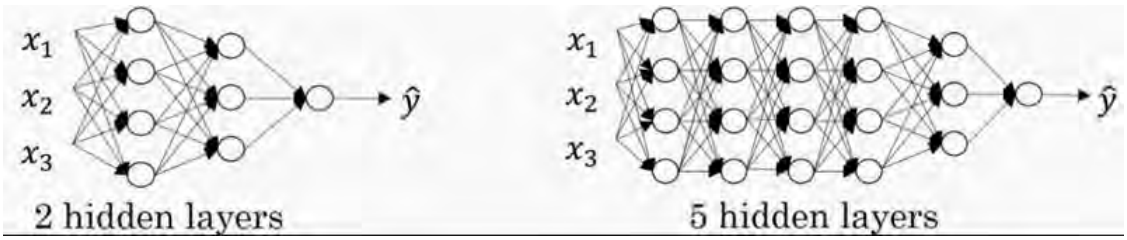
本周所要做的是把这些理念集合起来，就可以执行你自己的深度神经网络。

复习下前三周的课的内容：

1.逻辑回归，结构如下图左边。一个隐藏层的神经网络，结构下图右边：



注意，神经网络的层数是这么定义的：**从左到右，由 0 开始定义**，比如上边右图，x_1、x_2、x_3,这层是第 0 层，这层左边的隐藏层是第 1 层，由此类推。如下图左边是两个隐藏层的神经网络，右边是 5 个隐藏层的神经网络。

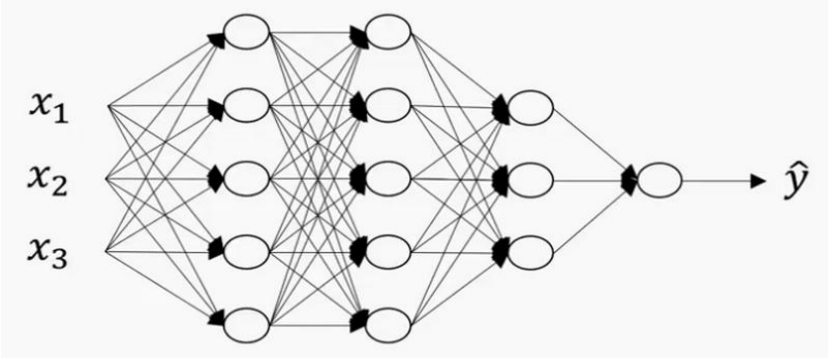


严格上来说逻辑回归也是一个一层的神经网络，而上边右图一个深得多的模型，浅与深仅仅是指一种程度。记住以下要点：

有一个隐藏层的神经网络，就是一个两层神经网络。记住当我们算神经网络的层数时，我们不算输入层，我们只算隐藏层和输出层。

但是在过去的几年中，**DLI**（深度学习学院 **deep learning institute**）已经意识到有一些函数，只有非常深的神经网络能学会，而更浅的模型则办不到。尽管对于任何给定的问题很难去提前预测到底需要多深的神经网络，所以先去尝试逻辑回归，尝试一层然后两层隐含层，然后把隐含层的数量看做是另一个可以自由选择大小的超参数，然后再保留交叉验证数据上评估，或者用你的开发集来评估。

我们再看下深度学习的符号定义：



上图是一个四层的神经网络，有三个隐藏层。我们可以看到，第一层（即左边数过去第二层，因为输入层是第 0 层）有 5 个神经元数目，第二层 5 个，第三层 3 个。

我们用 L 表示层数，上图：$L=4$，输入层的索引为“0”，第一个隐藏层$n^{[1]}=5$,表示有 5 个隐藏神经元，同理$n^{[2]}=5$，$n^{[3]}=3$，$n^{[4]}$=$n^{[L]}=1$（输出单元为 1）。而输入层，$n^{[0]}=n_x=3$。

在不同层所拥有的神经元的数目，对于每层 *l* 都用$a^{[l]}$来记作 *l* 层激活后结果，我们会在后面看到在正向传播时，最终能你会计算出$a^{[l]}$。

通过用激活函数 g 计算$z^{[l]}$，激活函数也被索引为层数l，然后我们用$w^{[l]}$来记作在 *l* 层计算$z^{[l]}$值的权重。类似的，$z^{[l]}$里的方程$b^{[l]}$也一样。

最后总结下符号约定：

输入的特征记作x，但是x同样也是 0 层的激活函数，所以$x=a^{[0]}$。

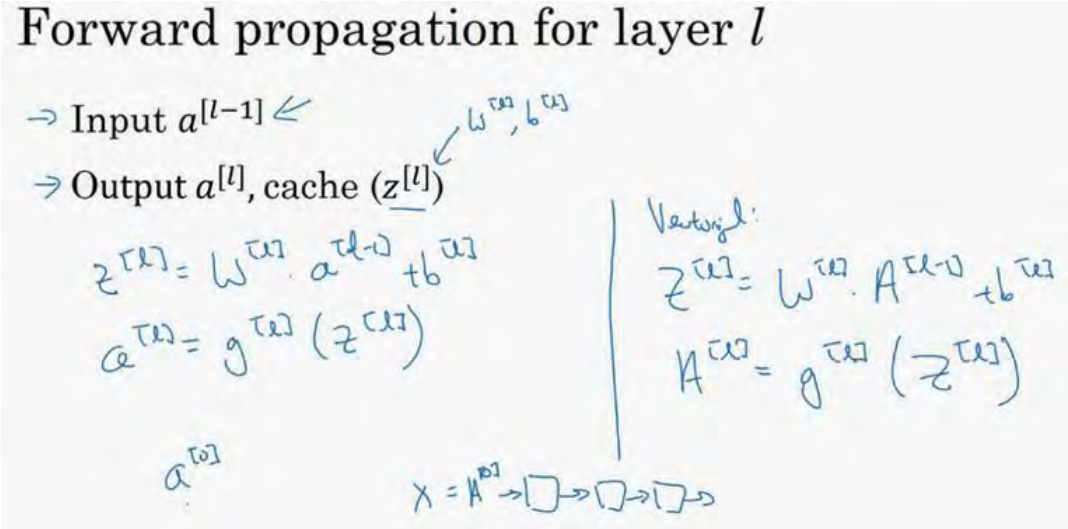
最后一层的激活函数，所以$a^{[L]}$是等于这个神经网络所预测的输出结果。

但是如果你忘记了某些符号的意义，请看笔记最后的附件：《深度学习符号指南》。

4.2 前向传播和反向传播（Forward and backward propagation）

之前我们学习了构成深度神经网络的基本模块，比如每一层都有前向传播步骤以及一个相反的反向传播步骤，这次视频我们讲讲如何实现这些步骤。

先讲前向传播，输入$a^{[l-1]}$，输出是$a^{[l]}$，缓存为$z^{[l]}$；从实现的角度来说我们可以缓存下$w^{[l]}$和$b^{[l]}$，这样更容易在不同的环节中调用函数。



所以前向传播的步骤可以写成： $z^{[l]} = W^{[l]} \cdot a^{[l]} + b^{[l]}$

$a^{[l]} = g^{[l]}(z^{[l]})$

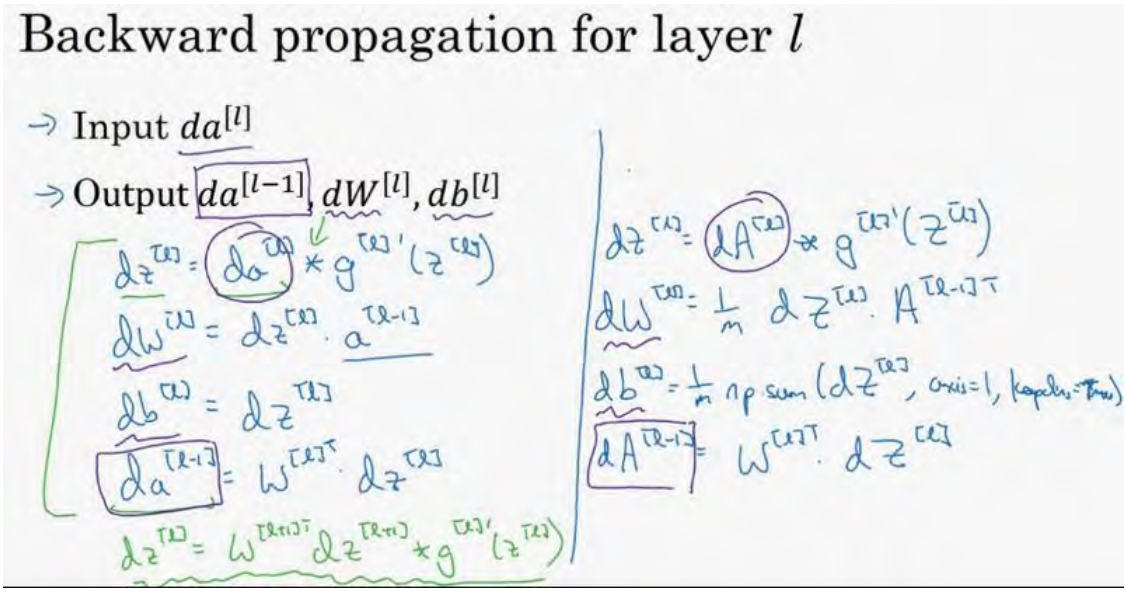
向量化实现过程可以写成： $z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$

$A^{[l]} = g^{[l]}(Z^{[l]})$

前向传播需要喂入$A^{[0]}$也就是X，来初始化；初始化的是第一层的输入值。$a^{[0]}$对应于一个训练样本的输入特征，而$A^{[0]}$对应于一整个训练样本的输入特征，所以这就是这条链的第一个前向函数的输入，重复这个步骤就可以从左到右计算前向传播。

下面讲反向传播的步骤：

输入为$da^{[l]}$，输出为$da^{[l-1]}$，$dw^{[l]}$，$db^{[l]}$



所以反向传播的步骤可以写成：

（1）$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$

（2）$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]}$

（3）$db^{[l]} = dz^{[l]}$

（4）$da^{[l-1]} = w^{[l]T} \cdot z^{[l]}$

（5）$dz^{[l]} = w^{[l+1]T} dz^{[l+1]} \cdot \ g^{[l]\prime}(z^{[l]})$

式子（5）由式子（4）带入式子（1）得到，前四个式子就可实现反向函数。

向量化实现过程可以写成：

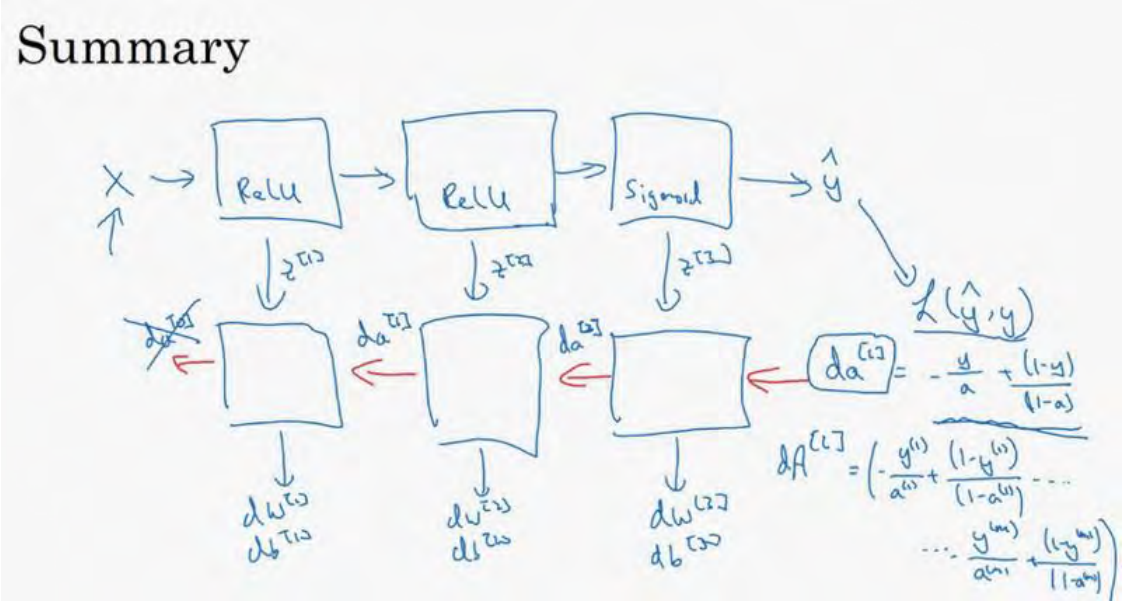
（6）$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$

（7）$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$

（8）$db^{[l]} = \frac{1}{m} \ np.sum(dz^{[l]}, axis = 1, keepdims = True)$

（9）$dA^{[l-1]} = W^{[l]T}.dZ^{[l]}$

总结一下：



第一层你可能有一个 **ReLU** 激活函数，第二层为另一个 **ReLU** 激活函数，第三层可能是 **sigmoid** 函数（如果你做二分类的话），输出值为，用来计算损失；这样你就可以向后迭代进行反向传播求导来求$dw^{[3]}$，$db^{[3]}$ ，$dw^{[2]}$ ，$db^{[2]}$ ，$dw^{[1]}$ ，$db^{[1]}$。在计算的时候，缓存会把$z^{[1]}$ $z^{[2]}z^{[3]}$传递过来，然后回传$da^{[2]}$，$da^{[1]}$ ，可以用来计算$da^{[0]}$，但我们不会使用它，这里讲述了一个三层网络的前向和反向传播，还有一个细节没讲就是前向递归——用输入数据来初始化，那么反向递归（使用 **Logistic** 回归做二分类）——对$A^{[l]}$ 求导。

忠告：补补微积分和线性代数，多推导，多实践。

4.3 深层网络中的前向传播（Forward propagation in a Deep Network）

跟往常一样，我们先来看对其中一个训练样本 x 如何应用前向传播，之后讨论向量化的版本。

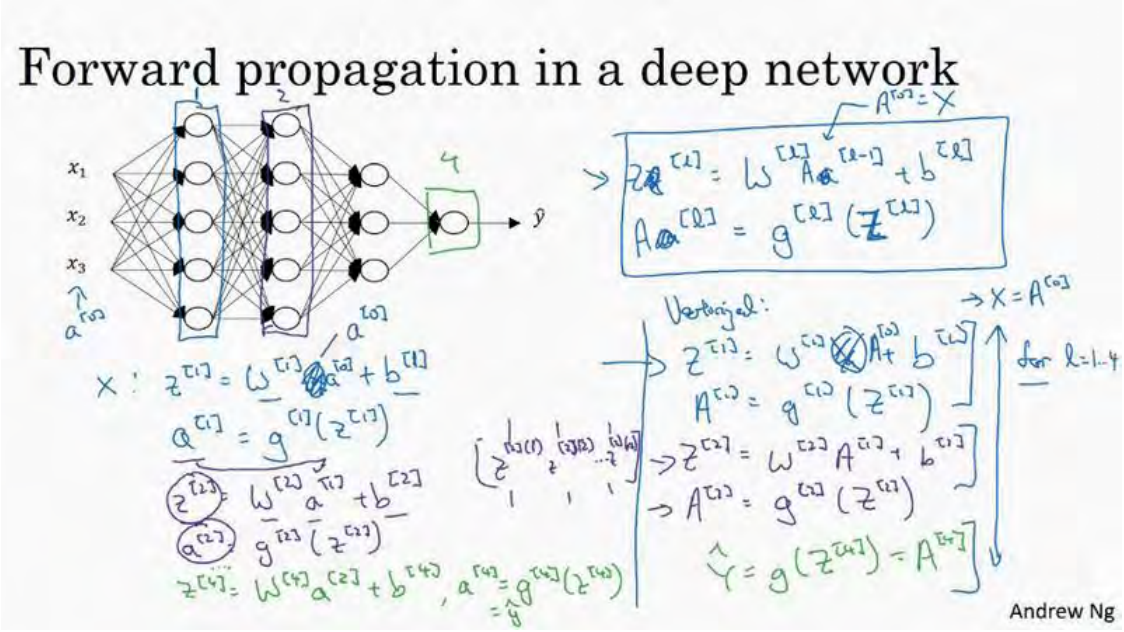
第一层需要计算$z^{[1]} = w^{[1]}x + b^{[1]}$，$a^{[1]} = g^{[1]}(z^{[1]})$（$x$可以看做$a^{[0]}$）

第二层需要计算$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$，$a^{[2]} = g^{[2]}(z^{[2]})$

以此类推，

第四层为$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$，$a^{[4]} = g^{[4]}(z^{[4]})$

前向传播可以归纳为多次迭代$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$，$a^{[l]} = g^{[l]}(z^{[l]})$。



向量化实现过程可以写成：

$Z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$，$A^{[l]} = g^{[l]}(Z^{[l]})$ $(A^{[0]} = X)$

这里只能用一个显式 **for** 循环，l从 1 到L，然后一层接着一层去计算。下一节讲的是避免代码产生 BUG，我所做的其中一件非常重要的工作。

4.4 核对矩阵的维数（Getting your matrix dimensions right）

当实现深度神经网络的时候，其中一个我常用的检查代码是否有错的方法就是拿出一张纸过一遍算法中矩阵的维数。

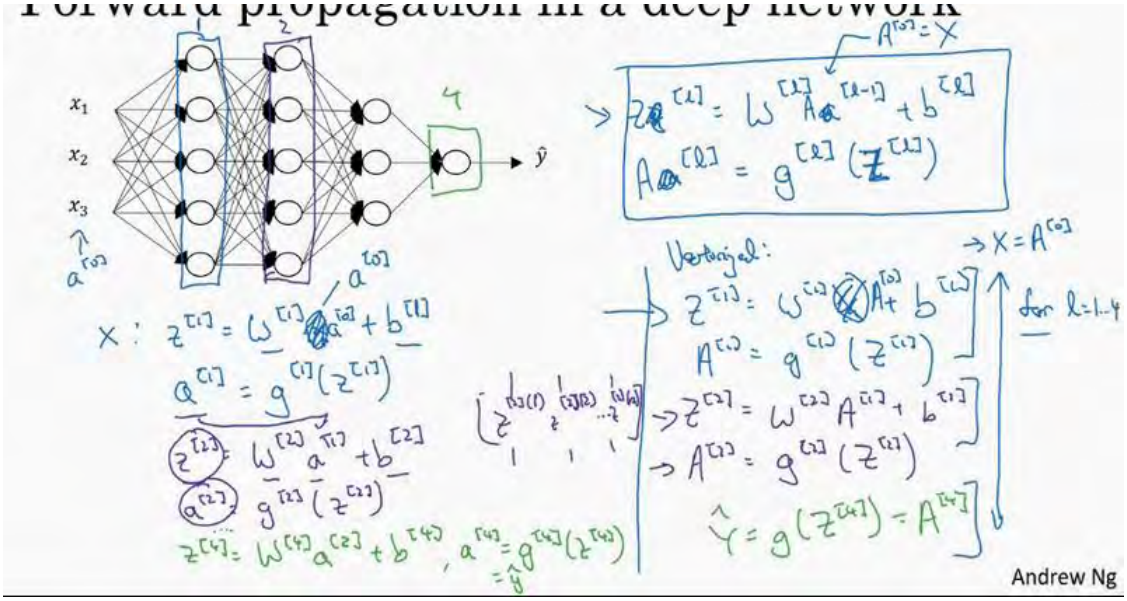
w的维度是（下一层的维数，前一层的维数），即$w^{[l]}$: $(n^{[l]},n^{[l-1]})$；

b的维度是（下一层的维数，1），即:

$b^{[l]}$: $(n^{[l]},1)$；

$z^{[l]},a^{[l]}$: $(n^{[l]},1)$;

$dw^{[l]}$和$w^{[l]}$维度相同，$db^{[l]}$和$b^{[l]}$维度相同，且w和b向量化维度不变，但z,a以及x的维度会向量化后发生变化。

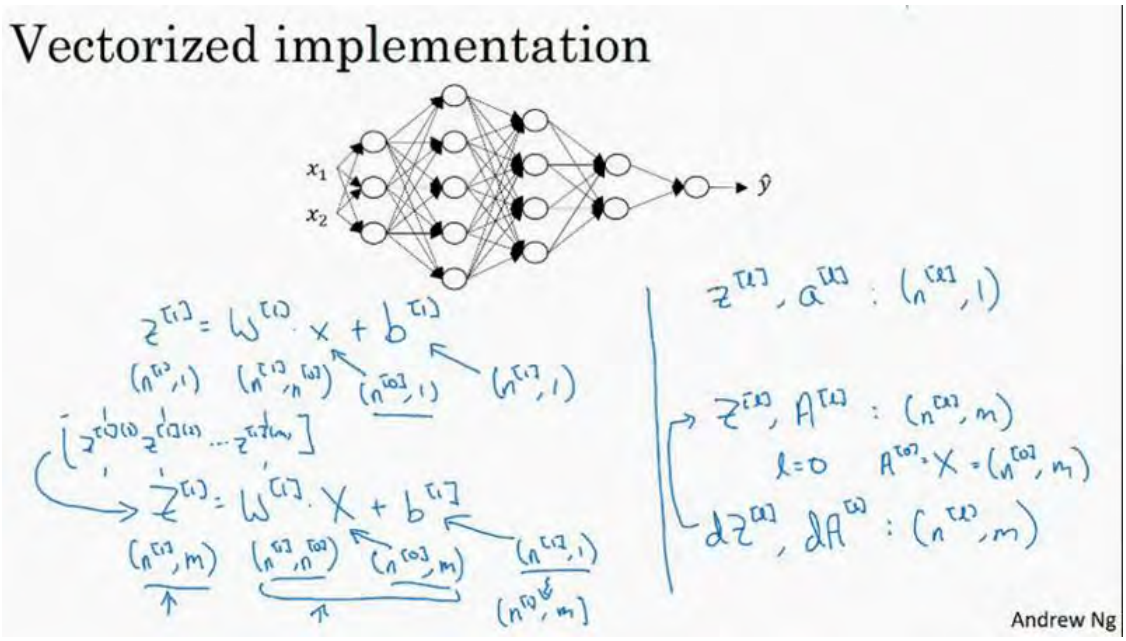


向量化后：

$Z^{[l]}$可以看成由每一个单独的$Z^{[l]}$叠加而得到，$Z^{[l]} = (z^{[l][1]}，z^{[l][2]}，z^{[l][3]}，\ldots，z^{[l][m]})$，

m为训练集大小，所以$Z^{[l]}$的维度不再是$(n^{[l]},1)$，而是$(n^{[l]},m)$。

$A^{[l]}$：$(n^{[l]},m)$，$A^{[0]} = X = (n^{[l]},m)$

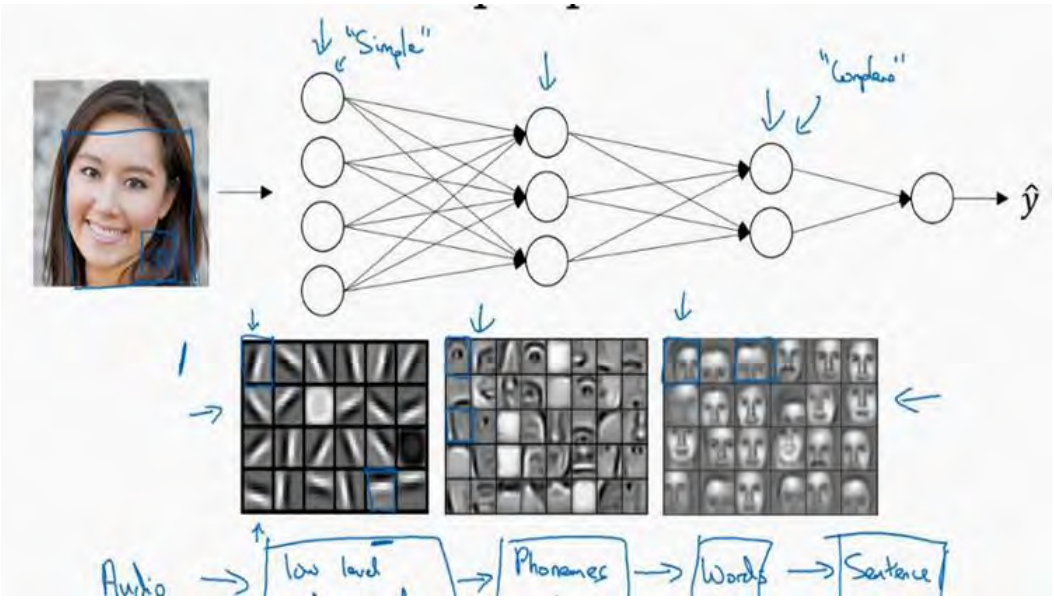


在你做深度神经网络的反向传播时，一定要确认所有的矩阵维数是前后一致的，可以大大提高代码通过率。下一节我们讲为什么深层的网络在很多问题上比浅层的好。

4.5 为什么使用深层表示？（Why deep representations?）

我们都知道深度神经网络能解决好多问题，其实并不需要很大的神经网络，但是得有深度，得有比较多的隐藏层，这是为什么呢？我们一起来看几个例子来帮助理解，为什么深度神经网络会很好用。

首先，深度网络在计算什么？

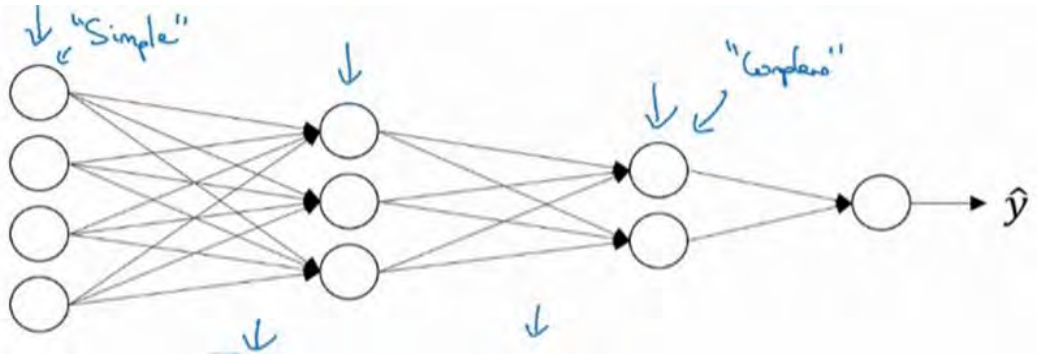


首先，深度网络究竟在计算什么？如果你在建一个人脸识别或是人脸检测系统，深度神经网络所做的事就是，当你输入一张脸部的照片，然后你可以把深度神经网络的第一层，当成一个特征探测器或者边缘探测器。在这个例子里，我会建一个大概有 20 个隐藏单元的深度神经网络，是怎么针对这张图计算的。隐藏单元就是这些图里这些小方块（第一张大图），举个例子，这个小方块（第一行第一列）就是一个隐藏单元，它会去找这张照片里“|”边缘的方向。那么这个隐藏单元（第四行第四列），可能是在找（“—”）水平向的边缘在哪里。之后的课程里，我们会讲专门做这种识别的卷积神经网络，到时候会细讲，为什么小单元是这么表示的。你可以先把神经网络的第一层当作看图，然后去找这张照片的各个边缘。我们可以把照片里组成边缘的像素们放在一起看，然后它可以把被探测到的边缘组合成面部的不同部分（第二张大图）。比如说，可能有一个神经元会去找眼睛的部分，另外还有别的在找鼻子的部分，然后把这许多的边缘结合在一起，就可以开始检测人脸的不同部分。最后再把这些部分放在一起，比如鼻子眼睛下巴，就可以识别或是探测不同的人脸（第三张大图）。

你可以直觉上把这种神经网络的前几层当作探测简单的函数，比如边缘，之后把它们跟后几层结合在一起，那么总体上就能学习更多复杂的函数。这些图的意义，我们在学习卷积

神经网络的时候再深入了解。还有一个技术性的细节需要理解的是，边缘探测器其实相对来说都是针对照片中非常小块的面积。就像这块（第一行第一列），都是很小的区域。面部探测器就会针对于大一些的区域，但是主要的概念是，一般你会从比较小的细节入手，比如边缘，然后再一步步到更大更复杂的区域，比如一只眼睛或是一个鼻子，再把眼睛鼻子装一块组成更复杂的部分。

这种从简单到复杂的金字塔状表示方法或者组成方法，也可以应用在图像或者人脸识别以外的其他数据上。比如当你想要建一个语音识别系统的时候，需要解决的就是如何可视化语音，比如你输入一个音频片段，那么神经网络的第一层可能就会去先开始试着探测比较低层次的音频波形的一些特征，比如音调是变高了还是低了，分辨白噪音，咝咝咝的声音，或者音调，可以选择这些相对程度比较低的波形特征，然后把这些波形组合在一起就能去探测声音的基本单元。在语言学中有个概念叫做音位，比如说单词 ca，c 的发音，“嗑”就是一个音位，a 的发音“啊”是个音位，t 的发音“特”也是个音位，有了基本的声音单元以后，组合起来，你就能识别音频当中的单词，单词再组合起来就能识别词组，再到完整的句子。



所以深度神经网络的这许多隐藏层中，较早的前几层能学习一些低层次的简单特征，等到后几层，就能把简单的特征结合起来，去探测更加复杂的东西。比如你录在音频里的单词、词组或是句子，然后就能运行语音识别了。同时我们所计算的之前的几层，也就是相对简单的输入函数，比如图像单元的边缘什么的。到网络中的深层时，你实际上就能做很多复杂的事，比如探测面部或是探测单词、短语或是句子。

有些人喜欢把深度神经网络和人类大脑做类比，这些神经科学家觉得人的大脑也是先探测简单的东西，比如你眼睛看得到的边缘，然后组合起来才能探测复杂的物体，比如脸。这种深度学习和人类大脑的比较，有时候比较危险。但是不可否认的是，我们对大脑运作机制的认识很有价值，有可能大脑就是先从简单的东西，比如边缘着手，再组合成一个完整的复

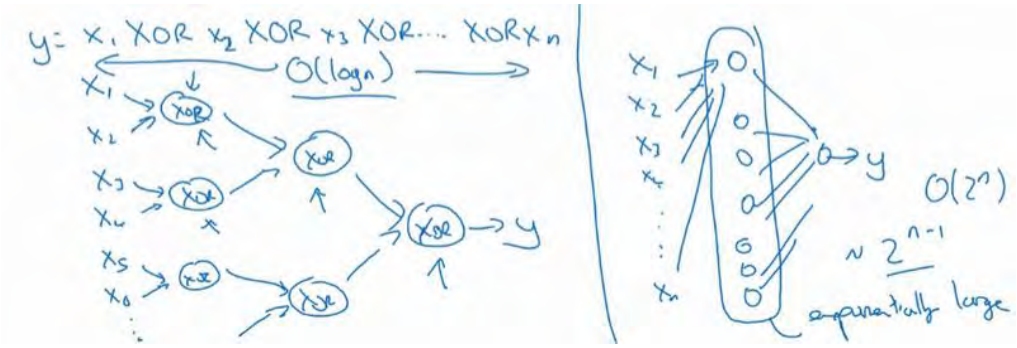
杂物体，这类简单到复杂的过程，同样也是其他一些深度学习的灵感来源，之后的视频我们也会继续聊聊人类或是生物学理解的大脑。

Small：隐藏单元的数量相对较少

Deep：隐藏层数目比较多

深层的网络隐藏单元数量相对较少，隐藏层数目较多，如果浅层的网络想要达到同样的计算结果则需要指数级增长的单元数量才能达到。

另外一个，关于神经网络为何有效的理论，来源于电路理论，它和你能够用电路元件计算哪些函数有着分不开的联系。根据不同的基本逻辑门，譬如与门、或门、非门。在非正式的情况下，这些函数都可以用相对较小，但很深的神经网络来计算，小在这里的意思是隐藏单元的数量相对比较小，但是如果你用浅一些的神经网络计算同样的函数，也就是说在我们不能用很多隐藏层时，你会需要成指数增长的单元数量才能达到同样的计算结果。



我再来举个例子，用没那么正式的语言介绍这个概念。假设你想要对输入特征计算异或或是奇偶性，你可以算$x_1XORx_2XORx_3XOR\ \ldots\ldots x_n$，假设你有$n$或者$n_x$个特征，如果你画一个异或的树图，先要计算$x_1$，$x_2$的异或，然后是$x_3$和$x_4$。技术上来说如果你只用或门，还有非门的话，你可能会需要几层才能计算异或函数，但是用相对小的电路，你应该就可以计算异或了。然后你可以继续建这样的一个异或树图（上图左），那么你最后会得到这样的电路来输出结果y，$\hat{y}=y$，也就是输入特征的异或，或是奇偶性，要计算异或关系。这种树图对应网络的深度应该是$O(log(n))$，那么节点的数量和电路部件，或是门的数量并不会很大，你也不需要太多门去计算异或。

但是如果你不能使用多隐层的神经网络的话，在这个例子中隐层数为$O(log(n))$，比如你被迫只能用单隐藏层来计算的话，这里全部都指向从这些隐藏单元到后面这里，再输出y，那么要计算奇偶性，或者异或关系函数就需要这一隐层（上图右方框部分）的单元数呈指数增长才行，因为本质上来说你需要列举耗尽2^n种可能的配置，或是2^n种输入比特的配置。异或运算的最终结果是 1 或 0，那么你最终就会需要一个隐藏层，其中单元数目随输入比特指

数上升。精确的说应该是2^{n-1}个隐藏单元数，也就是$O(2^n)$。

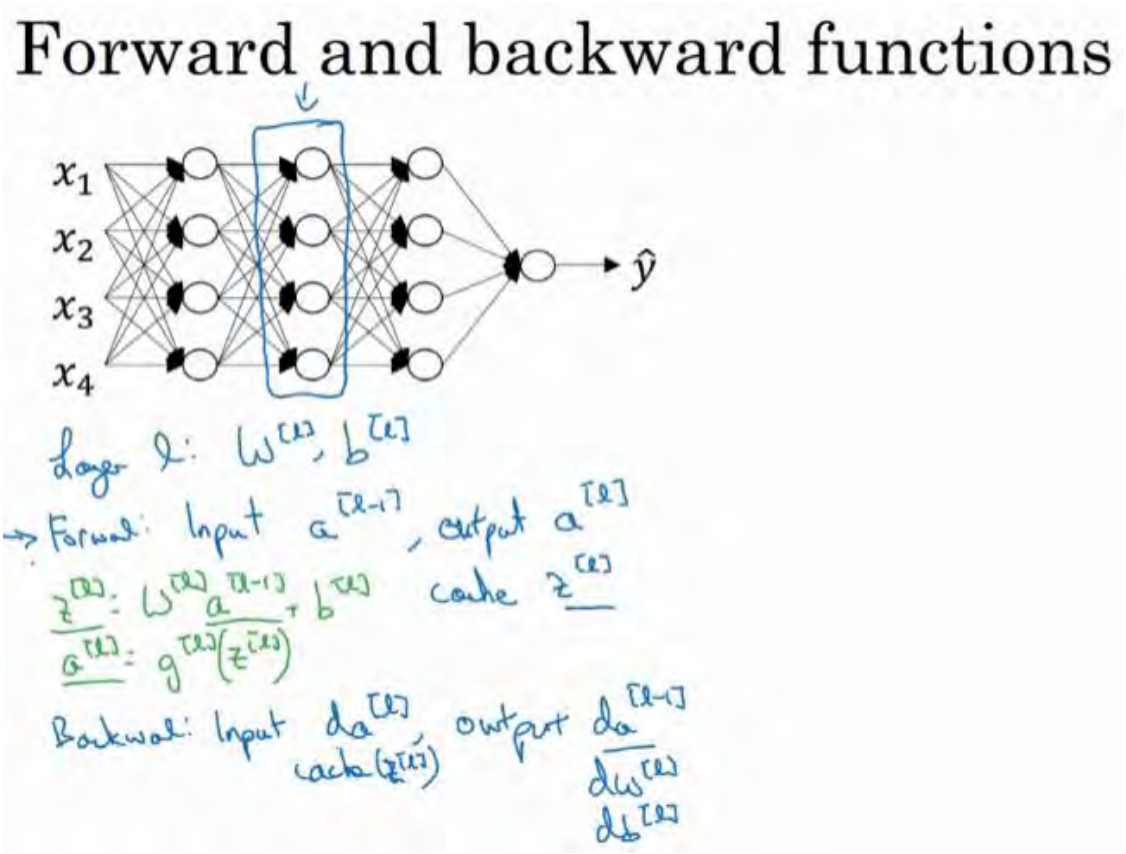
我希望这能让你有点概念，意识到有很多数学函数用深度网络计算比浅网络要容易得多，我个人倒是认为这种电路理论，对训练直觉思维没那么有用，但这个结果人们还是经常提到的，用来解释为什么需要更深层的网络。

除了这些原因，说实话，我认为“深度学习”这个名字挺唬人的，这些概念以前都统称为有很多隐藏层的神经网络，但是深度学习听起来多高大上，太深奥了，对么？这个词流传出去以后，这是神经网络的重新包装或是多隐藏层神经网络的重新包装，激发了大众的想象力。抛开这些公关概念重新包装不谈，深度网络确实效果不错，有时候人们还是会按照字面意思钻牛角尖，非要用很多隐层。但是当我开始解决一个新问题时，我通常会从 **logistic** 回归开始，再试试一到两个隐层，把隐藏层数量当作参数、超参数一样去调试，这样去找比较合适的深度。但是近几年以来，有一些人会趋向于使用非常非常深邃的神经网络，比如好几打的层数，某些问题中只有这种网络才是最佳模型。

这就是我想讲的，为什么深度学习效果拔群的直觉解释，现在我们来看看除了正向传播以外，反向传播该怎么具体实现。

4.6 搭建神经网络块（Building blocks of deep neural networks）

这周的前几个视频和之前几周的视频里，你已经看到过正向反向传播的基础组成部分了，它们也是深度神经网络的重要组成部分，现在我们来用它们建一个深度神经网络。

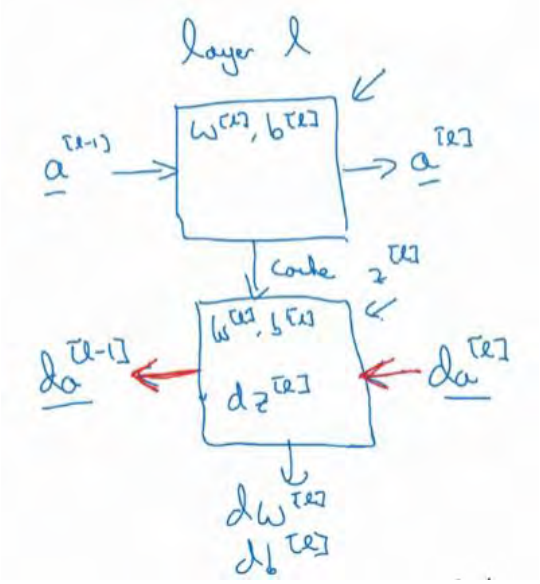


这是一个层数较少的神经网络，我们选择其中一层（方框部分），从这一层的计算着手。在第l层你有参数$W^{[l]}$和$b^{[l]}$，正向传播里有输入的激活函数，输入是前一层$a^{[l-1]}$，输出是$a^{[l]}$，我们之前讲过$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$,$a^{[l]} = g^{[l]}(z^{[l]})$，那么这就是你如何从输入$a^{[l-1]}$走到输出的$a^{[l]}$。之后你就可以把$z^{[l]}$的值缓存起来，我在这里也会把这包括在缓存中，因为缓存的$z^{[l]}$对以后的正向反向传播的步骤非常有用。

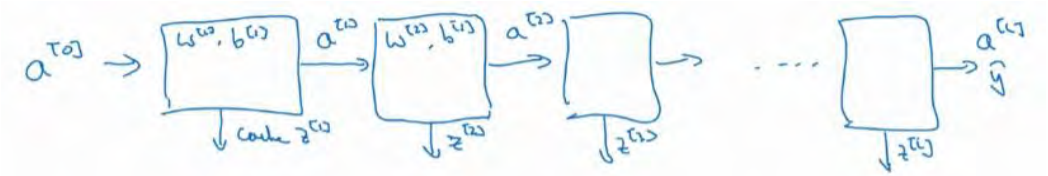
然后是反向步骤或者说反向传播步骤，同样也是第l层的计算，你会需要实现一个函数输入为$da^{[l]}$，输出$da^{[l-1]}$的函数。一个小细节需要注意，输入在这里其实是$da^{[l]}$以及所缓存的$z^{[l]}$值，之前计算好的$z^{[l]}$值，除了输出$da^{[l-1]}$的值以外，也需要输出你需要的梯度$dW^{[l]}$和$db^{[l]}$，这是为了实现梯度下降学习。

这就是基本的正向步骤的结构，我把它成为称为正向函数，类似的在反向步骤中会称为反向函数。总结起来就是，在 l 层，你会有正向函数，输入$a^{[l-1]}$并且输出$a^{[l]}$，为了计算结果你需要用$W^{[l]}$和$b^{[l]}$，以及输出到缓存的$z^{[l]}$。然后用作反向传播的反向函数，是另一个函

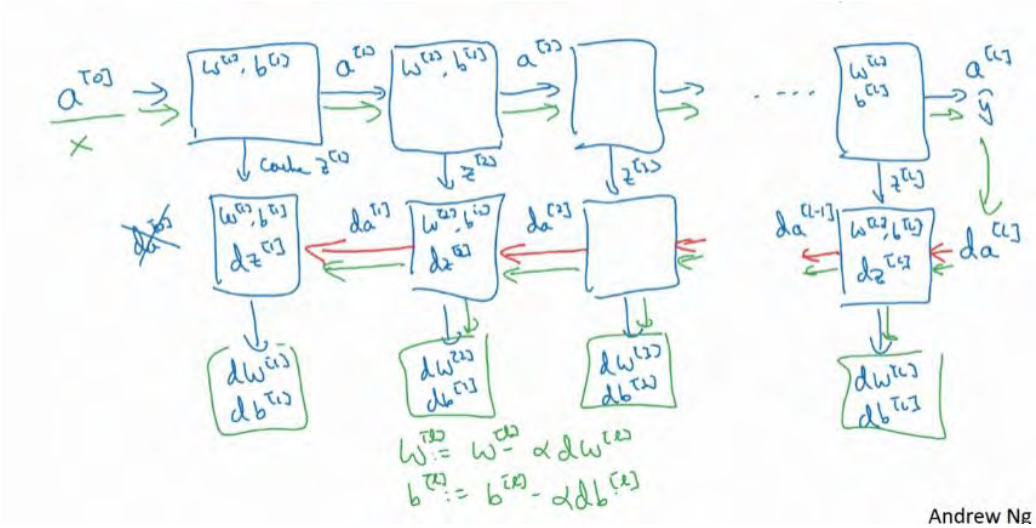
数，输入$da^{[l]}$，输出$da^{[l-1]}$，你就会得到对激活函数的导数，也就是希望的导数值$da^{[l]}$。$a^{[l-1]}$是会变的，前一层算出的激活函数导数。在这个方块（第二个）里你需要$W^{[l]}$和$b^{[l]}$，最后你要算的是$dz^{[l]}$。然后这个方块（第三个）中，这个反向函数可以计算输出$dW^{[l]}$和$db^{[l]}$。我会用红色箭头标注标注反向步骤，如果你们喜欢，我可以把这些箭头涂成红色。



然后如果实现了这两个函数（正向和反向），然后神经网络的计算过程会是这样的：



把输入特征$a^{[0]}$，放入第一层并计算第一层的激活函数，用$a^{[1]}$表示，你需要$W^{[1]}$和$b^{[1]}$来计算，之后也缓存$z^{[l]}$值。之后喂到第二层，第二层里，需要用到$W^{[2]}$和$b^{[2]}$，你会需要计算第二层的激活函数$a^{[2]}$。后面几层以此类推，直到最后你算出了$a^{[L]}$，第L层的最终输出值$\hat{y}$。在这些过程里我们缓存了所有的z值，这就是正向传播的步骤。



对反向传播的步骤而言，我们需要算一系列的反向迭代，就是这样反向计算梯度，你需要把$da^{[L]}$的值放在这里，然后这个方块会给我们$da^{[L-1]}$的值，以此类推，直到我们得到$da^{[2]}$和$da^{[1]}$，你还可以计算多一个输出值，就是$da^{[0]}$，但这其实是你的输入特征的导数，并不重要，起码对于训练监督学习的权重不算重要，你可以止步于此。反向传播步骤中也会输出$dW^{[l]}$和$db^{[l]}$，这会输出$dW^{[3]}$和$db^{[3]}$等等。目前为止你算好了所有需要的导数，稍微填一下这个流程图。

神经网络的一步训练包含了，从$a^{[0]}$开始，也就是 x 然后经过一系列正向传播计算得到$\hat{y}$，之后再用输出值计算这个（第二行最后方块），再实现反向传播。现在你就有所有的导数项了，W也会在每一层被更新为$W = W - \alpha dW$，b也一样，$b = b - \alpha db$，反向传播就都计算完毕，我们有所有的导数值，那么这是神经网络一个梯度下降循环。

继续下去之前再补充一个细节，概念上会非常有帮助，那就是把反向函数计算出来的z值缓存下来。当你做编程练习的时候去实现它时，你会发现缓存可能很方便，可以迅速得到$W^{[l]}$和$b^{[l]}$的值，非常方便的一个方法，在编程练习中你缓存了z，还有W和b对吧？从实现角度上看，我认为是一个很方便的方法，可以将参数复制到你在计算反向传播时所需要的地方。好，这就是实现过程的细节，做编程练习时会用到。

现在你们见过实现深度神经网络的基本元件，在每一层中有一个正向传播步骤，以及对应的反向传播步骤，以及把信息从一步传递到另一步的缓存。下一个视频我们会讲解这些元件具体实现过程，我们来看下一个视频吧。

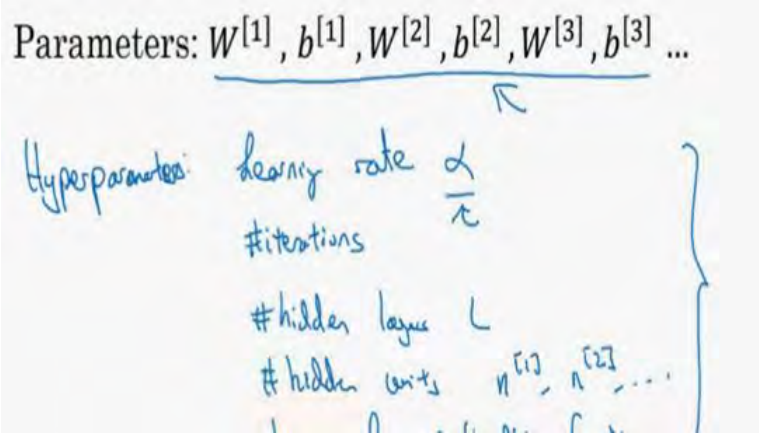
4.7 参数 VS 超参数（Parameters vs Hyperparameters）

想要你的深度神经网络起很好的效果，你还需要规划好你的参数以及超参数。

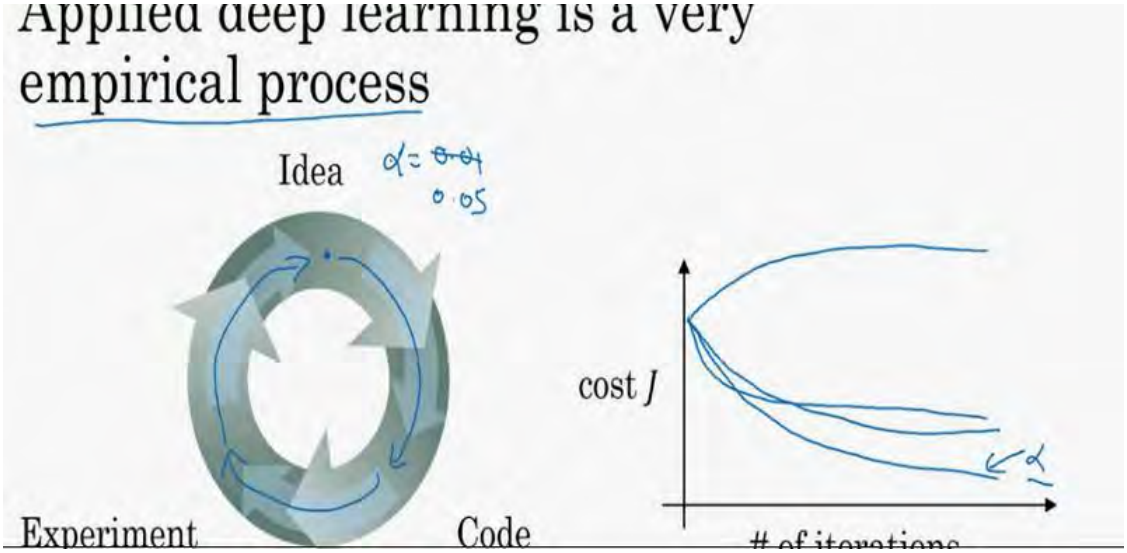
什么是超参数？

比如算法中的 **learning rate** a（学习率）、**iterations**(梯度下降法循环的数量)、L（隐藏层数目）、$n^{[l]}$（隐藏层单元数目）、**choice of activation function**（激活函数的选择）都需要你来设置，这些数字实际上控制了最后的参数W和b的值，所以它们被称作超参数。

实际上深度学习有很多不同的超参数，之后我们也会介绍一些其他的超参数，如 **momentum**、**mini batch size**、**regularization parameters** 等等。



如何寻找超参数的最优值？



走 **Idea—Code—Experiment—Idea** 这个循环，尝试各种不同的参数，实现模型并观察是否成功，然后再迭代。

今天的深度学习应用领域，还是很经验性的过程，通常你有个想法，比如你可能大致知

道一个最好的学习率值，可能说$a = 0.01$最好，我会想先试试看，然后你可以实际试一下，训练一下看看效果如何。然后基于尝试的结果你会发现，你觉得学习率设定再提高到 0.05 会比较好。如果你不确定什么值是最好的，你大可以先试试一个学习率a，再看看损失函数 J 的值有没有下降。然后你可以试一试大一些的值，然后发现损失函数的值增加并发散了。然后可能试试其他数，看结果是否下降的很快或者收敛到在更高的位置。你可能尝试不同的a并观察损失函数J这么变了，试试一组值，然后可能损失函数变成这样，这个a值会加快学习过程，并且收敛在更低的损失函数值上（箭头标识），我就用这个a值了。

在前面几页中，还有很多不同的超参数。然而，当你开始开发新应用时，预先很难确切知道，究竟超参数的最优值应该是什么。所以通常，你必须尝试很多不同的值，并走这个循环，试试各种参数。试试看 5 个隐藏层，这个数目的隐藏单元，实现模型并观察是否成功，然后再迭代。这页的标题是，应用深度学习领域，一个很大程度基于经验的过程，凭经验的过程通俗来说，就是试直到你找到合适的数值。

另一个近来深度学习的影响是它用于解决很多问题，从计算机视觉到语音识别，到自然语言处理，到很多结构化的数据应用，比如网络广告或是网页搜索或产品推荐等等。我所看到过的就有很多其中一个领域的研究员，这些领域中的一个，尝试了不同的设置，有时候这种设置超参数的直觉可以推广，但有时又不会。所以我经常建议人们，特别是刚开始应用于新问题的人们，去试一定范围的值看看结果如何。然后下一门课程，我们会用更系统的方法，用系统性的尝试各种超参数取值。然后其次，甚至是你已经用了很久的模型，可能你在做网络广告应用，在你开发途中，很有可能学习率的最优数值或是其他超参数的最优值是会变的，所以即使你每天都在用当前最优的参数调试你的系统，你还是会发现，最优值过一年就会变化，因为电脑的基础设施，**CPU** 或是 **GPU** 可能会变化很大。所以有一条经验规律可能每几个月就会变。如果你所解决的问题需要很多年时间，只要经常试试不同的超参数，勤于检验结果，看看有没有更好的超参数数值，相信你慢慢会得到设定超参数的直觉，知道你的问题最好用什么数值。

这可能的确是深度学习比较让人不满的一部分，也就是你必须尝试很多次不同可能性。但参数设定这个领域，深度学习研究还在进步中，所以可能过段时间就会有更好的方法决定超参数的值，也很有可能由于 **CPU**、**GPU**、网络和数据都在变化，这样的指南可能只会在一段时间内起作用，只要你不断尝试，并且尝试保留交叉检验或类似的检验方法，然后挑一个对你的问题效果比较好的数值。

近来受深度学习影响，很多领域发生了变化，从计算机视觉到语音识别到自然语言处理

到很多结构化的数据应用，比如网络广告、网页搜索、产品推荐等等；有些同一领域设置超参数的直觉可以推广，但有时又不可以，特别是那些刚开始研究新问题的人们应该去尝试一定范围内的结果如何，甚至那些用了很久的模型得学习率或是其他超参数的最优值也有可能会改变。

在下个课程我们会用系统性的方法去尝试各种超参数的取值。有一条经验规律：经常试试不同的超参数，勤于检查结果，看看有没有更好的超参数取值，你将会得到设定超参数的直觉。

4.8 深度学习和大脑的关联性（What does this have to do with the brain?）

深度学习和大脑有什么关联性吗？

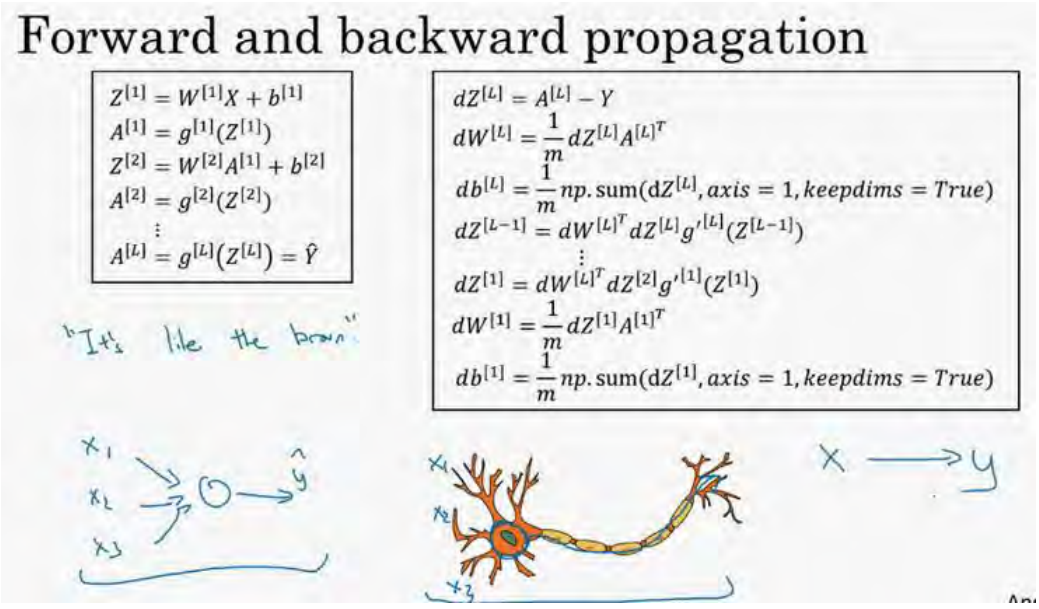
关联不大。

那么人们为什么会说深度学习和大脑相关呢？

当你在实现一个神经网络的时候，那些公式是你在做的东西，你会做前向传播、反向传播、梯度下降法，其实很难表述这些公式具体做了什么，深度学习像大脑这样的类比其实是过度简化了我们的大脑具体在做什么，但因为这种形式很简洁，也能让普通人更愿意公开讨论，也方便新闻报道并且吸引大众眼球，但这个类比是非常不准确的。

一个神经网络的逻辑单元可以看成是对一个生物神经元的过度简化，但迄今为止连神经科学家都很难解释究竟一个神经元能做什么，它可能是极其复杂的；它的一些功能可能真的类似 **logistic** 回归的运算，但单个神经元到底在做什么目前还没有人能够真正可以解释。

深度学习的确是个很好的工具来学习各种很灵活很复杂的函数，学习到从x到y的映射，在监督学习中学到输入到输出的映射。



但这个类比还是很粗略的，这是一个 **logistic** 回归单元的 **sigmoid** 激活函数，这里是一个大脑中的神经元，图中这个生物神经元，也是你大脑中的一个细胞，它能接受来自其他神经元的电信号，比如x_1, x_2, x_3，或可能来自于其他神经元a_1, a_2, a_3 。其中有一个简单的临界计算值，如果这个神经元突然激发了，它会让电脉冲沿着这条长长的轴突，或者说一条导线

传到另一个神经元。

所以这是一个过度简化的对比，把一个神经网络的逻辑单元和右边的生物神经元对比。至今为止其实连神经科学家们都很难解释，究竟一个神经元能做什么。一个小小的神经元其实却是极其复杂的，以至于我们无法在神经科学的角度描述清楚，它的一些功能，可能真的是类似 **logistic** 回归的运算，但单个神经元到底在做什么，目前还没有人能够真正解释，大脑中的神经元是怎么学习的，至今这仍是一个谜之过程。到底大脑是用类似于后向传播或是梯度下降的算法，或者人类大脑的学习过程用的是完全不同的原理。

所以虽然深度学习的确是个很好的工具，能学习到各种很灵活很复杂的函数来学到从 x 到 y 的映射。在监督学习中，学到输入到输出的映射，但这种和人类大脑的类比，在这个领域的早期也许值得一提。但现在这种类比已经逐渐过时了，我自己也在尽量少用这样的说法。

这就是神经网络和大脑的关系，我相信在计算机视觉，或其他的学科都曾受人类大脑启发，还有其他深度学习的领域也曾受人类大脑启发。但是个人来讲我用这个人类大脑类比的次数逐渐减少了。

第二门课 改善深层神经网络：超参数调试、正则化以及优化(Improving Deep Neural Networks:Hyperparameter tuning, Regularization and Optimization)

第一周：深度学习的实用层面(Practical aspects of Deep Learning)

1.1 训练，验证，测试集（Train / Dev / Test sets）

大家可能已经了解了，那么本周，我们将继续学习如何有效运作神经网络，内容涉及超参数调优，如何构建数据，以及如何确保优化算法快速运行，从而使学习算法在合理时间内完成自我学习。

第一周，我们首先说说神经网络机器学习中的问题，然后是随机神经网络，还会学习一些确保神经网络正确运行的技巧，带着这些问题，我们开始今天的课程。

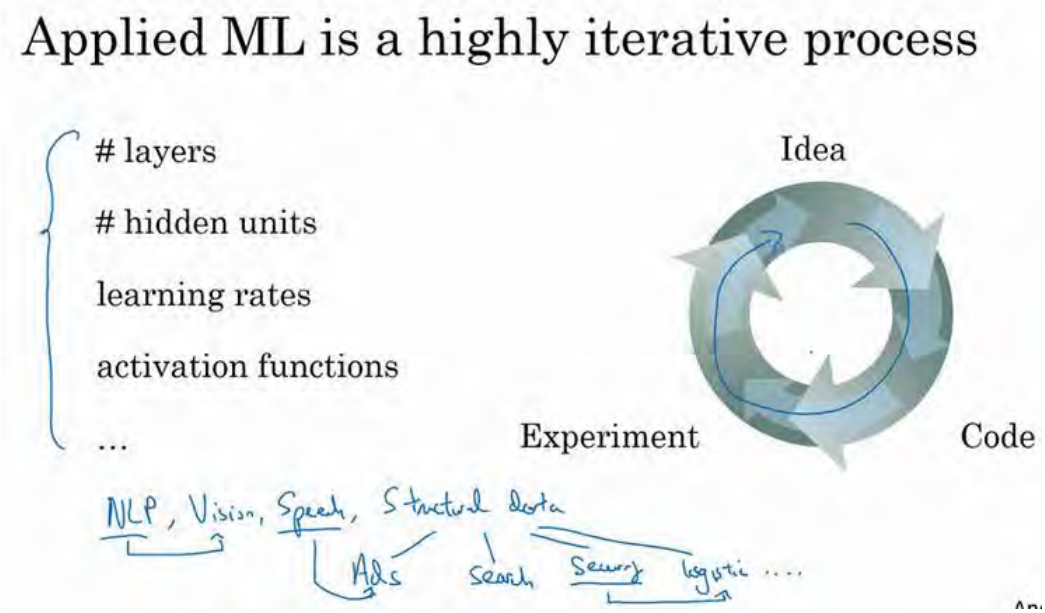
在配置训练、验证和测试数据集的过程中做出正确决策会在很大程度上帮助大家创建高效的神经网络。训练神经网络时，我们需要做出很多决策，例如：

神经网络分多少层

每层含有多少个隐藏单元

学习速率是多少

各层采用哪些激活函数



创建新应用的过程中，我们不可能从一开始就准确预测出这些信息和其他超级参数。实际上，应用型机器学习是一个高度迭代的过程，通常在项目启动时，我们会先有一个初步想法，比如构建一个含有特定层数，隐藏单元数量或数据集个数等等的神经网络，然后编码，并尝试运行这些代码，通过运行和测试得到该神经网络或这些配置信息的运行结果，你可能会根据输出结果重新完善自己的想法，改变策略，或者为了找到更好的神经网络不断迭代更新自己的方案。

现如今，深度学习已经在自然语言处理，计算机视觉，语音识别以及结构化数据应用等众多领域取得巨大成功。结构化数据无所不包，从广告到网络搜索。其中网络搜索不仅包括网络搜索引擎，还包括购物网站，从所有根据搜索栏词条传输结果的网站。再到计算机安全，物流，比如判断司机去哪接送货，范围之广，不胜枚举。

我发现，可能有自然语言处理方面的人才想踏足计算机视觉领域，或者经验丰富的语音识别专家想投身广告行业，又或者，有的人想从电脑安全领域跳到物流行业，在我看来，从一个领域或者应用领域得来的直觉经验，通常无法转移到其他应用领域，最佳决策取决于你所拥有的数据量，计算机配置中输入特征的数量，用 **GPU** 训练还是 **CPU**，**GPU** 和 **CPU** 的具体配置以及其他诸多因素。

目前为止，我觉得，对于很多应用系统，即使是经验丰富的深度学习行家也不太可能一开始就预设出最匹配的超级参数，所以说，应用深度学习是一个典型的迭代过程，需要多次循环往复，才能为应用程序找到一个称心的神经网络，因此循环该过程的效率是决定项目进展速度的一个关键因素，而创建高质量的训练数据集，验证集和测试集也有助于提高循环效

率。

假设这是训练数据，我用一个长方形表示，我们通常会将这些数据划分成几部分，一部分作为训练集，一部分作为简单交叉验证集，有时也称之为验证集，方便起见，我就叫它验证集（**dev set**），其实都是同一个概念，最后一部分则作为测试集。

接下来，我们开始对训练执行算法，通过验证集或简单交叉验证集选择最好的模型，经过充分验证，我们选定了最终模型，然后就可以在测试集上进行评估了，为了无偏评估算法的运行状况。

在机器学习发展的小数据量时代，常见做法是将所有数据三七分，就是人们常说的 70%验证集，30%测试集，如果没有明确设置验证集，也可以按照 60%训练，20%验证和 20%测试集来划分。这是前几年机器学习领域普遍认可的最好的实践方法。

如果只有 100 条，1000 条或者 1 万条数据，那么上述比例划分是非常合理的。

但是在大数据时代，我们现在的数据量可能是百万级别，那么验证集和测试集占数据总量的比例会趋向于变得更小。因为验证集的目的就是验证不同的算法，检验哪种算法更有效，因此，验证集要足够大才能评估，比如 2 个甚至 10 个不同算法，并迅速判断出哪种算法更有效。我们可能不需要拿出 20%的数据作为验证集。

比如我们有 100 万条数据，那么取 1 万条数据便足以进行评估，找出其中表现最好的 1-2 种算法。同样地，根据最终选择的分类器，测试集的主要目的是正确评估分类器的性能，所以，如果拥有百万数据，我们只需要 1000 条数据，便足以评估单个分类器，并且准确评估该分类器的性能。假设我们有 100 万条数据，其中 1 万条作为验证集，1 万条作为测试集，100 万里取 1 万，比例是 1%，即：训练集占 98%，验证集和测试集各占 1%。对于数据量过百万的应用，训练集可以占到 99.5%，验证和测试集各占 0.25%，或者验证集占 0.4%，测试集占 0.1%。

总结一下，在机器学习中，我们通常将样本分成训练集，验证集和测试集三部分，数据集规模相对较小，适用传统的划分比例，数据集规模较大的，验证集和测试集要小于数据总量的 20%或 10%。后面我会给出如何划分验证集和测试集的具体指导。

现代深度学习的另一个趋势是越来越多的人在训练和测试集分布不匹配的情况下进行训练，假设你要构建一个用户可以上传大量图片的应用程序，目的是找出并呈现所有猫咪图片，可能你的用户都是爱猫人士，训练集可能是从网上下载的猫咪图片，而验证集和测试集是用户在这个应用上上传的猫的图片，就是说，训练集可能是从网络上抓下来的图片。而验

证集和测试集是用户上传的图片。结果许多网页上的猫咪图片分辨率很高，很专业，后期制作精良，而用户上传的照片可能是用手机随意拍摄的，像素低，比较模糊，这两类数据有所不同，针对这种情况，根据经验，我建议大家要确保验证集和测试集的数据来自同一分布，关于这个问题我也会多讲一些。因为你们要用验证集来评估不同的模型，尽可能地优化性能。如果验证集和测试集来自同一个分布就会很好。

但由于深度学习算法需要大量的训练数据，为了获取更大规模的训练数据集，我们可以采用当前流行的各种创意策略，例如，网页抓取，代价就是训练集数据与验证集和测试集数据有可能不是来自同一分布。但只要遵循这个经验法则，你就会发现机器学习算法会变得更快。我会在后面的课程中更加详细地解释这条经验法则。

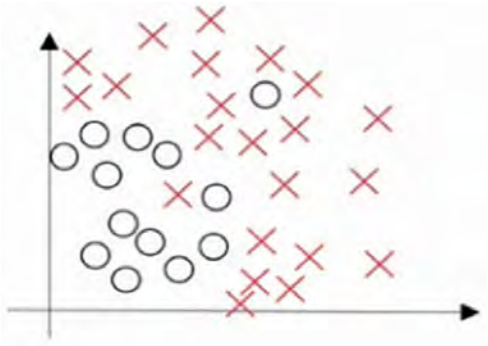
最后一点，就算没有测试集也不要紧，测试集的目的是对最终所选定的神经网络系统做出无偏估计，如果不需要无偏估计，也可以不设置测试集。所以如果只有验证集，没有测试集，我们要做的就是，在训练集上训练，尝试不同的模型框架，在验证集上评估这些模型，然后迭代并选出适用的模型。因为验证集中已经涵盖测试集数据，其不再提供无偏性能评估。当然，如果你不需要无偏估计，那就再好不过了。

在机器学习中，如果只有一个训练集和一个验证集，而没有独立的测试集，遇到这种情况，训练集还被人们称为训练集，而验证集则被称为测试集，不过在实际应用中，人们只是把测试集当成简单交叉验证集使用，并没有完全实现该术语的功能，因为他们把验证集数据过度拟合到了测试集中。如果某团队跟你说他们只设置了一个训练集和一个测试集，我会很谨慎，心想他们是不是真的有训练验证集，因为他们把验证集数据过度拟合到了测试集中，让这些团队改变叫法，改称其为“训练验证集”，而不是“训练测试集”，可能不太容易。即便我认为“训练验证集“在专业用词上更准确。实际上，如果你不需要无偏评估算法性能，那么这样是可以的。

所以说，搭建训练验证集和测试集能够加速神经网络的集成，也可以更有效地衡量算法地偏差和方差，从而帮助我们更高效地选择合适方法来优化算法。

1.2 偏差，方差（Bias /Variance）

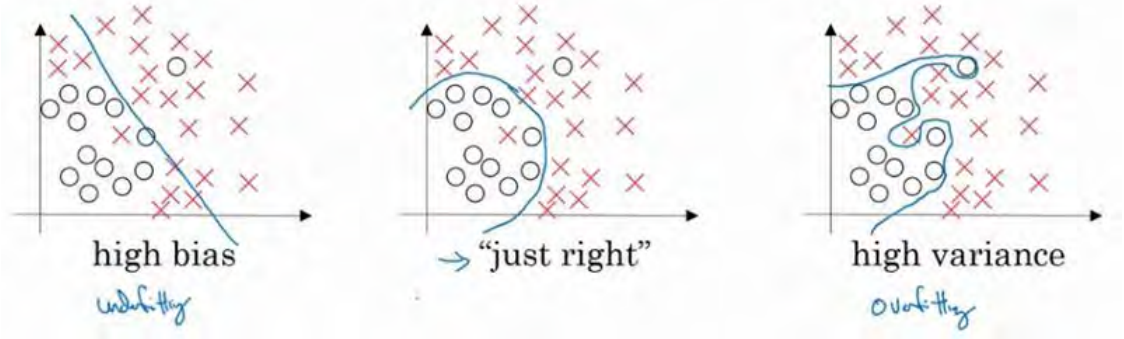
我注意到，几乎所有机器学习从业人员都期望深刻理解偏差和方差，这两个概念易学难精，即使你自己认为已经理解了偏差和方差的基本概念，却总有一些意想不到的新东西出现。关于深度学习的误差问题，另一个趋势是对偏差和方差的权衡研究甚浅，你可能听说过这两个概念，但深度学习的误差很少权衡二者，我们总是分别考虑偏差和方差，却很少谈及偏差和方差的权衡问题，下面我们来一探究竟。



假设这就是数据集，如果给这个数据集拟合一条直线，可能得到一个逻辑回归拟合，但它并不能很好地拟合该数据，这是高偏差（**high bias**）的情况，我们称为“欠拟合”（**underfitting**）。

相反的如果我们拟合一个非常复杂的分类器，比如深度神经网络或含有隐藏单元的神经网络，可能就非常适用于这个数据集，但是这看起来也不是一种很好的拟合方式分类器方差较高（**high variance**），数据过度拟合（**overfitting**）。

在两者之间，可能还有一些像图中这样的，复杂程度适中，数据拟合适度的分类器，这个数据拟合看起来更加合理，我们称之为“适度拟合”（**just right**）是介于过度拟合和欠拟合中间的一类。



在这样一个只有x_1和x_2两个特征的二维数据集中，我们可以绘制数据，将偏差和方差可视化。在多维空间数据中，绘制数据和可视化分割边界无法实现，但我们可以通过几个指标，

来研究偏差和方差。

我们沿用猫咪图片分类这个例子，左边一张是猫咪图片，右边一张不是。理解偏差和方差的两个关键数据是训练集误差（**Train set error**）和验证集误差（**Dev set error**），为了方便论证，假设我们可以辨别图片中的小猫，我们用肉眼识别几乎是不会出错的。

假定训练集误差是 1%，为了方便论证，假定验证集误差是 11%，可以看出训练集设置得非常好，而验证集设置相对较差，我们可能过度拟合了训练集，在某种程度上，验证集并没有充分利用交叉验证集的作用，像这种情况，我们称之为“高方差”。

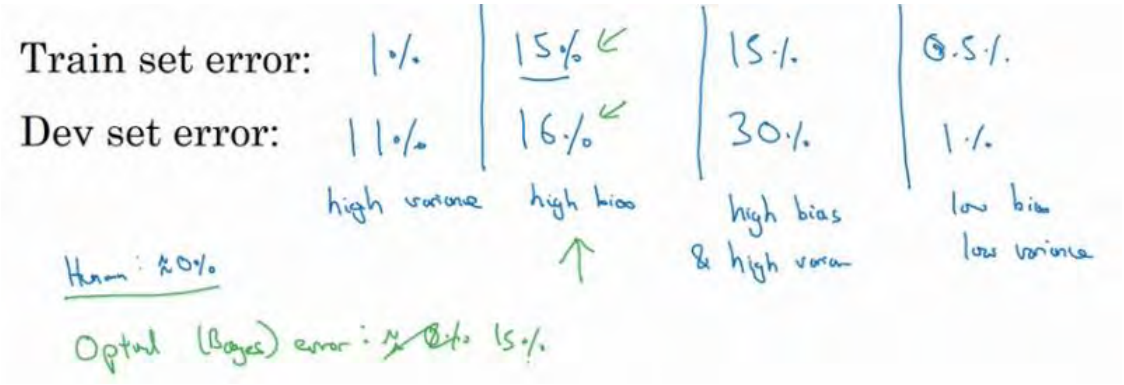
通过查看训练集误差和验证集误差，我们便可以诊断算法是否具有高方差。也就是说衡量训练集和验证集误差就可以得出不同结论。

假设训练集误差是 15%，我们把训练集误差写在首行，验证集误差是 16%，假设该案例中人的错误率几乎为 0%，人们浏览这些图片，分辨出是不是猫。算法并没有在训练集中得到很好训练，如果训练数据的拟合度不高，就是数据欠拟合，就可以说这种算法偏差比较高。相反，它对于验证集产生的结果却是合理的，验证集中的错误率只比训练集的多了 1%，所以这种算法偏差高，因为它甚至不能拟合训练集，这与上一张幻灯片最左边的图片相似。

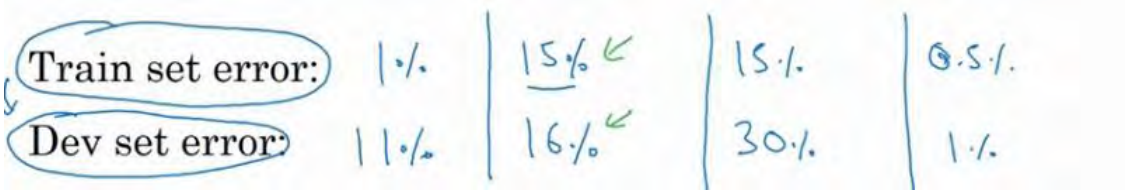
再举一个例子，训练集误差是 15%，偏差相当高，但是，验证集的评估结果更糟糕，错误率达到 30%，在这种情况下，我会认为这种算法偏差高，因为它在训练集上结果不理想，而且方差也很高，这是方差偏差都很糟糕的情况。

再看最后一个例子，训练集误差是 0.5%，验证集误差是 1%，用户看到这样的结果会很开心，猫咪分类器只有 1%的错误率，偏差和方差都很低。

有一点我先在这个简单提一下，具体的留在后面课程里讲，这些分析都是基于假设预测的，假设人眼辨别的错误率接近 0%，一般来说，最优误差也被称为贝叶斯误差，所以，最优误差接近 0%，我就不在这里细讲了，如果最优误差或贝叶斯误差非常高，比如 15%。我们再看看这个分类器（训练误差 15%，验证误差 16%），15%的错误率对训练集来说也是非常合理的，偏差不高，方差也非常低。

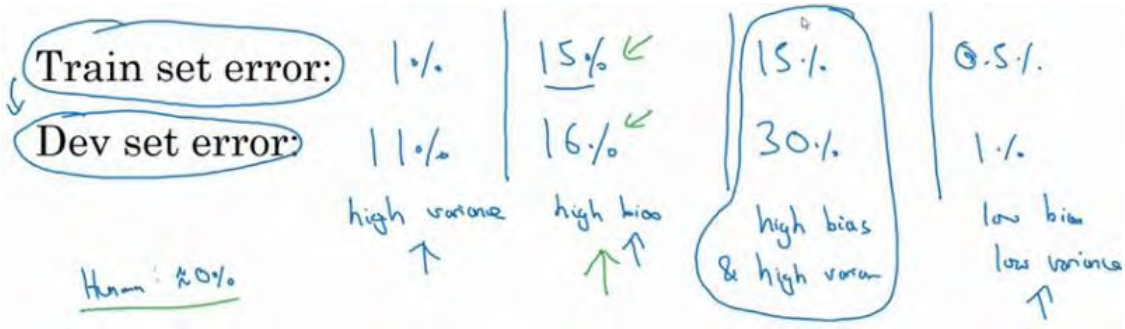


当所有分类器都不适用时，如何分析偏差和方差呢？比如，图片很模糊，即使是人眼，或者没有系统可以准确无误地识别图片，在这种情况下，最优误差会更高，那么分析过程就要做些改变了，我们暂时先不讨论这些细微差别，重点是通过查看训练集误差，我们可以判断数据拟合情况，至少对于训练数据是这样，可以判断是否有偏差问题，然后查看错误率有多高。当完成训练集训练，开始使用验证集验证时，我们可以判断方差是否过高，从训练集到验证集的这个过程中，我们可以判断方差是否过高。

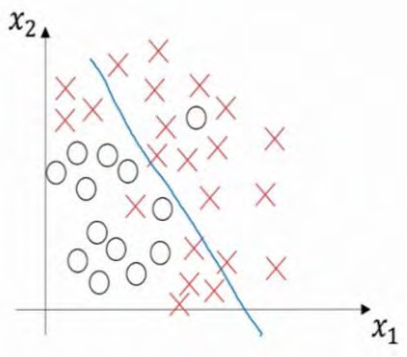


以上分析的前提都是假设基本误差很小，训练集和验证集数据来自相同分布，如果没有这些假设作为前提，分析过程更加复杂，我们将会在稍后课程里讨论。

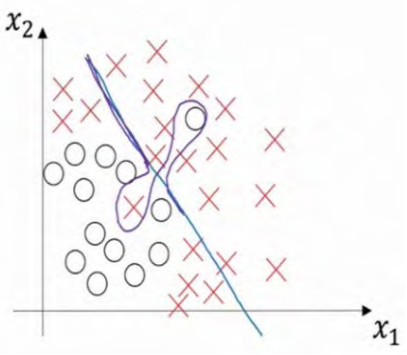
上一张幻灯片，我们讲了高偏差和高方差的情况，大家应该对优质分类器有了一定的认识，偏差和方差都高是什么样子呢？这种情况对于两个衡量标准来说都是非常糟糕的。



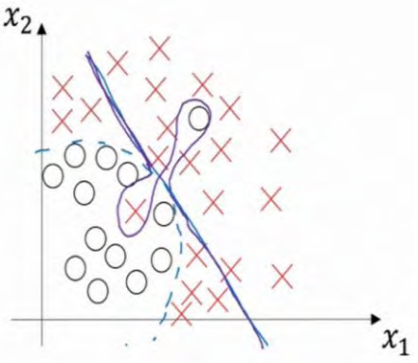
我们之前讲过，这样的分类器，会产生高偏差，因为它的数据拟合度低，像这种接近线性的分类器，数据拟合度低。



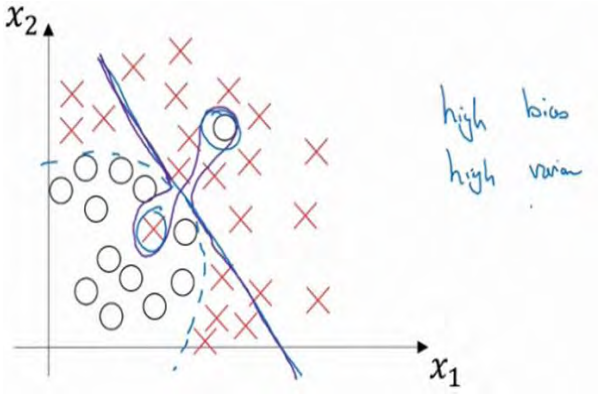
但是如果我们稍微改变一下分类器，我用紫色笔画出，它会过度拟合部分数据，用紫色线画出的分类器具有高偏差和高方差，偏差高是因为它几乎是一条线性分类器，并未拟合数据。



这种二次曲线能够很好地拟合数据。



这条曲线中间部分灵活性非常高，却过度拟合了这两个样本，这类分类器偏差很高，因为它几乎是线性的。



而采用曲线函数或二次元函数会产生高方差，因为它曲线灵活性太高以致拟合了这两个错误样本和中间这些活跃数据。

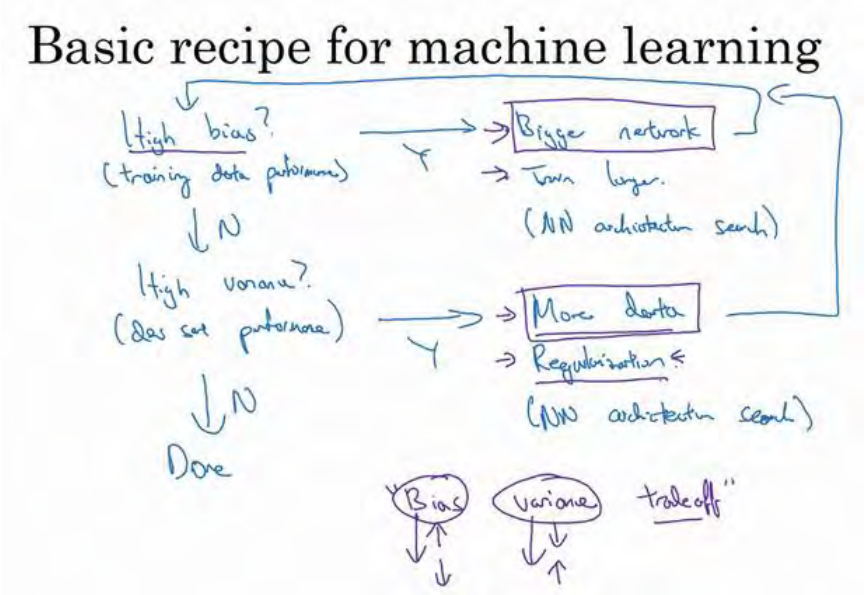
这看起来有些不自然，从两个维度上看都不太自然，但对于高维数据，有些数据区域偏差高，有些数据区域方差高，所以在高维数据中采用这种分类器看起来就不会那么牵强了。

总结一下，我们讲了如何通过分析在训练集上训练算法产生的误差和验证集上验证算法产生的误差来诊断算法是否存在高偏差和高方差，是否两个值都高，或者两个值都不高，根据算法偏差和方差的具体情况决定接下来你要做的工作，下节课，我会根据算法偏差和方差的高低情况讲解一些机器学习的基本方法，帮助大家更系统地优化算法，我们下节课见。

1.3 机器学习基础（Basic Recipe for Machine Learning）

上节课我们讲的是如何通过训练误差和验证集误差判断算法偏差或方差是否偏高，帮助我们更加系统地在机器学习中运用这些方法来优化算法性能。

下图就是我在训练神经网络用到的基本方法：（尝试这些方法，可能有用，可能没用）



这是我在训练神经网络时用到地基本方法，初始模型训练完成后，我首先要知道算法的偏差高不高，如果偏差较高，试着评估训练集或训练数据的性能。如果偏差的确很高，甚至无法拟合训练集，那么你要做的就是选择一个新的网络，比如含有更多隐藏层或者隐藏单元的网络，或者花费更多时间来训练网络，或者尝试更先进的优化算法，后面我们会讲到这部分内容。你也可以尝试其他方法，可能有用，也可能没用。

一会儿我们会看到许多不同的神经网络架构，或许你能找到一个更合适解决此问题的新的网络架构，加上括号，因为其中一条就是你必须去尝试，可能有用，也可能没用，不过采用规模更大的网络通常都会有所帮助，延长训练时间不一定有用，但也没什么坏处。训练学习算法时，我会不断尝试这些方法，直到解决掉偏差问题，这是最低标准，反复尝试，直到可以拟合数据为止，至少能够拟合训练集。

如果网络足够大，通常可以很好的拟合训练集，只要你能扩大网络规模，如果图片很模糊，算法可能无法拟合该图片，但如果有人可以分辨出图片，如果你觉得基本误差不是很高，那么训练一个更大的网络，你就应该可以……至少可以很好地拟合训练集，至少可以拟合或者过拟合训练集。一旦偏差降低到可以接受的数值，检查一下方差有没有问题，为了评估方

差，我们要查看验证集性能，我们能从一个性能理想的训练集推断出验证集的性能是否也理想，如果方差高，最好的解决办法就是采用更多数据，如果你能做到，会有一定的帮助，但有时候，我们无法获得更多数据，我们也可以尝试通过正则化来减少过拟合，这个我们下节课会讲。有时候我们不得不反复尝试，但是，如果能找到更合适的神经网络框架，有时它可能会一箭双雕，同时减少方差和偏差。如何实现呢？想系统地说出做法很难，总之就是不断重复尝试，直到找到一个低偏差，低方差的框架，这时你就成功了。

有两点需要大家注意：

第一点，高偏差和高方差是两种不同的情况，我们后续要尝试的方法也可能完全不同，我通常会用训练验证集来诊断算法是否存在偏差或方差问题，然后根据结果选择尝试部分方法。举个例子，如果算法存在高偏差问题，准备更多训练数据其实也没什么用处，至少这不是更有效的方法，所以大家要清楚存在的问题是偏差还是方差，还是两者都有问题，明确这一点有助于我们选择出最有效的方法。

第二点，在机器学习的初期阶段，关于所谓的偏差方差权衡的讨论屡见不鲜，原因是我们能尝试的方法有很多。可以增加偏差，减少方差，也可以减少偏差，增加方差，但是在深度学习的早期阶段，我们没有太多工具可以做到只减少偏差或方差却不影响到另一方。但在当前的深度学习和大数据时代，只要持续训练一个更大的网络，只要准备了更多数据，那么也并非只有这两种情况，我们假定是这样，那么，只要正则适度，通常构建一个更大的网络便可以，在不影响方差的同时减少偏差，而采用更多数据通常可以在不过多影响偏差的同时减少方差。这两步实际要做的工作是：训练网络，选择网络或者准备更多数据，现在我们有工具可以做到在减少偏差或方差的同时，不对另一方产生过多不良影响。我觉得这就是深度学习对监督式学习大有裨益的一个重要原因，也是我们不用太过关注如何平衡偏差和方差的一个重要原因，但有时我们有很多选择，减少偏差或方差而不增加另一方。最终，我们会得到一个非常规范化的网络。从下节课开始，我们将讲解正则化，训练一个更大的网络几乎没有任何负面影响，而训练一个大型神经网络的主要代价也只是计算时间，前提是网络是比较规范化的。

今天我们讲了如何通过组织机器学习来诊断偏差和方差的基本方法，然后选择解决问题的正确操作，希望大家有所了解和认识。我在课上不止一次提到了正则化，它是一种非常实用的减少方差的方法，正则化时会出现偏差方差权衡问题，偏差可能略有增加，如果网络足够大，增幅通常不会太高，我们下节课再细讲，以便大家更好理解如何实现神经网络的正

则化。

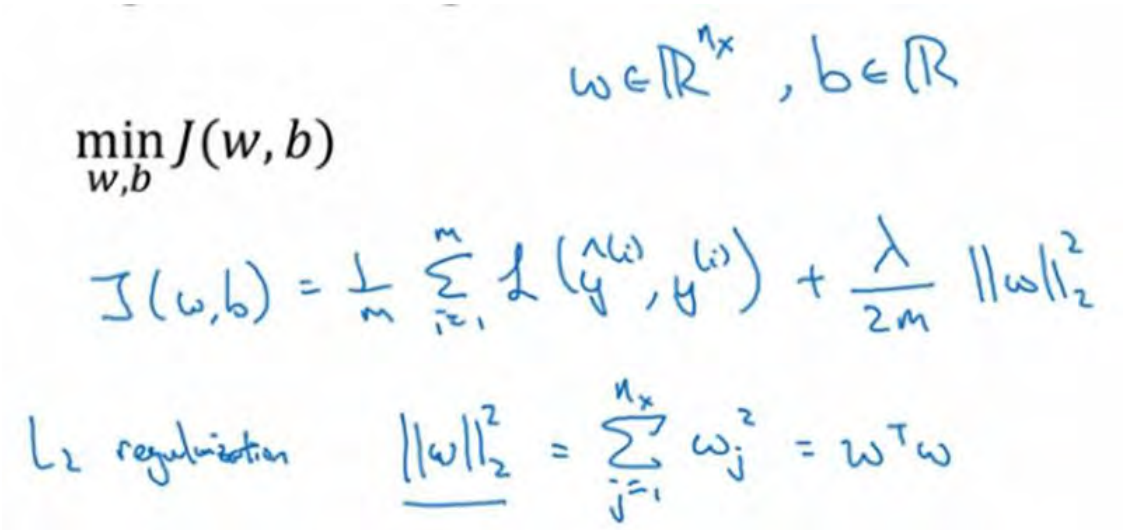
1.4 正则化（Regularization）

深度学习可能存在过拟合问题——高方差，有两个解决方法，一个是正则化，另一个是准备更多的数据，这是非常可靠的方法，但你可能无法时时刻刻准备足够多的训练数据或者获取更多数据的成本很高，但正则化通常有助于避免过拟合或减少你的网络误差。

如果你怀疑神经网络过度拟合了数据，即存在高方差问题，那么最先想到的方法可能是正则化，另一个解决高方差的方法就是准备更多数据，这也是非常可靠的办法，但你可能无法时时准备足够多的训练数据，或者，获取更多数据的成本很高，但正则化有助于避免过度拟合，或者减少网络误差，下面我们就来讲讲正则化的作用原理。

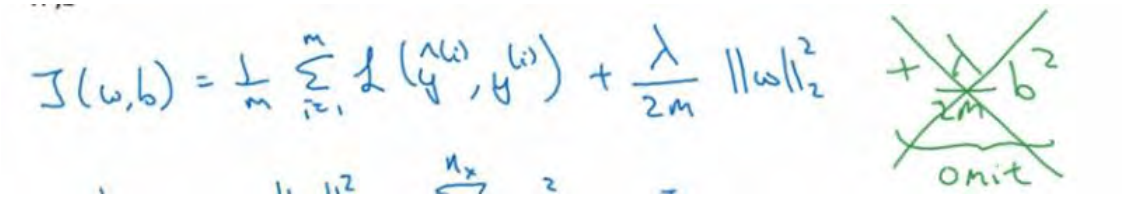
我们用逻辑回归来实现这些设想，求成本函数J的最小值，它是我们定义的成本函数，参数包含一些训练数据和不同数据中个体预测的损失，w和b是逻辑回归的两个参数，w是一个多维度参数矢量，b是一个实数。在逻辑回归函数中加入正则化，只需添加参数 λ，也就是正则化参数，一会儿再详细讲。

$\frac{\lambda}{2m}$乘以w范数的平方，w欧几里德范数的平方等于w_j（j 值从 1 到n_x）平方的和，也可表示为w^Tw，也就是向量参数w 的欧几里德范数（2 范数）的平方，此方法称为$L2$正则化。因为这里用了欧几里德法线，被称为向量参数 w 的$L2$范数。

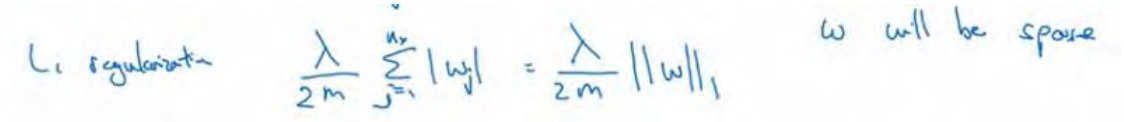


为什么只正则化参数w？为什么不再加上参数 b 呢？你可以这么做，只是我习惯省略不写，因为w通常是一个高维参数矢量，已经可以表达高偏差问题，w可能包含有很多参数，我们不可能拟合所有参数，而b只是单个数字，所以w几乎涵盖所有参数，而不是b，如果加了参数b，其实也没太大影响，因为b只是众多参数中的一个，所以我通常省略不计，如果你

想加上这个参数，完全没问题。

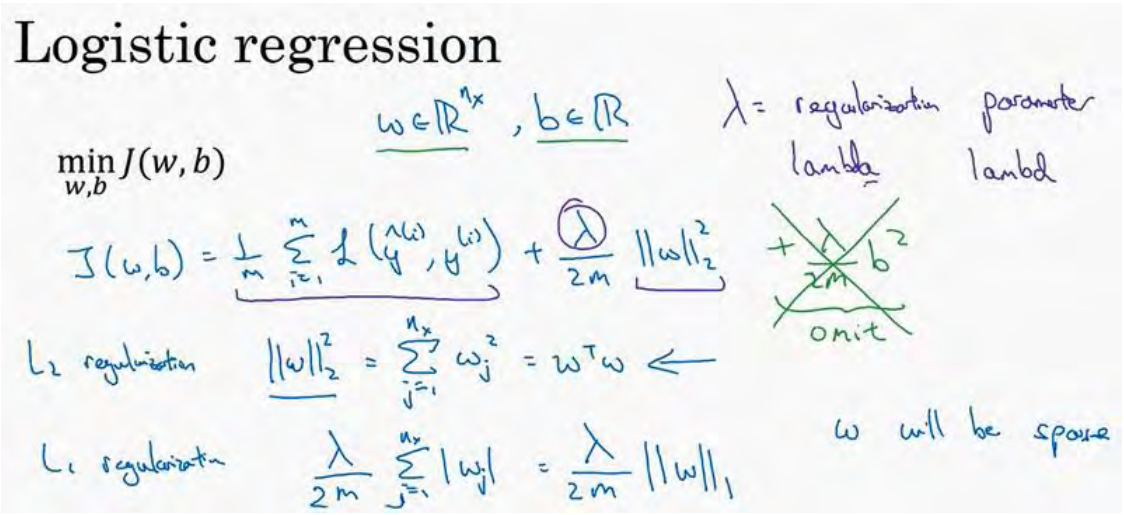


$L2$正则化是最常见的正则化类型，你们可能听说过$L1$正则化，$L1$正则化，加的不是$L2$范数，而是正则项$\frac{\lambda}{m}$乘以$\sum_{j=1}^{n_x}|w|$，$\sum_{j=1}^{n_x}|w|$也被称为参数w向量的$L1$范数，无论分母是m还是$2m$，它都是一个比例常量。

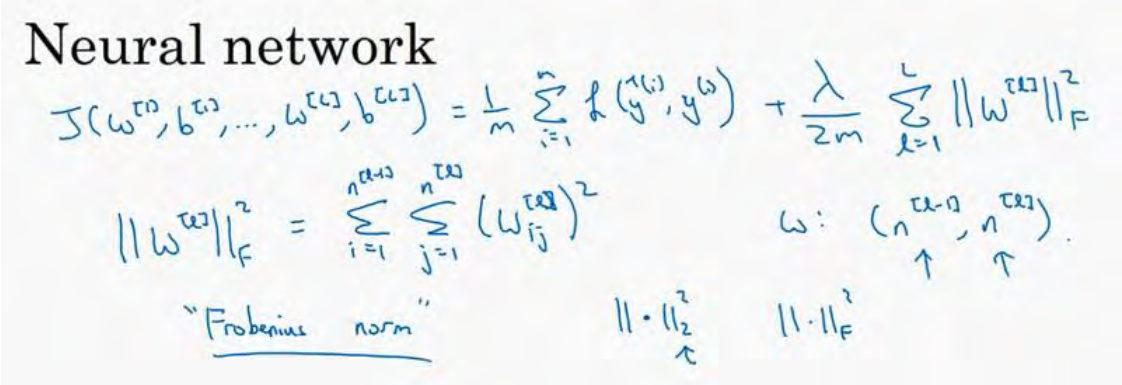


如果用的是$L1$正则化，w最终会是稀疏的，也就是说w向量中有很多 0，有人说这样有利于压缩模型，因为集合中参数均为 0，存储模型所占用的内存更少。实际上，虽然$L1$正则化使模型变得稀疏，却没有降低太多存储内存，所以我认为这并不是$L1$正则化的目的，至少不是为了压缩模型，人们在训练网络时，越来越倾向于使用$L2$正则化。

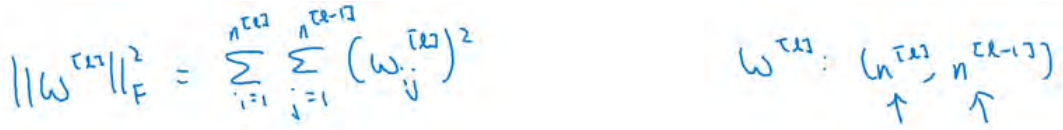
我们来看最后一个细节，λ是正则化参数，我们通常使用验证集或交叉验证集来配置这个参数，尝试各种各样的数据，寻找最好的参数，我们要考虑训练集之间的权衡，把参数设置为较小值，这样可以避免过拟合，所以 λ 是另外一个需要调整的超级参数，顺便说一下，为了方便写代码，在 **Python** 编程语言中，λ是一个保留字段，编写代码时，我们删掉a，写成$lambd$，以免与 **Python** 中的保留字段冲突，这就是在逻辑回归函数中实现$L2$正则化的过程，如何在神经网络中实现$L2$正则化呢？



神经网络含有一个成本函数，该函数包含$W^{[1]}$，$b^{[1]}$到$W^{[l]}$，$b^{[l]}$所有参数，字母 L 是神经网络所含的层数，因此成本函数等于m个训练样本损失函数的总和乘以$\frac{1}{m}$，正则项为$\frac{\lambda}{2m}\sum_1^L\|W^{[l]}\|^2$，我们称$||W^{[l]}||^2$为范数平方，这个矩阵范数$||W^{[l]}||^2$（即平方范数），被定义为矩阵中所有元素的平方求和，



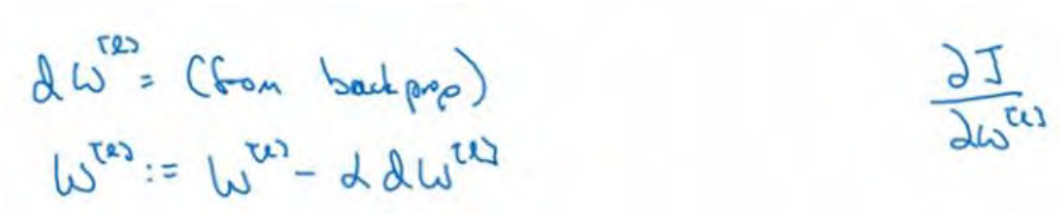
我们看下求和公式的具体参数，第一个求和符号其值 i 从 1 到$n^{[l-1]}$，第二个其J值从 1 到$n^{[l]}$，因为W是一个$n^{[l]} \times n^{[l-1]}$的多维矩阵，$n^{[l]}$表示l 层单元的数量，$n^{[l-1]}$表示第$l-1$层隐藏单元的数量。



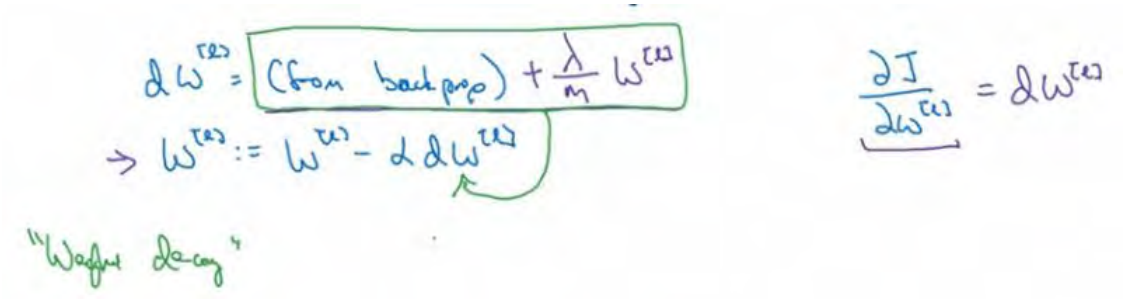
该矩阵范数被称作“弗罗贝尼乌斯范数”，用下标F标注”，鉴于线性代数中一些神秘晦涩的原因，我们不称之为“矩阵$L2$范数”，而称它为“弗罗贝尼乌斯范数”，矩阵$L2$范数听起来更自然，但鉴于一些大家无须知道的特殊原因，按照惯例，我们称之为“弗罗贝尼乌斯范数”，它表示一个矩阵中所有元素的平方和。

该如何使用该范数实现梯度下降呢？

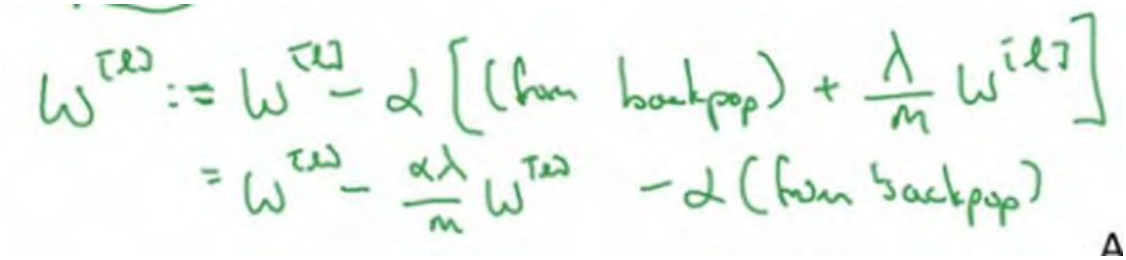
用 **backprop** 计算出dW的值，**backprop** 会给出J对W的偏导数，实际上是$W^{[l]}$，把$W^{[l]}$替换为$W^{[l]}$减去学习率乘以dW。



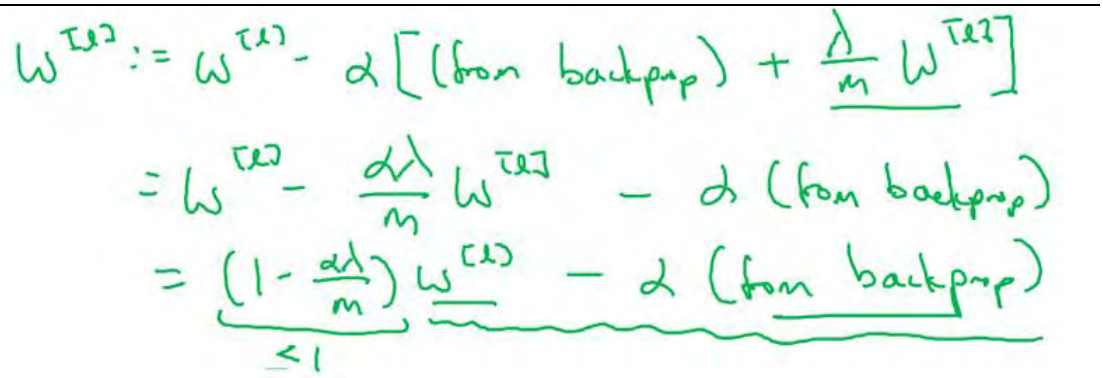
这就是之前我们额外增加的正则化项，既然已经增加了这个正则项，现在我们要做的就是给dW加上这一项$\frac{\lambda}{m}W^{[l]}$，然后计算这个更新项，使用新定义的$dW^{[l]}$，它的定义含有相关参数代价函数导数和，以及最后添加的额外正则项，这也是$L2$正则化有时被称为“权重衰减”的原因。



我们用$dW^{[l]}$的定义替换此处的$dW^{[l]}$，可以看到，$W^{[l]}$的定义被更新为$W^{[l]}$减去学习率a 乘以 **backprop** 再加上$\frac{\lambda}{m}W^{[l]}$。



该正则项说明，不论$W^{[l]}$是什么，我们都试图让它变得更小，实际上，相当于我们给矩阵 W 乘以$(1-a\frac{\lambda}{m})$倍的权重，矩阵W减去$\alpha\frac{\lambda}{m}$倍的它，也就是用这个系数$(1-a\frac{\lambda}{m})$乘以矩阵W，该系数小于 1，因此$L2$范数正则化也被称为“权重衰减”，因为它就像一般的梯度下降，W被更新为少了a乘以 **backprop** 输出的最初梯度值，同时W也乘以了这个系数，这个系数小于 1，因此$L2$正则化也被称为“权重衰减”。

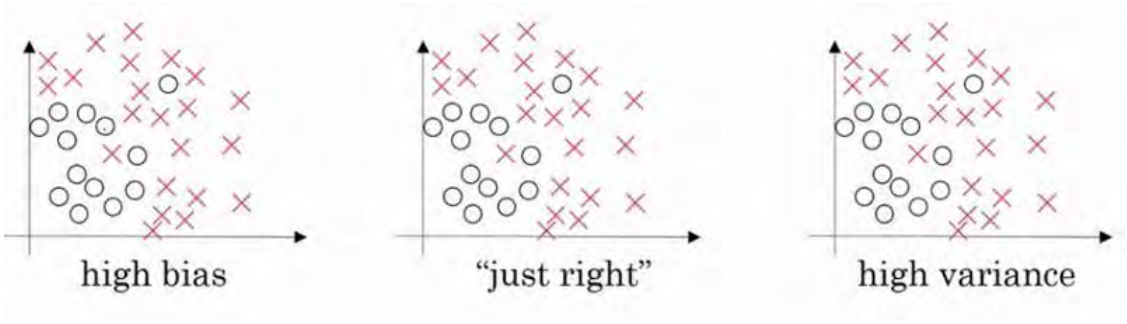


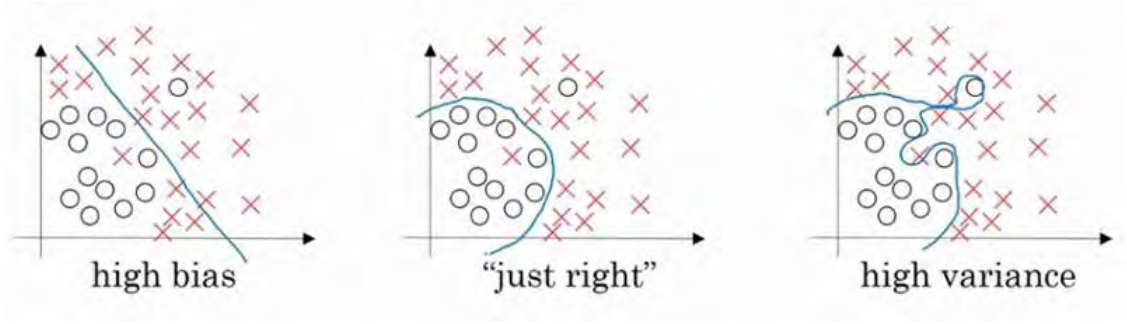
我不打算这么叫它，之所以叫它“权重衰减”是因为这两项相等，权重指标乘以了一个小于 1 的系数。

以上就是在神经网络中应用$L2$正则化的过程，有人会问我，为什么正则化可以预防过拟合，我们放在下节课讲，同时直观感受一下正则化是如何预防过拟合的。

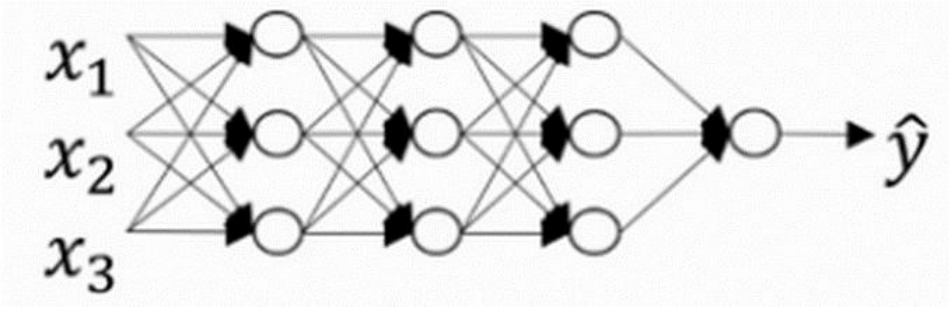
1.5 为什么正则化有利于预防过拟合呢？（Why regularization reduces overfitting?）

为什么正则化有利于预防过拟合呢？为什么它可以减少方差问题？我们通过两个例子来直观体会一下。





左图是高偏差，右图是高方差，中间是 **Just Right**，这几张图我们在前面课程中看到过。



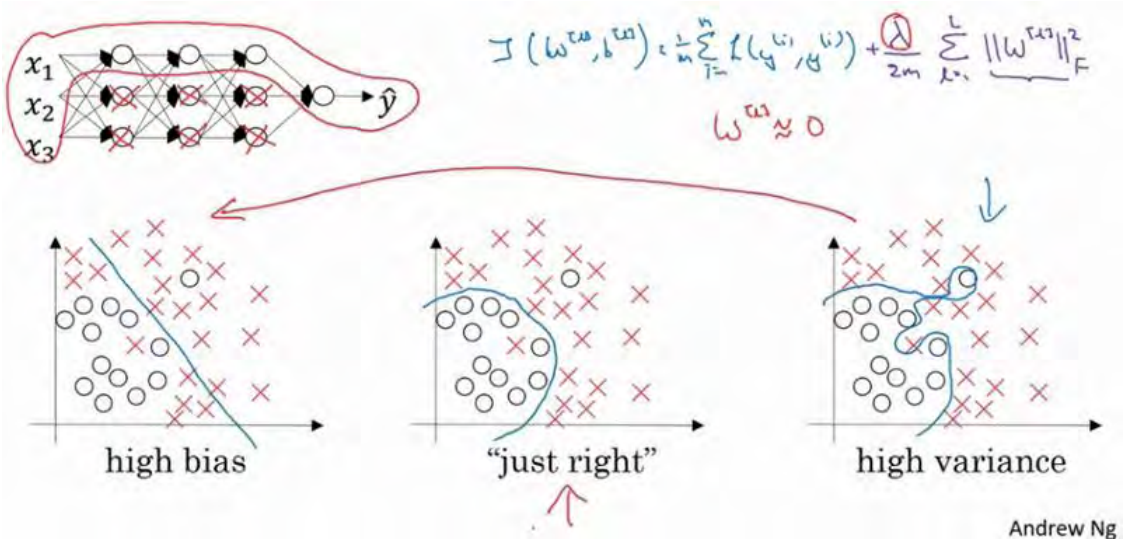
现在我们来看下这个庞大的深度拟合神经网络。我知道这张图不够大，深度也不够，但你可以想象这是一个过拟合的神经网络。这是我们的代价函数J，含有参数W，b。我们添加正则项，它可以避免数据权值矩阵过大，这就是弗罗贝尼乌斯范数，为什么压缩$L2$范数，或者弗罗贝尼乌斯范数或者参数可以减少过拟合？

直观上理解就是如果正则化λ设置得足够大，权重矩阵W被设置为接近于 0 的值，直观理解就是把多隐藏单元的权重设为 0，于是基本上消除了这些隐藏单元的许多影响。如果是

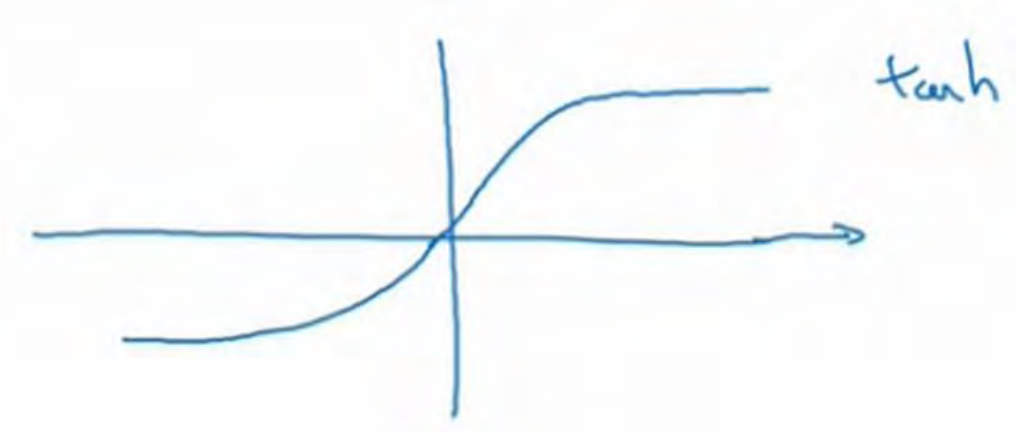
这种情况，这个被大大简化了的神经网络会变成一个很小的网络，小到如同一个逻辑回归单元，可是深度却很大，它会使这个网络从过度拟合的状态更接近左图的高偏差状态。

但是λ会存在一个中间值，于是会有一个接近“**Just Right**”的中间状态。

直观理解就是λ增加到足够大，W会接近于 0，实际上是不会发生这种情况的，我们尝试消除或至少减少许多隐藏单元的影响，最终这个网络会变得更简单，这个神经网络越来越接近逻辑回归，我们直觉上认为大量隐藏单元被完全消除了，其实不然，实际上是该神经网络的所有隐藏单元依然存在，但是它们的影响变得更小了。神经网络变得更简单了，貌似这样更不容易发生过拟合，因此我不确定这个直觉经验是否有用，不过在编程中执行正则化时，你实际看到一些方差减少的结果。

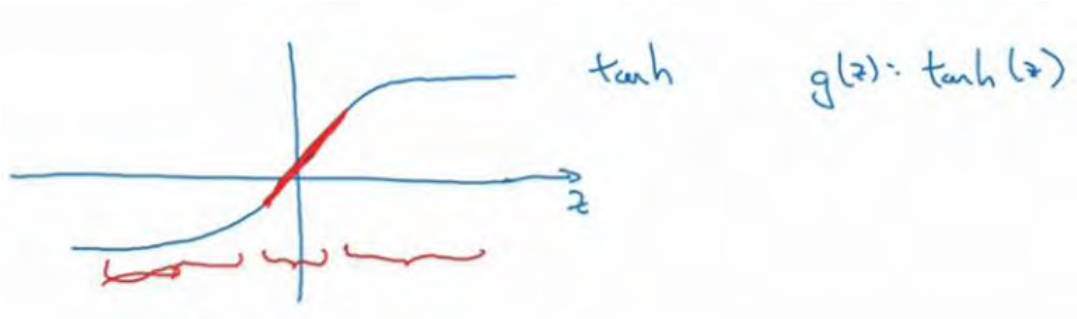


我们再来直观感受一下，正则化为什么可以预防过拟合，假设我们用的是这样的双曲线激活函数。

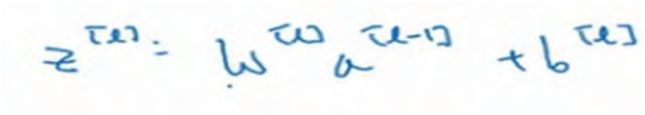


用$g(z)$表示$tanh(z)$,那么我们发现，只要z非常小，如果z只涉及少量参数，这里我们利用了双曲正切函数的线性状态，只要z可以扩展为这样的更大值或者更小值，激活函数开始

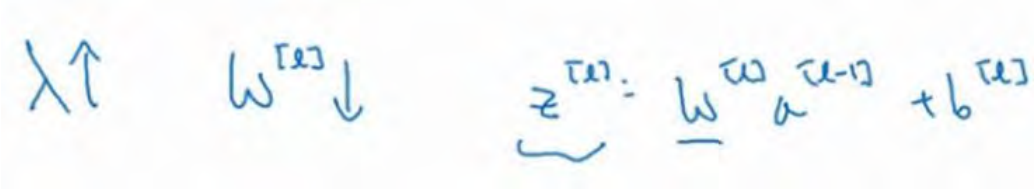
变得非线性。



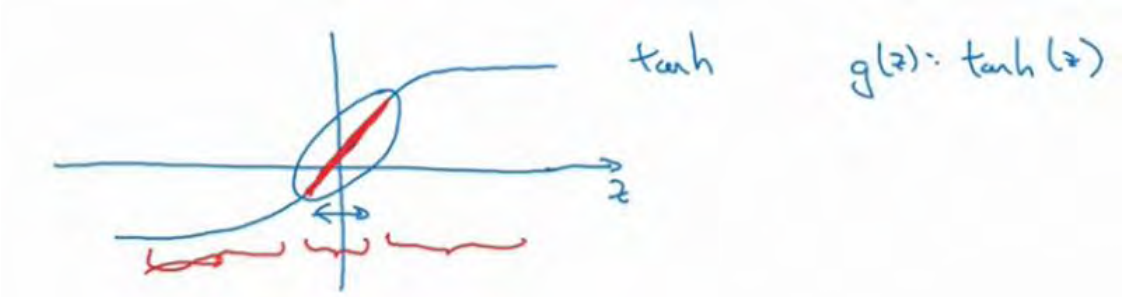
现在你应该摒弃这个直觉，如果正则化参数 λ 很大，激活函数的参数会相对较小，因为代价函数中的参数变大了，如果W很小，



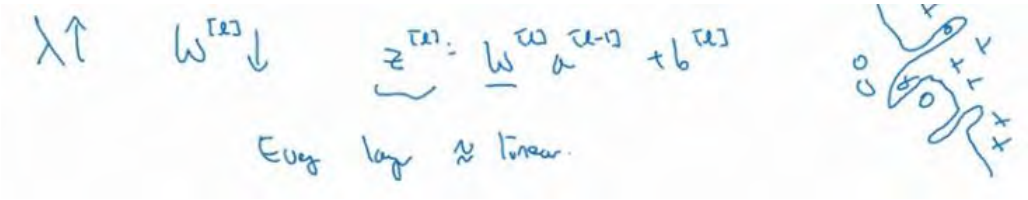
如果W很小，相对来说，z也会很小。



特别是，如果z的值最终在这个范围内，都是相对较小的值，$g(z)$大致呈线性，每层几乎都是线性的，和线性回归函数一样。

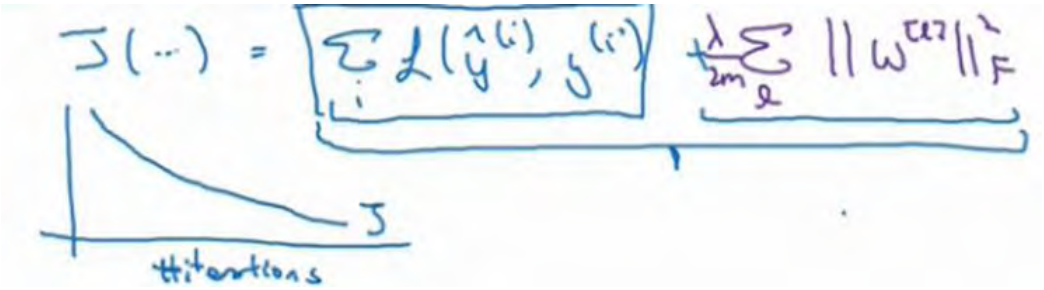


第一节课我们讲过，如果每层都是线性的，那么整个网络就是一个线性网络，即使是一个非常深的深层网络，因具有线性激活函数的特征，最终我们只能计算线性函数，因此，它不适用于非常复杂的决策，以及过度拟合数据集的非线性决策边界，如同我们在幻灯片中看到的过度拟合高方差的情况。



总结一下，如果正则化参数变得很大，参数W很小，z也会相对变小，此时忽略b的影响，z会相对变小，实际上，z的取值范围很小，这个激活函数，也就是曲线函数$tanh$会相对呈线性，整个神经网络会计算离线性函数近的值，这个线性函数非常简单，并不是一个极复杂的高度非线性函数，不会发生过拟合。

大家在编程作业里实现正则化的时候，会亲眼看到这些结果，总结正则化之前，我给大家一个执行方面的小建议，在增加正则化项时，应用之前定义的代价函数J，我们做过修改，增加了一项，目的是预防权重过大。

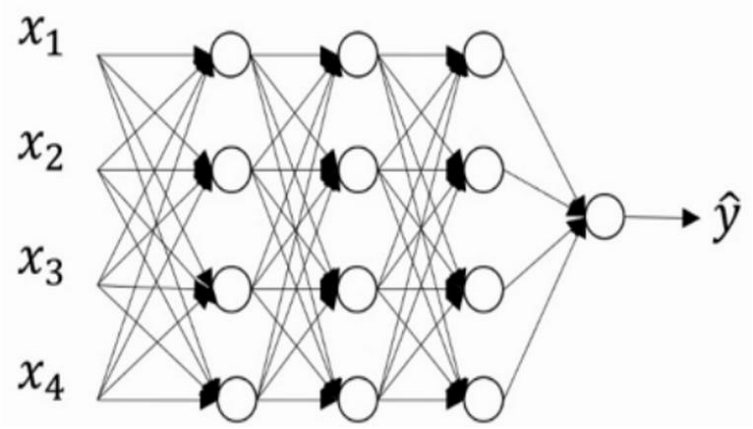


如果你使用的是梯度下降函数，在调试梯度下降时，其中一步就是把代价函数J设计成这样一个函数，在调试梯度下降时，其中一步就是把代价函数J设计成这样一个函数，它代表梯度下降的调幅数量。可以看到，代价函数对于梯度下降的每个调幅都单调递减。如果你实施的是正则化函数，请牢记，J已经有一个全新的定义。如果你用的是原函数J，也就是这第一个项正则化项，你可能看不到单调递减现象，为了调试梯度下降，请务必使用新定义的J函数，它包含第二个正则化项，否则函数J可能不会在所有调幅范围内都单调递减。

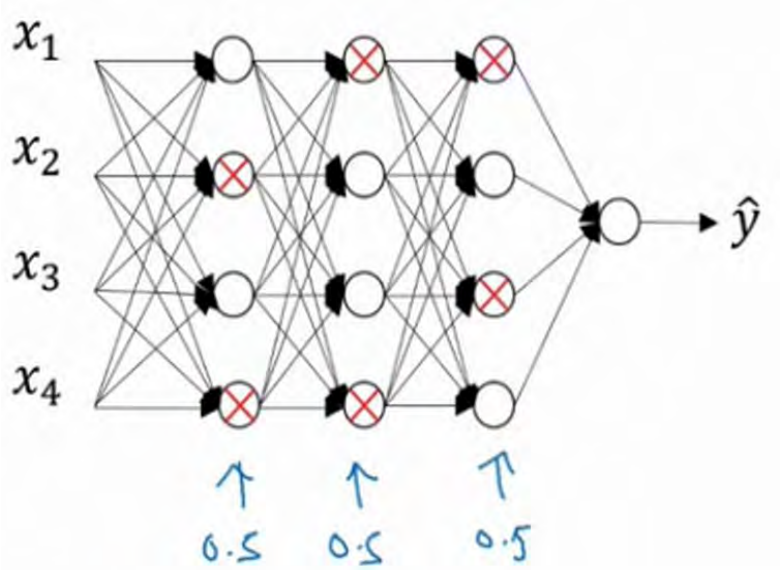
这就是$L2$正则化，它是我在训练深度学习模型时最常用的一种方法。在深度学习中，还有一种方法也用到了正则化，就是 **dropout** 正则化，我们下节课再讲。

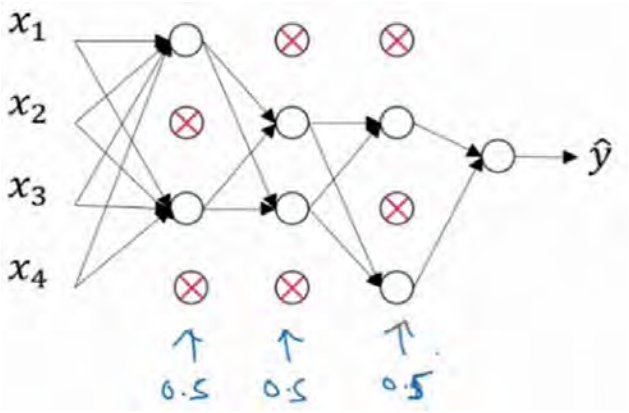
1.6 dropout 正则化（Dropout Regularization）

除了$L2$正则化，还有一个非常实用的正则化方法——“**Dropout**（随机失活）”，我们来看看它的工作原理。

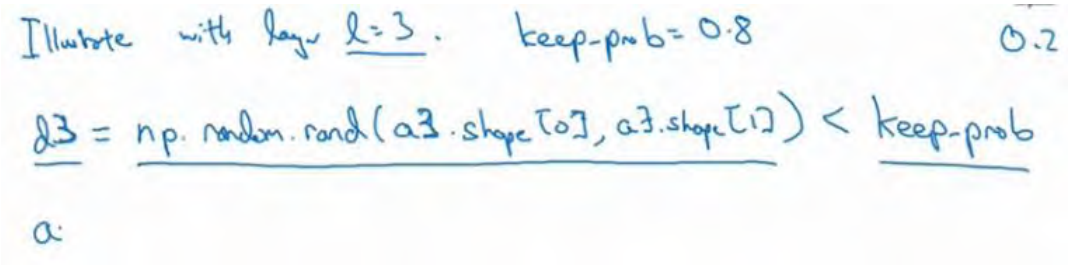


假设你在训练上图这样的神经网络，它存在过拟合，这就是 **dropout** 所要处理的，我们复制这个神经网络，**dropout** 会遍历网络的每一层，并设置消除神经网络中节点的概率。假设网络中的每一层，每个节点都以抛硬币的方式设置概率，每个节点得以保留和消除的概率都是 0.5，设置完节点概率，我们会消除一些节点，然后删除掉从该节点进出的连线，最后得到一个节点更少，规模更小的网络，然后用 **backprop** 方法进行训练。





这是网络节点精简后的一个样本，对于其它样本，我们照旧以抛硬币的方式设置概率，保留一类节点集合，删除其它类型的节点集合。对于每个训练样本，我们都将采用一个精简后神经网络来训练它，这种方法似乎有点怪，单纯遍历节点，编码也是随机的，可它真的有效。不过可想而知，我们针对每个训练样本训练规模极小的网络，最后你可能会认识到为什么要正则化网络，因为我们在训练极小的网络。



如何实施 **dropout** 呢？方法有几种，接下来我要讲的是最常用的方法，即 **inverted dropout**（反向随机失活），出于完整性考虑，我们用一个三层（$l=3$）网络来举例说明。编码中会有很多涉及到 3 的地方。我只举例说明如何在某一层中实施 **dropout**。

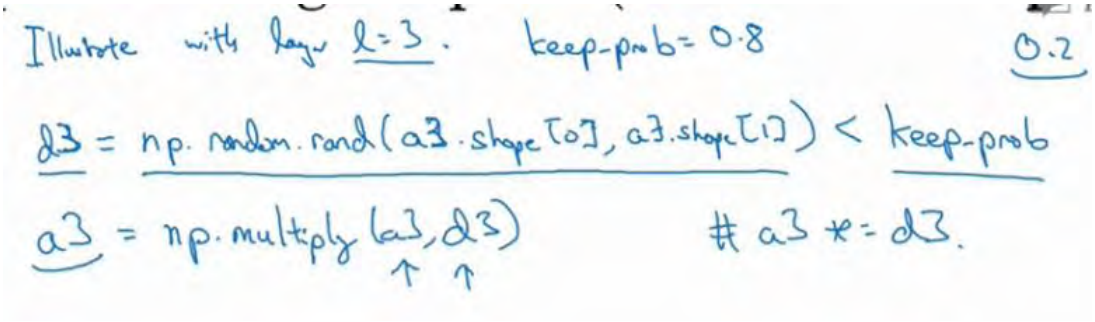
首先要定义向量d，$d^{[3]}$表示一个三层的 **dropout** 向量：

```
d3 = np.random.rand(a3.shape[0],a3.shape[1])
```

然后看它是否小于某数，我们称之为 **keep-prob**，**keep-prob** 是一个具体数字，上个示例中它是 0.5，而本例中它是 0.8，它表示保留某个隐藏单元的概率，此处 **keep-prob** 等于 0.8，它意味着消除任意一个隐藏单元的概率是 0.2，它的作用就是生成随机矩阵，如果对$a^{[3]}$进行因子分解，效果也是一样的。$d^{[3]}$是一个矩阵，每个样本和每个隐藏单元，其中$d^{[3]}$中的对应值为 1 的概率都是 0.8，对应为 0 的概率是 0.2，随机数字小于 0.8。它等于 1 的概率是 0.8，等于 0 的概率是 0.2。

接下来要做的就是从第三层中获取激活函数，这里我们叫它$a^{[3]}$，$a^{[3]}$含有要计算的激活函数，$a^{[3]}$等于上面的$a^{[3]}$乘以$d^{\[3]}$，`a3 =np.multiply( a3,d3)`，这里是元素相乘，

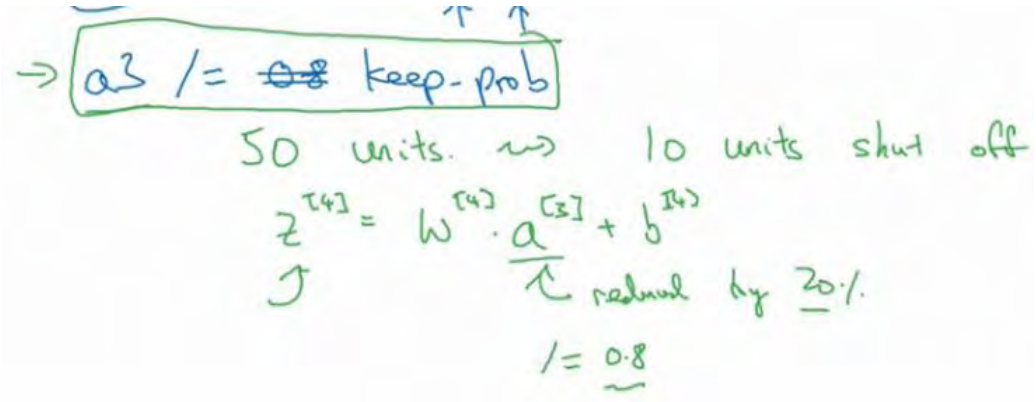
也可写为$a3 *= d3$，它的作用就是让$d^{[3]}$中所有等于 0 的元素（输出），而各个元素等于 0 的概率只有 20%，乘法运算最终把$d^{[3]}$中相应元素输出，即让$d^{[3]}$中 0 元素与$a^{[3]}$中相对元素归零。



如果用 python 实现该算法的话，$d^{[3]}$则是一个布尔型数组，值为 **true** 和 **false**，而不是 1 和 0，乘法运算依然有效，python 会把 **true** 和 **false** 翻译为 1 和 0，大家可以用 python 尝试一下。

最后，我们向外扩展$a^{[3]}$，用它除以 0.8，或者除以 **keep-prob** 参数。

下面我解释一下为什么要这么做，为方便起见，我们假设第三隐藏层上有 50 个单元或 50 个神经元，在一维上$a^{[3]}$是 50，我们通过因子分解将它拆分成$50 \times m$维的，保留和删除它们的概率分别为 80%和 20%，这意味着最后被删除或归零的单元平均有 10（50×20%=10）个，现在我们看下$z^{[4]}$，$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$，我们的预期是，$a^{[3]}$减少 20%，也就是说$a^{[3]}$中有 20%的元素被归零，为了不影响$z^{[4]}$的期望值，我们需要用$w^{[4]}a^{[3]}/0.8$，它将会修正或弥补我们所需的那 20%，$a^{[3]}$的期望值不会变，划线部分就是所谓的 **dropout** 方法。



它的功能是，不论 **keep-prop** 的值是多少 0.8，0.9 甚至是 1，如果 **keep-prop** 设置为 1，那么就不存在 **dropout**，因为它会保留所有节点。反向随机失活（**inverted dropout**）方法通过除以 **keep-prob**，确保$a^{[3]}$的期望值不变。

事实证明，在测试阶段，当我们评估一个神经网络时，也就是用绿线框标注的反向随机失活方法，使测试阶段变得更容易，因为它的数据扩展问题变少，我们将在下节课讨论。

据我了解，目前实施 **dropout** 最常用的方法就是 **Inverted dropout**，建议大家动手实践一下。**Dropout** 早期的迭代版本都没有除以 **keep-prob**，所以在测试阶段，平均值会变得越来越复杂，不过那些版本已经不再使用了。

现在你使用的是d向量，你会发现，不同的训练样本，清除不同的隐藏单元也不同。实际上，如果你通过相同训练集多次传递数据，每次训练数据的梯度不同，则随机对不同隐藏单元归零，有时却并非如此。比如，需要将相同隐藏单元归零，第一次迭代梯度下降时，把一些隐藏单元归零，第二次迭代梯度下降时，也就是第二次遍历训练集时，对不同类型的隐藏层单元归零。向量d或$d^{[3]}$用来决定第三层中哪些单元归零，无论用 **foreprop** 还是 **backprop**，这里我们只介绍了 **foreprob**。

如何在测试阶段训练算法，在测试阶段，我们已经给出了x，或是想预测的变量，用的是标准计数法。我用$a^{[0]}$，第 0 层的激活函数标注为测试样本x，我们在测试阶段不使用 **dropout** 函数，尤其是像下列情况：

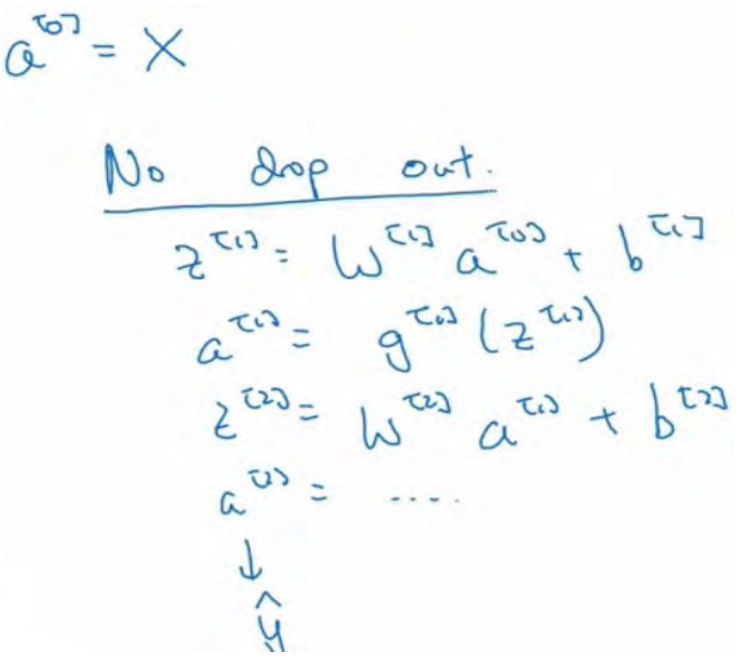
$$z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \cdots$$

以此类推直到最后一层，预测值为$\hat{y}$。



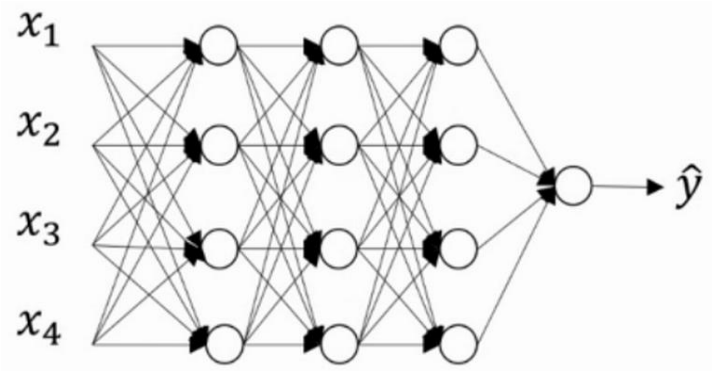
显然在测试阶段，我们并未使用 **dropout**，自然也就不用抛硬币来决定失活概率，以及要消除哪些隐藏单元了，因为在测试阶段进行预测时，我们不期望输出结果是随机的，如果测试阶段应用 **dropout** 函数，预测会受到干扰。理论上，你只需要多次运行预测处理过程，每一次，不同的隐藏单元会被随机归零，预测处理遍历它们，但计算效率低，得出的结果也几乎相同，与这个不同程序产生的结果极为相似。

Inverted dropout 函数在除以 **keep-prob** 时可以记住上一步的操作，目的是确保即使在测试阶段不执行 **dropout** 来调整数值范围，激活函数的预期结果也不会发生变化，所以没必要在测试阶段额外添加尺度参数，这与训练阶段不同。

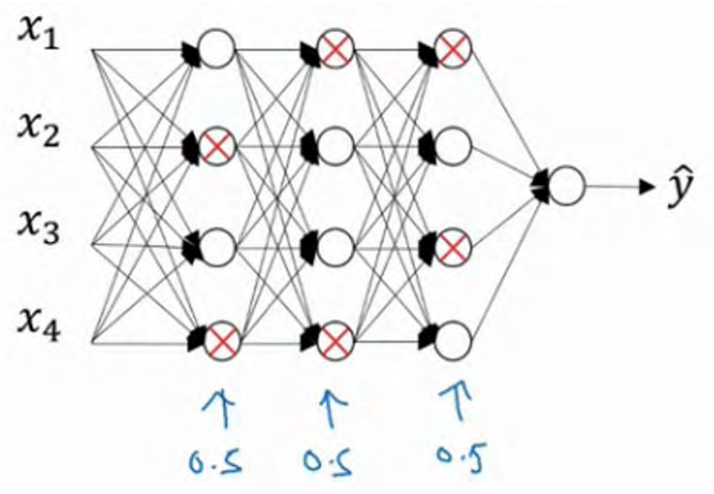
$$l = keep - prob$$

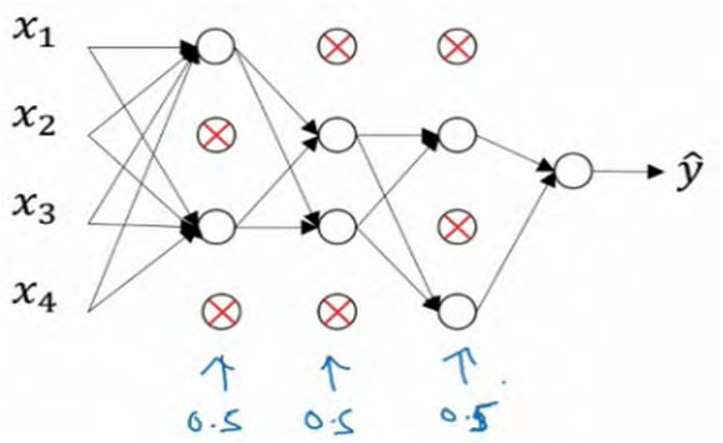
这就是 **dropout**，大家可以通过本周的编程练习来执行这个函数，亲身实践一下。

为什么 **dropout** 会起作用呢？下节课我们将更加直观地了解 **dropout** 的具体功能除了$L2$正则化，还有一个非常实用的正则化方法——“**Dropout**（随机失活）”，我们来看看它的工作原理。



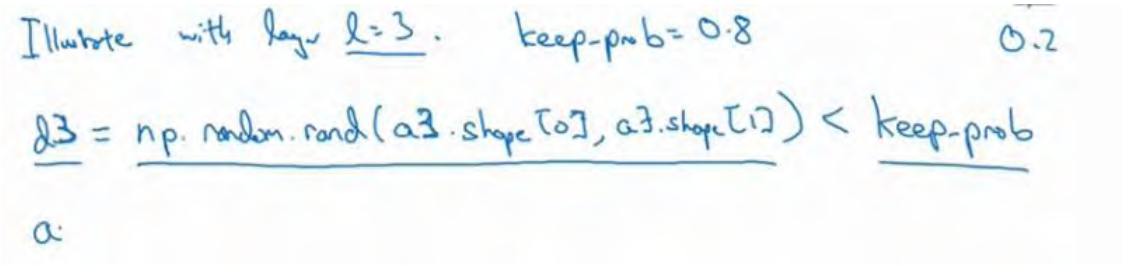
假设你在训练上图这样的神经网络，它存在过拟合，这就是 **dropout** 所要处理的，我们复制这个神经网络，**dropout** 会遍历网络的每一层，并设置消除神经网络中节点的概率。假设网络中的每一层，每个节点都以抛硬币的方式设置概率，每个节点得以保留和消除的概率都是 0.5，设置完节点概率，我们会消除一些节点，然后删除掉从该节点进出的连线，最后得到一个节点更少，规模更小的网络，然后用 **backprop** 方法进行训练。





这是网络节点精简后的一个样本，对于其它样本，我们照旧以抛硬币的方式设置概率，保留一类节点集合，删除其它类型的节点集合。对于每个训练样本，我们都将采用一个精简

后神经网络来训练它，这种方法似乎有点怪，单纯遍历节点，编码也是随机的，可它真的有效。不过可想而知，我们针对每个训练样本训练规模极小的网络，最后你可能会认识到为什么要正则化网络，因为我们在训练极小的网络。



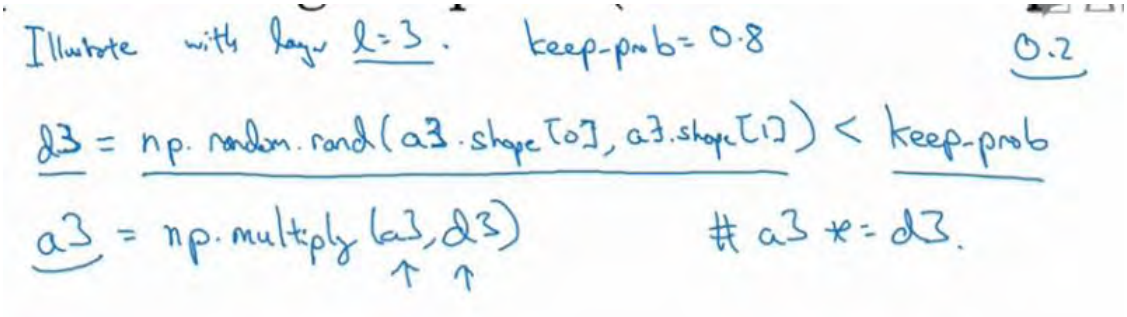
如何实施 **dropout** 呢？方法有几种，接下来我要讲的是最常用的方法，即 **inverted dropout**（反向随机失活），出于完整性考虑，我们用一个三层（$l=3$）网络来举例说明。编码中会有很多涉及到 3 的地方。我只举例说明如何在某一层中实施 **dropout**。

首先要定义向量d，$d^{[3]}$表示一个三层的 **dropout** 向量：

```
d3 = np.random.rand( a3.shape[0],a3.shape[1])
```

然后看它是否小于某数，我们称之为 **keep-prob**，**keep-prob** 是一个具体数字，上个示例中它是 0.5，而本例中它是 0.8，它表示保留某个隐藏单元的概率，此处 **keep-prob** 等于 0.8，它意味着消除任意一个隐藏单元的概率是 0.2，它的作用就是生成随机矩阵，如果对$a^{[3]}$进行因子分解，效果也是一样的。$d^{[3]}$是一个矩阵，每个样本和每个隐藏单元，其中$d^{[3]}$中的对应值为 1 的概率都是 0.8，对应为 0 的概率是 0.2，随机数字小于 0.8。它等于 1 的概率是 0.8，等于 0 的概率是 0.2。

接下来要做的就是从第三层中获取激活函数，这里我们叫它$a^{[3]}$，$a^{[3]}$含有要计算的激活函数，$a^{[3]}$等于上面的$a^{[3]}$乘以$d^{[3]}$，`a3 =np.multiply(a3,d3)`，这里是元素相乘，也可写为$a3 *= d3$，它的作用就是让$d^{[3]}$中所有等于 0 的元素（输出），而各个元素等于 0 的概率只有 20%，乘法运算最终把$d^{[3]}$中相应元素输出，即让$d^{[3]}$中 0 元素与$a^{[3]}$中相对元素归零。

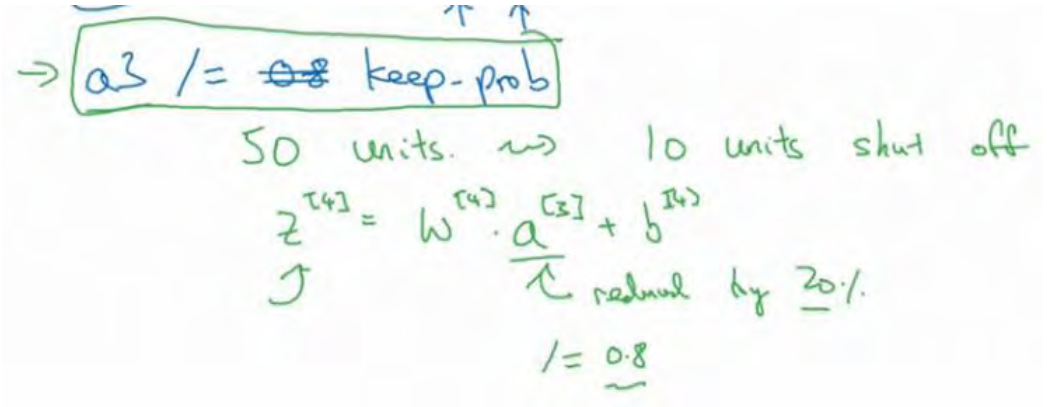


如果用 **python** 实现该算法的话，$d^{[3]}$则是一个布尔型数组，值为 **true** 和 **false**，而不是

1 和 0，乘法运算依然有效，**python** 会把 **true** 和 **false** 翻译为 1 和 0，大家可以用 **python** 尝试一下。

最后，我们向外扩展$a^{[3]}$，用它除以 0.8，或者除以 **keep-prob** 参数。

下面我解释一下为什么要这么做，为方便起见，我们假设第三隐藏层上有 50 个单元或 50 个神经元，在一维上$a^{[3]}$是 50，我们通过因子分解将它拆分成$50 \times m$维的，保留和删除它们的概率分别为 80%和 20%，这意味着最后被删除或归零的单元平均有 10（50×20%=10）个，现在我们看下$z^{[4]}$，$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$，我们的预期是，$a^{[3]}$减少 20%，也就是说$a^{[3]}$中有 20%的元素被归零，为了不影响$z^{[4]}$的期望值，我们需要用$w^{[4]}a^{[3]}/0.8$，它将会修正或弥补我们所需的那 20%，$a^{[3]}$的期望值不会变，划线部分就是所谓的 **dropout** 方法。



它的功能是，不论 **keep-prop** 的值是多少 0.8，0.9 甚至是 1，如果 **keep-prop** 设置为 1，那么就不存在 **dropout**，因为它会保留所有节点。反向随机失活（**inverted dropout**）方法通过除以 **keep-prob**，确保$a^{[3]}$的期望值不变。

事实证明，在测试阶段，当我们评估一个神经网络时，也就是用绿线框标注的反向随机失活方法，使测试阶段变得更容易，因为它的数据扩展问题变少，我们将在下节课讨论。

据我了解，目前实施 **dropout** 最常用的方法就是 **Inverted dropout**，建议大家动手实践一下。**Dropout** 早期的迭代版本都没有除以 **keep-prob**，所以在测试阶段，平均值会变得越来越复杂，不过那些版本已经不再使用了。

现在你使用的是d向量，你会发现，不同的训练样本，清除不同的隐藏单元也不同。实际上，如果你通过相同训练集多次传递数据，每次训练数据的梯度不同，则随机对不同隐藏单元归零，有时却并非如此。比如，需要将相同隐藏单元归零，第一次迭代梯度下降时，把一些隐藏单元归零，第二次迭代梯度下降时，也就是第二次遍历训练集时，对不同类型的隐

藏层单元归零。向量d或$d^{[3]}$用来决定第三层中哪些单元归零，无论用 **foreprop** 还是 **backprop**，这里我们只介绍了 **foreprob**。

如何在测试阶段训练算法，在测试阶段，我们已经给出了x，或是想预测的变量，用的是标准计数法。我用$a^{[0]}$，第 0 层的激活函数标注为测试样本x，我们在测试阶段不使用 **dropout** 函数，尤其是像下列情况：

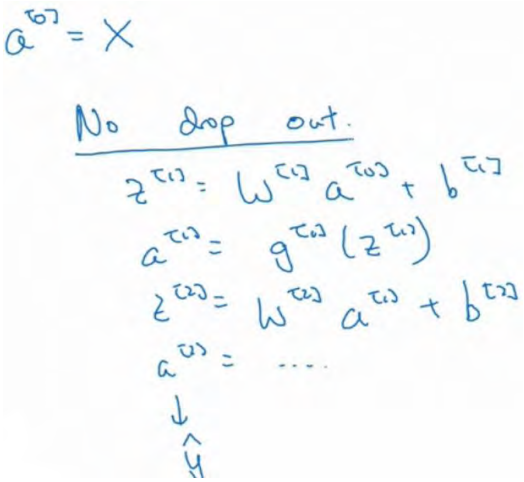
$$z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \cdots$$

以此类推直到最后一层，预测值为$\hat{y}$。



显然在测试阶段，我们并未使用 **dropout**，自然也就不用抛硬币来决定失活概率，以及要消除哪些隐藏单元了，因为在测试阶段进行预测时，我们不期望输出结果是随机的，如果测试阶段应用 **dropout** 函数，预测会受到干扰。理论上，你只需要多次运行预测处理过程，每一次，不同的隐藏单元会被随机归零，预测处理遍历它们，但计算效率低，得出的结果也几乎相同，与这个不同程序产生的结果极为相似。

Inverted dropout 函数在除以 **keep-prob** 时可以记住上一步的操作，目的是确保即使在测试阶段不执行 **dropout** 来调整数值范围，激活函数的预期结果也不会发生变化，所以没必要在测试阶段额外添加尺度参数，这与训练阶段不同。

$$l = keep - prob$$

这就是 **dropout**，大家可以通过本周的编程练习来执行这个函数，亲身实践一下。

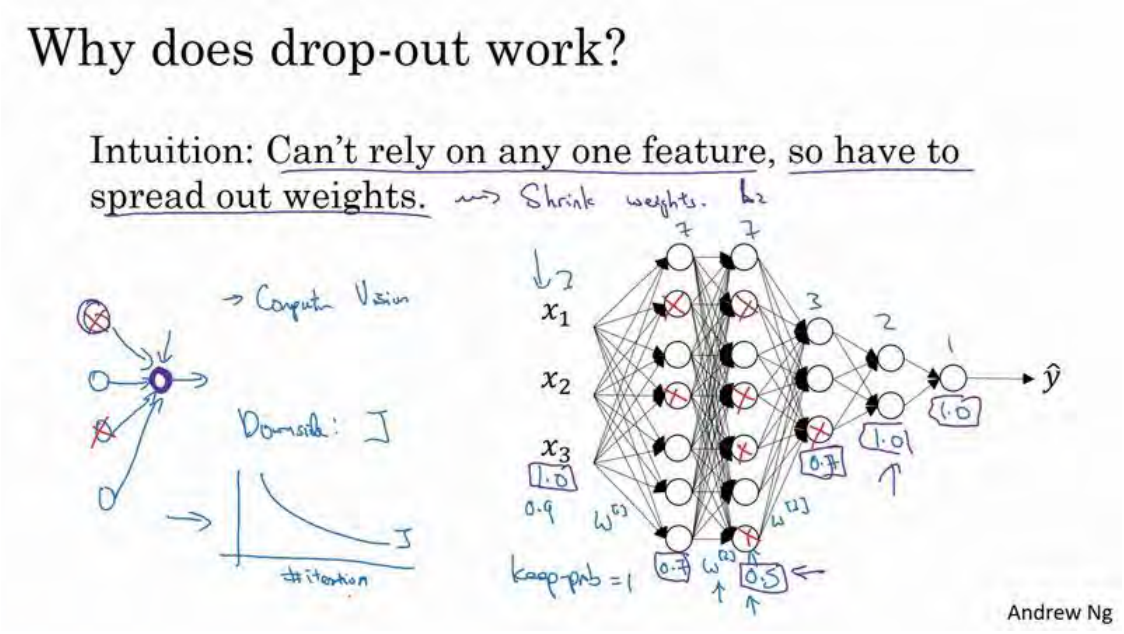
为什么 **dropout** 会起作用呢？下节课我们将更加直观地了解 **dropout** 的具体功能。

1.7 理解 dropout（Understanding Dropout）

Dropout 可以随机删除网络中的神经单元，他为什么可以通过正则化发挥如此大的作用呢？

直观上理解：不要依赖于任何一个特征，因为该单元的输入可能随时被清除，因此该单元通过这种方式传播下去，并为单元的四个输入增加一点权重，通过传播所有权重，**dropout** 将产生收缩权重的平方范数的效果，和之前讲的$L2$正则化类似；实施 **dropout** 的结果实它会压缩权重，并完成一些预防过拟合的外层正则化；$L2$对不同权重的衰减是不同的，它取决于激活函数倍增的大小。

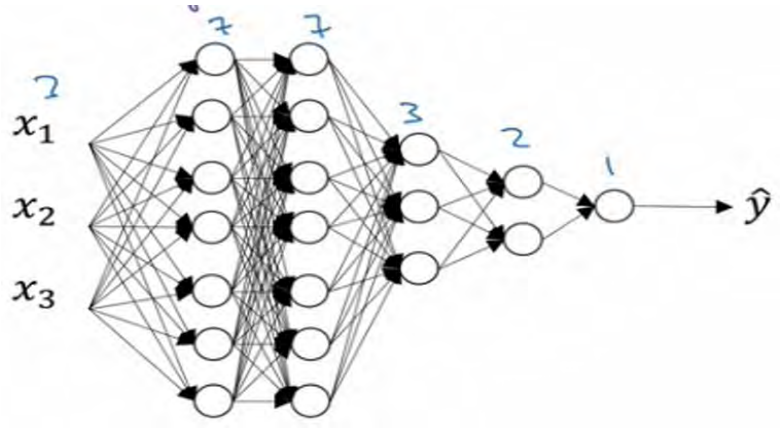
总结一下，**dropout** 的功能类似于$L2$正则化，与$L2$正则化不同的是应用方式不同会带来一点点小变化，甚至更适用于不同的输入范围。



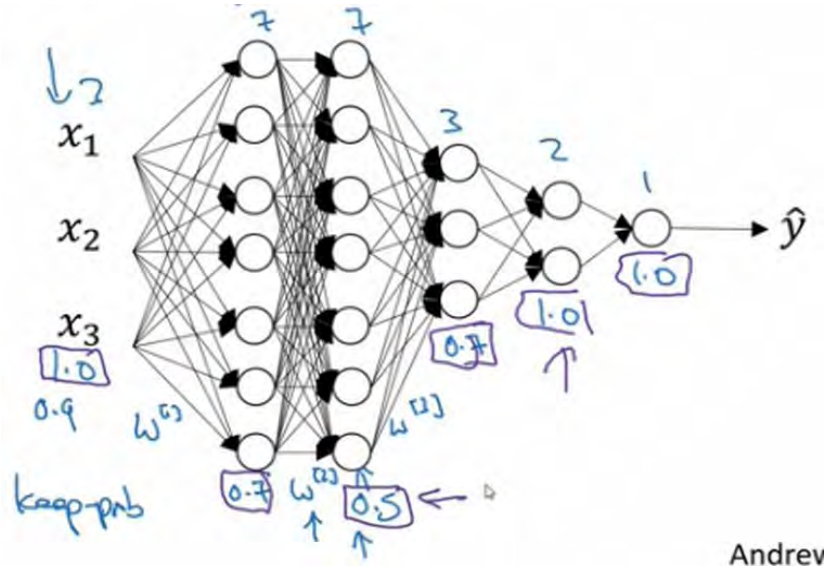
第二个直观认识是，我们从单个神经元入手，如图，这个单元的工作就是输入并生成一些有意义的输出。通过 **dropout**，该单元的输入几乎被消除，有时这两个单元会被删除，有时会删除其它单元，就是说，我用紫色圈起来的这个单元，它不能依靠任何特征，因为特征都有可能被随机清除，或者说该单元的输入也都可能被随机清除。我不愿意把所有赌注都放在一个节点上，不愿意给任何一个输入加上太多权重，因为它可能会被删除，因此该单元将通过这种方式积极地传播开，并为单元的四个输入增加一点权重，通过传播所有权重，**dropout** 将产生收缩权重的平方范数的效果，和我们之前讲过的$L2$正则化类似，实施 **dropout** 的结果是它会压缩权重，并完成一些预防过拟合的外层正则化。

事实证明，**dropout** 被正式地作为一种正则化的替代形式，$L2$对不同权重的衰减是不同的，它取决于倍增的激活函数的大小。

总结一下，**dropout** 的功能类似于$L2$正则化，与$L2$正则化不同的是，被应用的方式不同，**dropout** 也会有所不同，甚至更适用于不同的输入范围。



实施 **dropout** 的另一个细节是，这是一个拥有三个输入特征的网络，其中一个要选择的参数是 **keep-prob**，它代表每一层上保留单元的概率。所以不同层的 **keep-prob** 也可以变化。第一层，矩阵$W^{[1]}$是 7×3，第二个权重矩阵$W^{[2]}$是 7×7，第三个权重矩阵$W^{[3]}$是 3×7，以此类推，$W^{[2]}$是最大的权重矩阵，因为$W^{[2]}$拥有最大参数集，即 7×7，为了预防矩阵的过拟合，对于这一层，我认为这是第二层，它的 **keep-prob** 值应该相对较低，假设是 0.5。对于其它层，过拟合的程度可能没那么严重，它们的 **keep-prob** 值可能高一些，可能是 0.7，这里是 0.7。如果在某一层，我们不必担心其过拟合的问题，那么 **keep-prob** 可以为 1，为了表达清除，我用紫色线笔把它们圈出来，每层 **keep-prob** 的值可能不同。

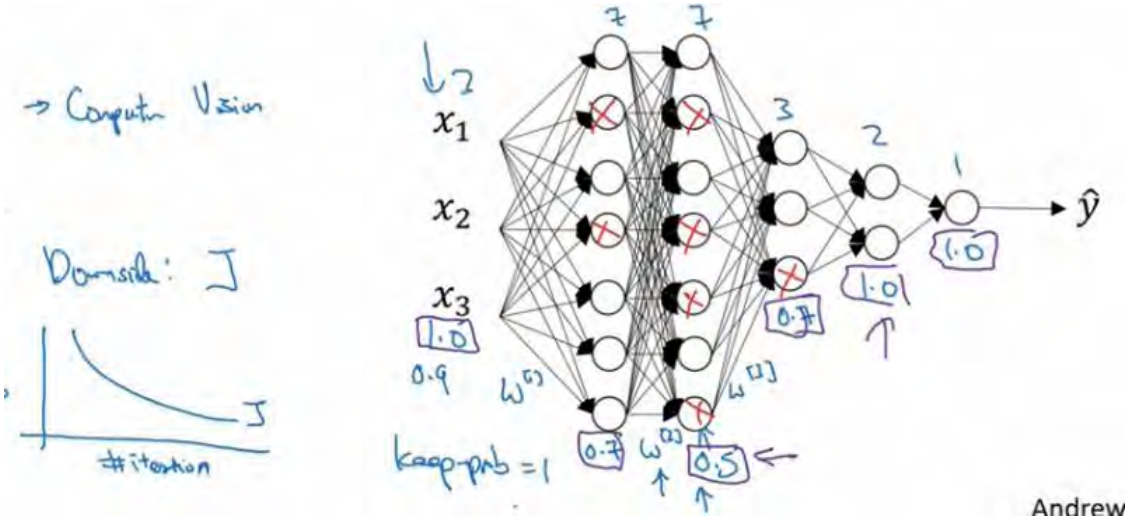


注意 **keep-prob** 的值是 1，意味着保留所有单元，并且不在这一层使用 **dropout**，对于有

可能出现过拟合，且含有诸多参数的层，我们可以把 **keep-prob** 设置成比较小的值，以便应用更强大的 **dropout**，有点像在处理$L2$正则化的正则化参数λ，我们尝试对某些层施行更多正则化，从技术上讲，我们也可以对输入层应用 **dropout**，我们有机会删除一个或多个输入特征，虽然现实中我们通常不这么做，**keep-prob** 的值为 1，是非常常用的输入值，也可以用更大的值，或许是 0.9。但是消除一半的输入特征是不太可能的，如果我们遵守这个准则，**keep-prob** 会接近于 1，即使你对输入层应用 **dropout**。

总结一下，如果你担心某些层比其它层更容易发生过拟合，可以把某些层的 **keep-prob** 值设置得比其它层更低，缺点是为了使用交叉验证，你要搜索更多的超级参数，另一种方案是在一些层上应用 **dropout**，而有些层不用 **dropout**，应用 **dropout** 的层只含有一个超级参数，就是 **keep-prob**。

结束前分享两个实施过程中的技巧，实施 **dropout**，在计算机视觉领域有很多成功的第一次。计算视觉中的输入量非常大，输入太多像素，以至于没有足够的数据，所以 **dropout** 在计算机视觉中应用得比较频繁，有些计算机视觉研究人员非常喜欢用它，几乎成了默认的选择，但要牢记一点，**dropout** 是一种正则化方法，它有助于预防过拟合，因此除非算法过拟合，不然我是不会使用 **dropout** 的，所以它在其它领域应用得比较少，主要存在于计算机视觉领域，因为我们通常没有足够的数据，所以一直存在过拟合，这就是有些计算机视觉研究人员如此钟情于 **dropout** 函数的原因。直观上我认为不能概括其它学科。

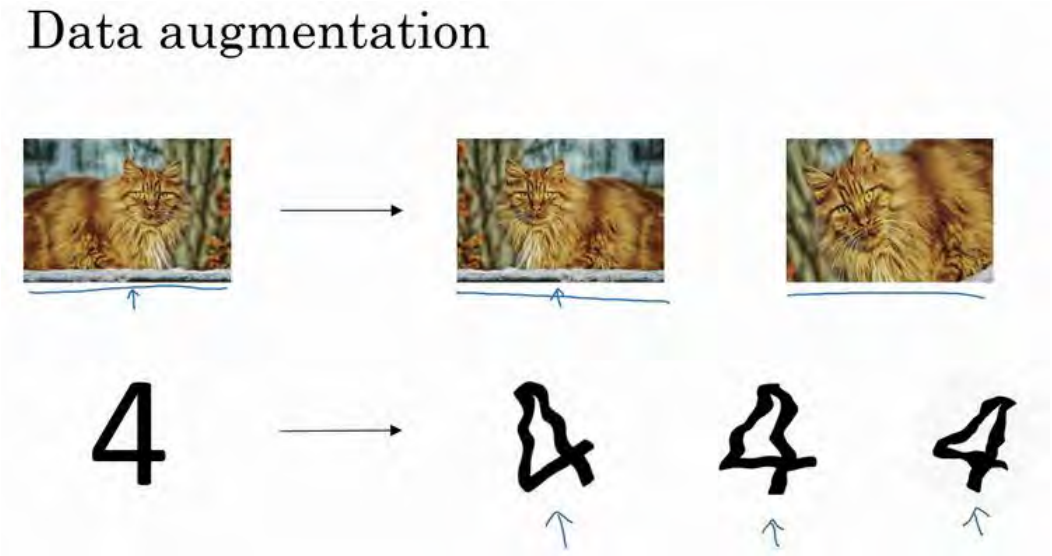


dropout 一大缺点就是代价函数 J 不再被明确定义，每次迭代，都会随机移除一些节点，如果再三检查梯度下降的性能，实际上是很难进行复查的。定义明确的代价函数 J 每次迭代后都会下降，因为我们所优化的代价函数 J 实际上并没有明确定义，或者说在某种程度上很难计算，所以我们失去了调试工具来绘制这样的图片。我通常会关闭 **dropout** 函数，将 **keep-**

prob 的值设为 1，运行代码，确保 J 函数单调递减。然后打开 **dropout** 函数，希望在 **dropout** 过程中，代码并未引入 bug。我觉得你也可以尝试其它方法，虽然我们并没有关于这些方法性能的数据统计，但你可以把它们与 **dropout** 方法一起使用。

1.8 其他正则化方法（Other regularization methods）

除了$L2$正则化和随机失活（**dropout**）正则化，还有几种方法可以减少神经网络中的过拟合:



一.数据扩增

假设你正在拟合猫咪图片分类器，如果你想通过扩增训练数据来解决过拟合，但扩增数据代价高，而且有时候我们无法扩增数据，但我们可以通过添加这类图片来增加训练集。例如，水平翻转图片，并把它添加到训练集。所以现在训练集中有原图，还有翻转后的这张图片，所以通过水平翻转图片，训练集则可以增大一倍，因为训练集有冗余，这虽然不如我们额外收集一组新图片那么好，但这样做节省了获取更多猫咪图片的花费。

除了水平翻转图片，你也可以随意裁剪图片，这张图是把原图旋转并随意放大后裁剪的，仍能辨别出图片中的猫咪。

通过随意翻转和裁剪图片，我们可以增大数据集，额外生成假训练数据。和全新的，独立的猫咪图片数据相比，这些额外的假的数据无法包含像全新数据那么多的信息，但我们这么做基本没有花费，代价几乎为零，除了一些对抗性代价。以这种方式扩增算法数据，进而

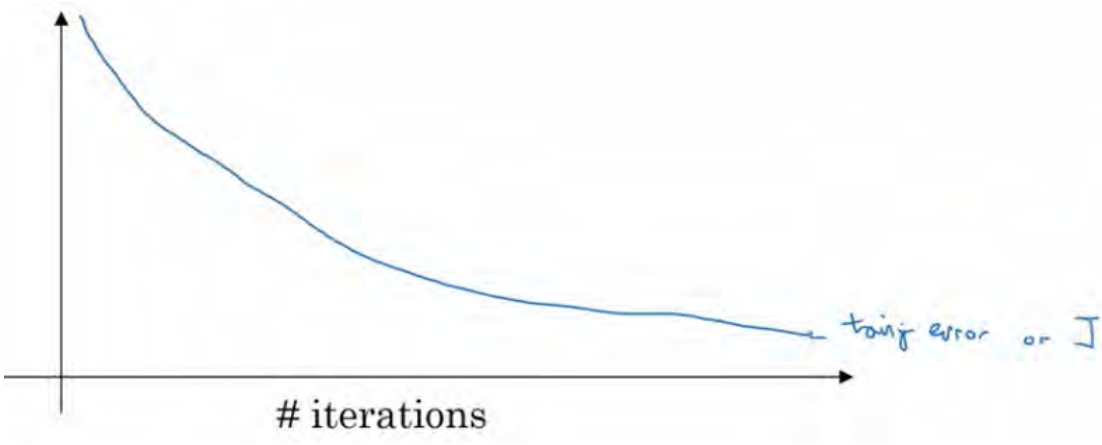
正则化数据集，减少过拟合比较廉价。

像这样人工合成数据的话，我们要通过算法验证，图片中的猫经过水平翻转之后依然是猫。大家注意，我并没有垂直翻转，因为我们不想上下颠倒图片，也可以随机选取放大后的部分图片，猫可能还在上面。

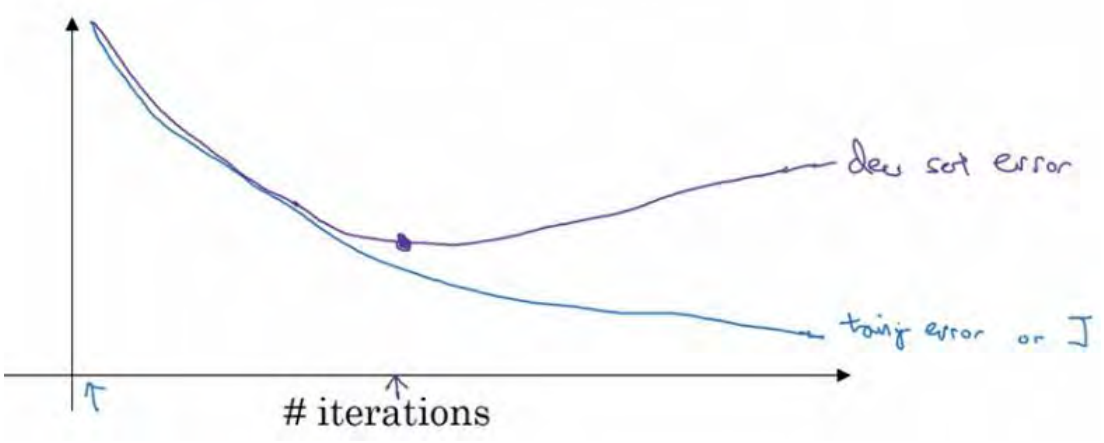
对于光学字符识别，我们还可以通过添加数字，随意旋转或扭曲数字来扩增数据，把这些数字添加到训练集，它们仍然是数字。为了方便说明，我对字符做了强变形处理，所以数字 4 看起来是波形的，其实不用对数字 4 做这么夸张的扭曲，只要轻微的变形就好，我做成这样是为了让大家看的更清楚。实际操作的时候，我们通常对字符做更轻微的变形处理。因为这几个 4 看起来有点扭曲。所以，数据扩增可作为正则化方法使用，实际功能上也与正则化相似。

二.**early stopping**

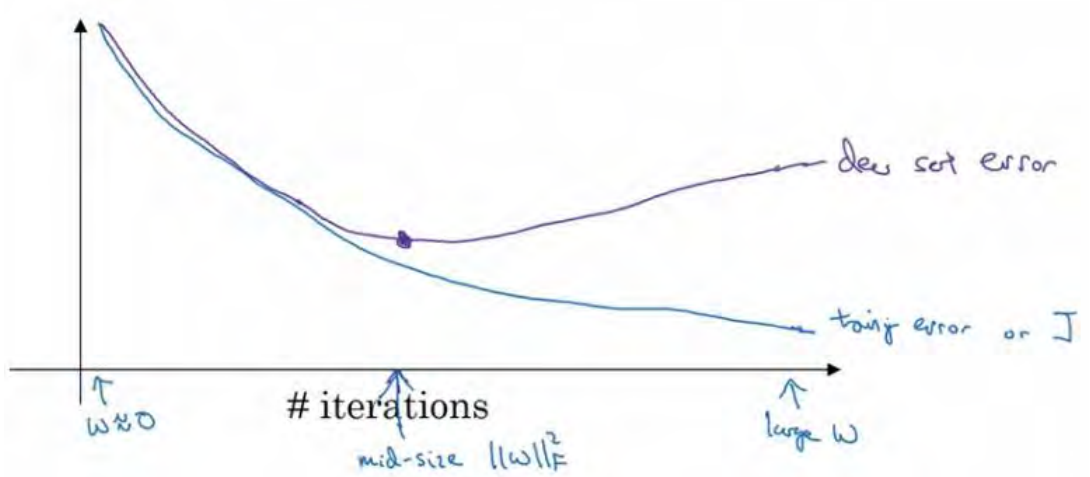
还有另外一种常用的方法叫作 **early stopping**，运行梯度下降时，我们可以绘制训练误差，或只绘制代价函数 J 的优化过程，在训练集上用 0-1 记录分类误差次数。呈单调下降趋势，如图。



因为在训练过程中，我们希望训练误差，代价函数 J 都在下降，通过 **early stopping**，我们不但可以绘制上面这些内容，还可以绘制验证集误差，它可以是验证集上的分类误差，或验证集上的代价函数，逻辑损失和对数损失等，你会发现，验证集误差通常会先呈下降趋势，然后在某个节点处开始上升，**early stopping** 的作用是，你会说，神经网络已经在这个迭代过程中表现得很好了，我们在此停止训练吧，得到验证集误差，它是怎么发挥作用的？

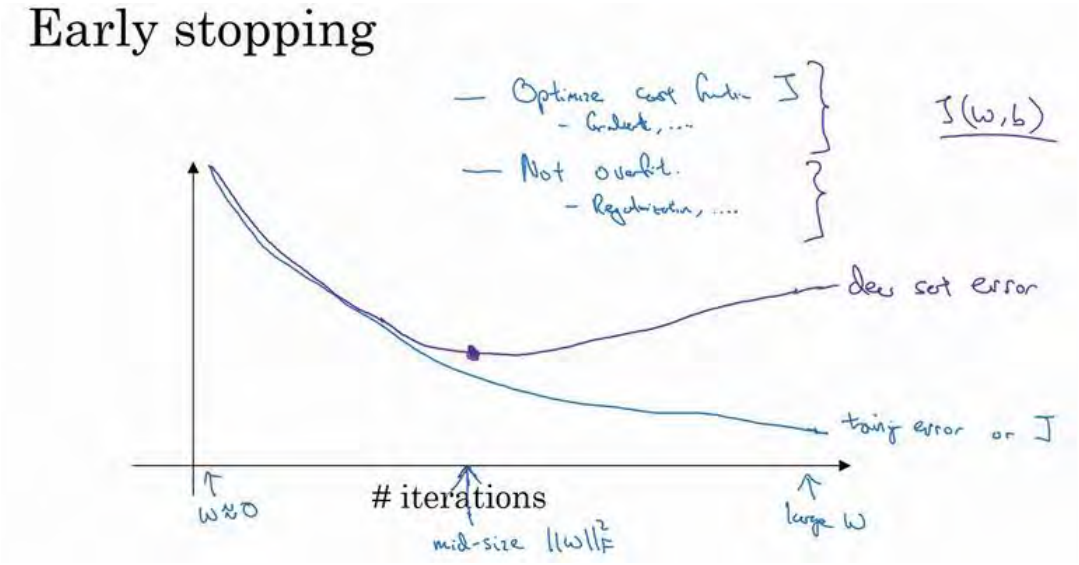


当你还未在神经网络上运行太多迭代过程的时候，参数w接近 0，因为随机初始化w值时，它的值可能都是较小的随机值，所以在你长期训练神经网络之前w依然很小，在迭代过程和训练过程中w的值会变得越来越大，比如在这儿，神经网络中参数w的值已经非常大了，所以 **early stopping** 要做就是在中间点停止迭代过程，我们得到一个w值中等大小的弗罗贝尼乌斯范数，与$L2$正则化相似，选择参数 w 范数较小的神经网络，但愿你的神经网络过度拟合不严重。



术语 **early stopping** 代表提早停止训练神经网络，训练神经网络时，我有时会用到 **early stopping**，但是它也有一个缺点，我们来了解一下。

我认为机器学习过程包括几个步骤，其中一步是选择一个算法来优化代价函数J，我们有很多种工具来解决这个问题，如梯度下降，后面我会介绍其它算法，例如 **Momentum**，**RMSprop** 和 **Adam** 等等，但是优化代价函数J之后，我也不想发生过拟合，也有一些工具可以解决该问题，比如正则化，扩增数据等等。



在机器学习中，超级参数激增，选出可行的算法也变得越来越复杂。我发现，如果我们用一组工具优化代价函数J，机器学习就会变得更简单，在重点优化代价函数J时，你只需要留意w和b，$J(w,b)$的值越小越好，你只需要想办法减小这个值，其它的不用关注。然后，预防过拟合还有其他任务，换句话说就是减少方差，这一步我们用另外一套工具来实现，这个原理有时被称为“正交化”。思路就是在一个时间做一个任务，后面课上我会具体介绍正交化，如果你还不了解这个概念，不用担心。

但对我来说 **early stopping** 的主要缺点就是你不能独立地处理这两个问题，因为提早停止梯度下降，也就是停止了优化代价函数J，因为现在你不再尝试降低代价函数 J，所以代价函数J的值可能不够小，同时你又希望不出现过拟合，你没有采取不同的方式来解决这两个问题，而是用一种方法同时解决两个问题，这样做的结果是我要考虑的东西变得更复杂。

如果不用 **early stopping**，另一种方法就是$L2$正则化，训练神经网络的时间就可能很长。我发现，这导致超级参数搜索空间更容易分解，也更容易搜索，但是缺点在于，你必须尝试很多正则化参数λ的值，这也导致搜索大量λ值的计算代价太高。

Early stopping 的优点是，只运行一次梯度下降，你可以找出w的较小值，中间值和较大值，而无需尝试$L2$正则化超级参数λ的很多值。

如果你还不能完全理解这个概念，没关系，下节课我们会详细讲解正交化，这样会更好理解。

虽然$L2$正则化有缺点，可还是有很多人愿意用它。吴恩达老师个人更倾向于使用$L2$正则化，尝试许多不同的λ值，假设你可以负担大量计算的代价。而使用 **early stopping** 也能得到相似结果，还不用尝试这么多λ值。

这节课我们讲了如何使用数据扩增，以及如何使用 **early stopping** 降低神经网络中的方差或预防过拟合。

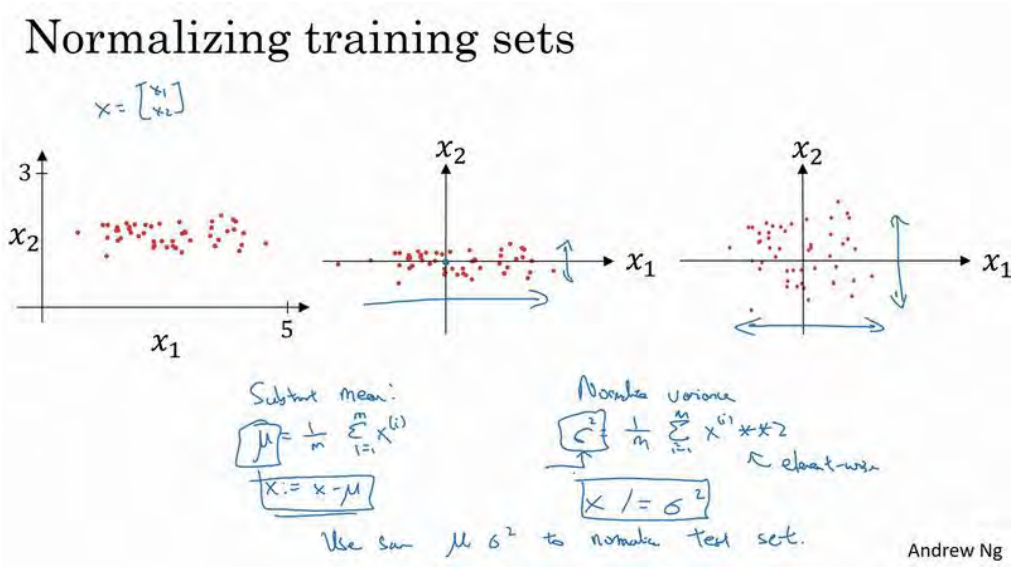
1.9 归一化输入（Normalizing inputs）

训练神经网络，其中一个加速训练的方法就是归一化输入。假设一个训练集有两个特征，输入特征为 2 维，归一化需要两个步骤：

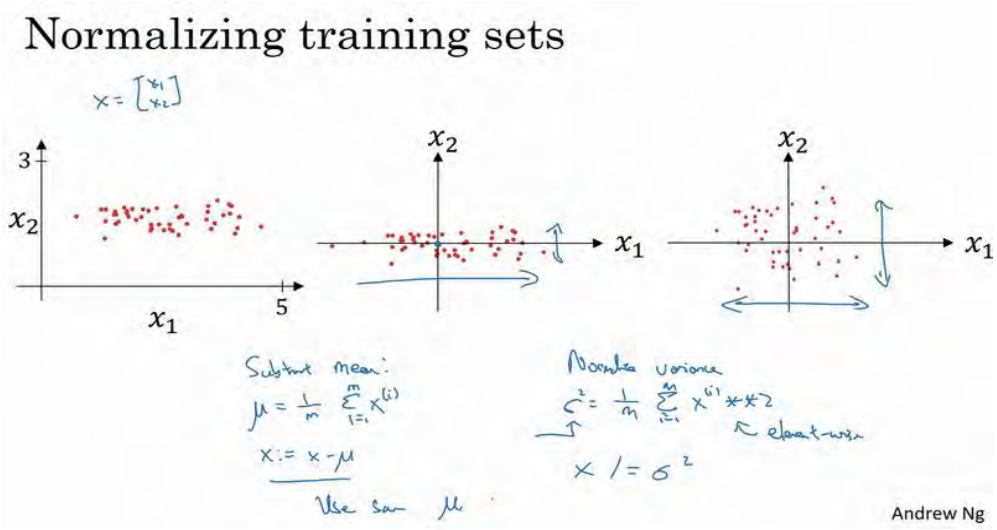
零均值

归一化方差；

我们希望无论是训练集和测试集都是通过相同的μ和σ^2定义的数据转换，这两个是由训练集得出来的。



第一步是零均值化，$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$，它是一个向量，$x$等于每个训练数据 x减去μ，意思是移动训练集，直到它完成零均值化。



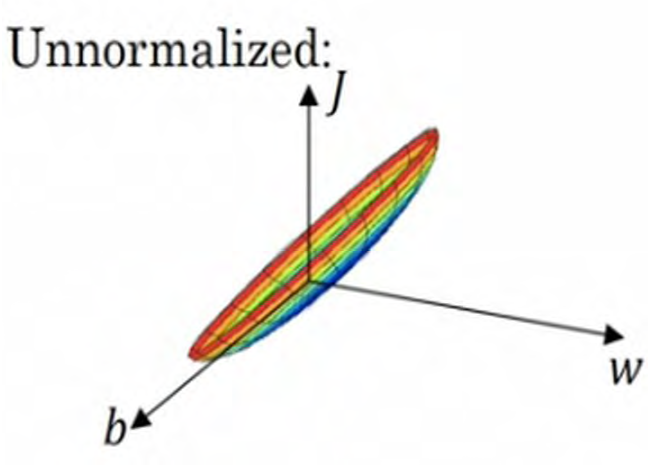
第二步是归一化方差，注意特征x_1的方差比特征x_2的方差要大得多，我们要做的是给σ赋值，$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^2$，这是节点$y$ 的平方，σ^2是一个向量，它的每个特征都有方差，注意，我们已经完成零值均化，$(x^{(i)})^2$元素y^2就是方差，我们把所有数据除以向量σ^2，最后变成上图形式。

x_1和x_2的方差都等于 1。提示一下，如果你用它来调整训练数据，那么用相同的 μ 和 σ^2来归一化测试集。尤其是，你不希望训练集和测试集的归一化有所不同，不论μ的值是什么，也不论σ^2的值是什么，这两个公式中都会用到它们。所以你要用同样的方法调整测试集，而不是在训练集和测试集上分别预估μ 和 σ^2。因为我们希望不论是训练数据还是测试数据，都是通过相同 μ 和σ^2定义的相同数据转换，其中μ和σ^2是由训练集数据计算得来的。

我们为什么要这么做呢？为什么我们想要归一化输入特征，回想一下右上角所定义的代价函数。

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

如果你使用非归一化的输入特征，代价函数会像这样：

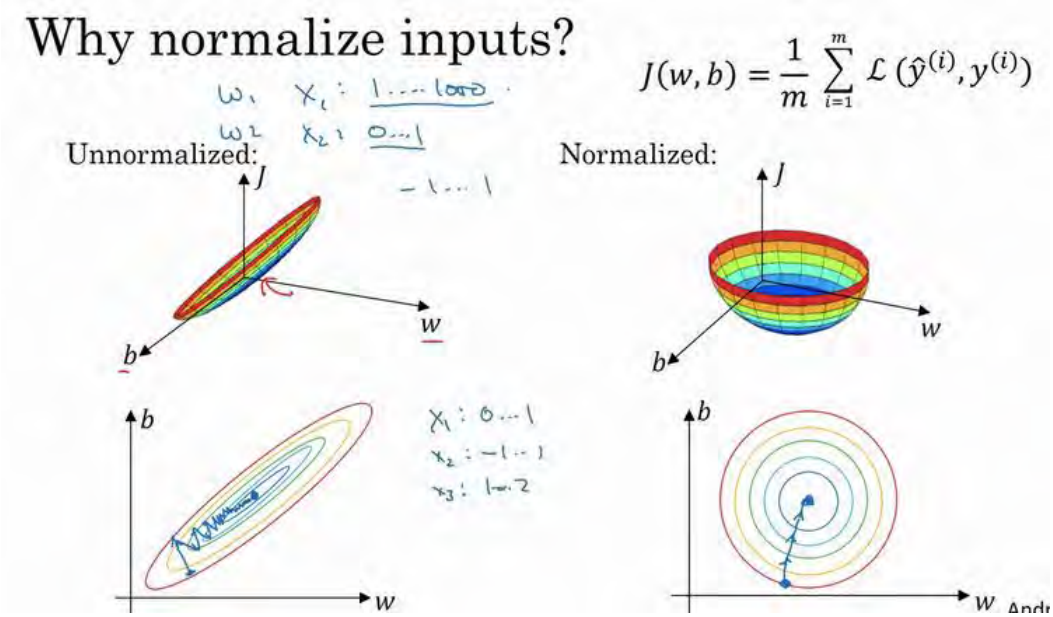


这是一个非常细长狭窄的代价函数，你要找的最小值应该在这里。但如果特征值在不同范围，假如x_1取值范围从 1 到 1000，特征x_2的取值范围从 0 到 1，结果是参数w_1和w_2值的范围或比率将会非常不同，这些数据轴应该是w_1和w_2，但直观理解，我标记为w和b，代价函数就有点像狭长的碗一样，如果你能画出该函数的部分轮廓，它会是这样一个狭长的函数。

然而如果你归一化特征，代价函数平均起来看更对称，如果你在上图这样的代价函数上运行梯度下降法，你必须使用一个非常小的学习率。因为如果是在这个位置，梯度下降法可能需要多次迭代过程，直到最后找到最小值。但如果函数是一个更圆的球形轮廓，那么不论

从哪个位置开始，梯度下降法都能够更直接地找到最小值，你可以在梯度下降法中使用较大步长，而不需要像在左图中那样反复执行。

当然，实际上w是一个高维向量，因此用二维绘制w并不能正确地传达并直观理解，但总地直观理解是代价函数会更圆一些，而且更容易优化，前提是特征都在相似范围内，而不是从 1 到 1000，0 到 1 的范围，而是在-1 到 1 范围内或相似偏差，这使得代价函数J优化起来更简单快速。



实际上如果假设特征x_1范围在 0-1 之间，x_2的范围在-1 到 1 之间，x_3范围在 1-2 之间，它们是相似范围，所以会表现得很好。

当它们在非常不同的取值范围内，如其中一个从 1 到 1000，另一个从 0 到 1，这对优化算法非常不利。但是仅将它们设置为均化零值，假设方差为 1，就像上一张幻灯片里设定的那样，确保所有特征都在相似范围内，通常可以帮助学习算法运行得更快。

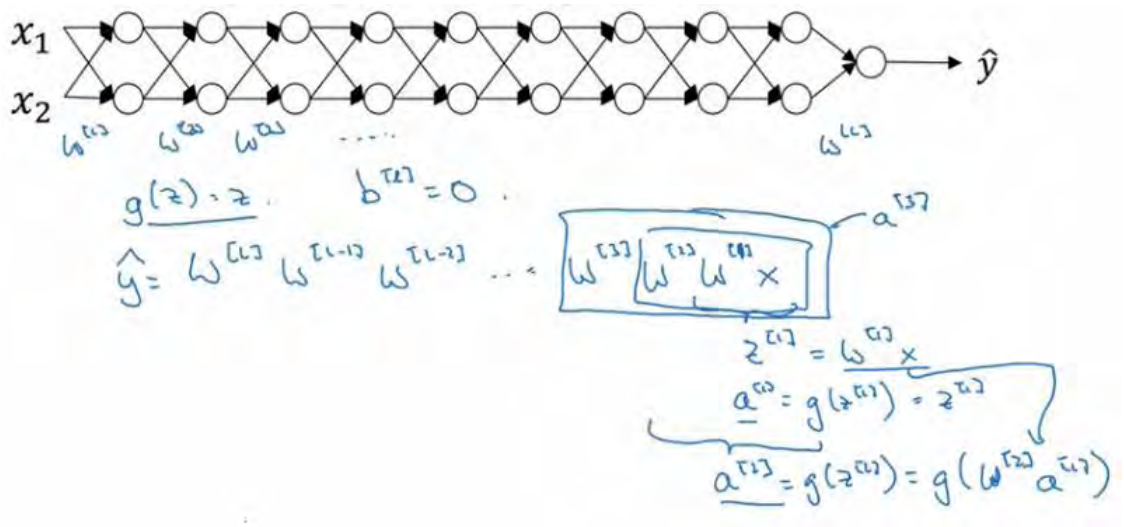
所以如果输入特征处于不同范围内，可能有些特征值从 0 到 1，有些从 1 到 1000，那么归一化特征值就非常重要了。如果特征值处于相似范围内，那么归一化就不是很重要了。执行这类归一化并不会产生什么危害，我通常会做归一化处理，虽然我不确定它能否提高训练或算法速度。

这就是归一化特征输入，下节课我们将继续讨论提升神经网络训练速度的方法。

1.10 梯度消失/梯度爆炸（Vanishing / Exploding gradients）

训练神经网络，尤其是深度神经所面临的一个问题就是梯度消失或梯度爆炸，也就是你训练神经网络的时候，导数或坡度有时会变得非常大，或者非常小，甚至于以指数方式变小，这加大了训练的难度。

这节课，你将会了解梯度消失或梯度爆炸的真正含义，以及如何更明智地选择随机初始化权重，从而避免这个问题。 假设你正在训练这样一个极深的神经网络，为了节约幻灯片上的空间，我画的神经网络每层只有两个隐藏单元，但它可能含有更多，但这个神经网络会有参数$W^{[1]}$，$W^{[2]}$，$W^{[3]}$等等，直到$W^{[l]}$，为了简单起见，假设我们使用激活函数$g(z)=z$，也就是线性激活函数，我们忽略b，假设$b^{[l]}$=0，如果那样的话，输出$y=W^{[l]}W^{[L-1]}W^{[L-2]}\ldots W^{[3]}W^{[2]}W^{[1]}x$，如果你想考验我的数学水平，$W^{[1]}x=z^{[1]}$，因为$b=0$，所以我想$z^{[1]}=W^{[1]}x$，$a^{[1]}=g(z^{[1]})$，因为我们使用了一个线性激活函数，它等于$z^{[1]}$，所以第一项$W^{[1]}x=a^{[1]}$，通过推理，你会得出$W^{[2]}W^{[1]}x=a^{[2]}$，因为$a^{[2]}=g(z^{[2]})$，还等于$g(W^{[2]}a^{[1]})$，可以用$W^{[1]}x$替换$a^{[1]}$，所以这一项就等于$a^{[2]}$，这个就是$a^{[3]}(W^{[3]}W^{[2]}W^{[1]}x)$。

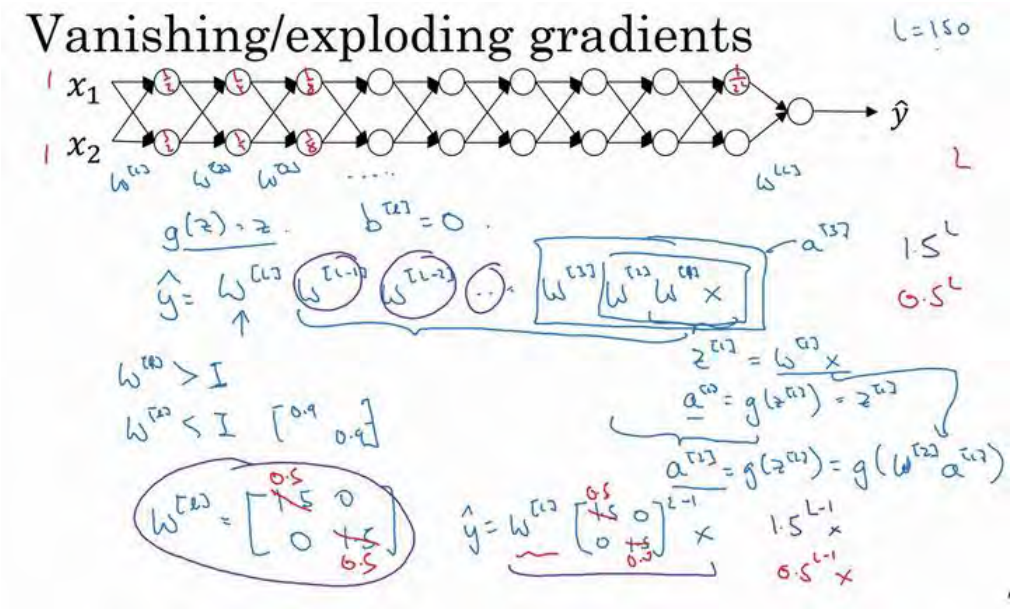


所有这些矩阵数据传递的协议将给出 ŷ 而不是y的值。

假设每个权重矩阵$W^{[l]}=\begin{bmatrix}1.5 & 0\\ 0 & 1.5\end{bmatrix}$，从技术上来讲，最后一项有不同维度，可能它就是余下的权重矩阵，$y=W^{[1]}\begin{bmatrix}1.5 & 0\\ 0 & 1.5\end{bmatrix}^{(L-1)}x$，因为我们假设所有矩阵都等于它，它是 1.5 倍的单位矩阵，最后的计算结果就是$\hat{y}$，$\hat{y}$也就是等于$1.5^{(L-1)}x$。如果对于一个深度神经网络来说 L 值较大，那么$\hat{y}$的值也会非常大，实际上它呈指数级增长的，它增长的比率是1.5^{L}，因此对于一个深度神经网络，y的值将爆炸式增长。

相反的，如果权重是 0.5，$W^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$，它比 1 小，这项也就变成了$0.5^L$，矩阵$y = W^{[1]}\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{(L-1)} x$，再次忽略$W^{[L]}$，因此每个矩阵都小于 1，假设$x_1$和$x_2$都是 1，激活函数将变成$\frac{1}{2}$，$\frac{1}{2}$，$\frac{1}{4}$，$\frac{1}{4}$，$\frac{1}{8}$，$\frac{1}{8}$等，直到最后一项变成$\frac{1}{2^L}$，所以作为自定义函数，激活函数的值将以指数级下降，它是与网络层数数量L相关的函数，在深度网络中，激活函数以指数级递减。

我希望你得到的直观理解是，权重W只比 1 略大一点，或者说只是比单位矩阵大一点，深度神经网络的激活函数将爆炸式增长，如果W比 1 略小一点，可能是$\begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix}$。



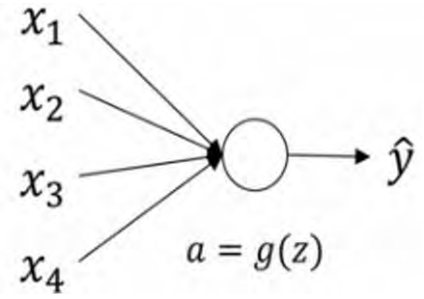
在深度神经网络中，激活函数将以指数级递减，虽然我只是讨论了激活函数以与L相关的指数级数增长或下降，它也适用于与层数 L 相关的导数或梯度函数，也是呈指数级增长或呈指数递减。

对于当前的神经网络，假设$L = 150$，最近 **Microsoft** 对 152 层神经网络的研究取得了很大进展，在这样一个深度神经网络中，如果激活函数或梯度函数以与L相关的指数增长或递减，它们的值将会变得极大或极小，从而导致训练难度上升，尤其是梯度指数小于L时，梯度下降算法的步长会非常非常小，梯度下降算法将花费很长时间来学习。

总结一下，我们讲了深度神经网络是如何产生梯度消失或爆炸问题的，实际上，在很长一段时间内，它曾是训练深度神经网络的阻力，虽然有一个不能彻底解决此问题的解决方案，但是已在如何选择初始化权重问题上提供了很多帮助。

1.11 神经网络的权重初始化（Weight Initialization for Deep NetworksVanishing / Exploding gradients）

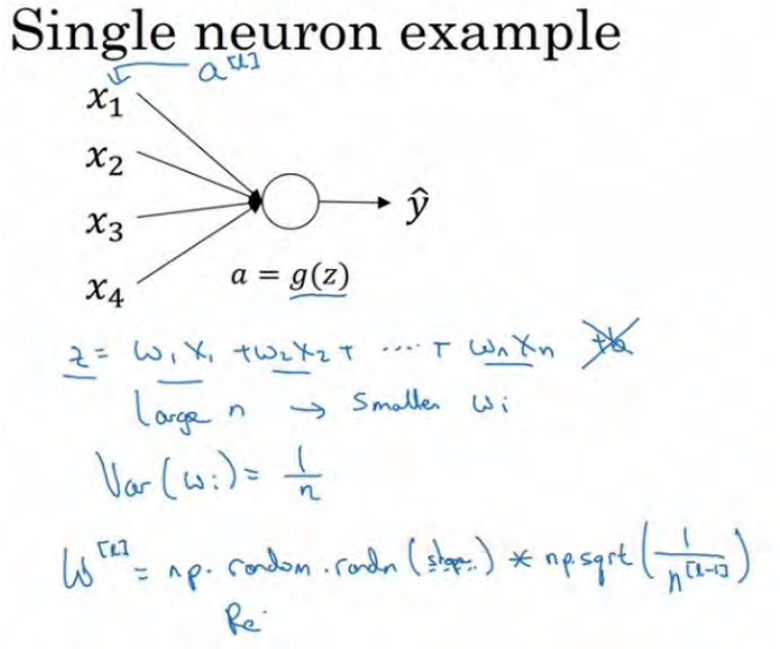
上节课，我们学习了深度神经网络如何产生梯度消失和梯度爆炸问题，最终针对该问题，我们想出了一个不完整的解决方案，虽然不能彻底解决问题，却很有用，有助于我们为神经网络更谨慎地选择随机初始化参数，为了更好地理解它，我们先举一个神经单元初始化地例子，然后再演变到整个深度网络。



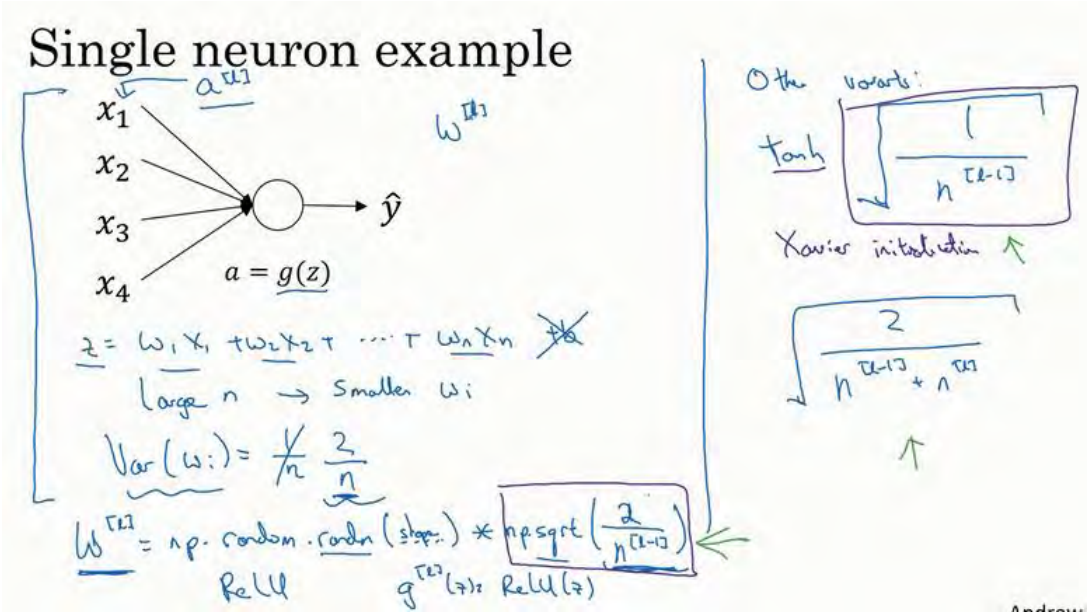
我们来看看只有一个神经元的情况，然后才是深度网络。

单个神经元可能有 4 个输入特征，从x_1到x_4，经过$a = g(z)$处理，最终得到$\hat{y}$，稍后讲深度网络时，这些输入表示为$a^{[l]}$，暂时我们用x表示。

$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n$，$b = 0$，暂时忽略$b$，为了预防$z$值过大或过小，你可以看到$n$越大，你希望$w_i$越小，因为$z$是$w_ix_i$的和，如果你把很多此类项相加，希望每项值更小，最合理的方法就是设置$w_i = \frac{1}{n}$，n表示神经元的输入特征数量，实际上，你要做的就是设置某层权重矩阵$w^{[l]} = np.random.randn(\text{shape}) * \text{np.sqrt}(\frac{1}{n^{[l-1]}})$，$n^{[l-1]}$就是我喂给第$l$层神经单元的数量（即第$l-1$层神经元数量）。



结果，如果你是用的是 **Relu** 激活函数，而不是$\frac{1}{n}$，方差设置为$\frac{2}{n}$，效果会更好。你常常发现，初始化时，尤其是使用 **Relu** 激活函数时，$g^{[l]}(z) = Relu(z)$,它取决于你对随机变量的熟悉程度，这是高斯随机变量，然后乘以它的平方根，也就是引用这个方差$\frac{2}{n}$。这里，我用的是$n^{[l-1]}$，因为本例中，逻辑回归的特征是不变的。但一般情况下l层上的每个神经元都有$n^{[l-1]}$个输入。如果激活函数的输入特征被零均值和标准方差化，方差是 1，z也会调整到相似范围，这就没解决问题（梯度消失和爆炸问题）。但它确实降低了梯度消失和爆炸问题，因为它给权重矩阵w设置了合理值，你也知道，它不能比 1 大很多，也不能比 1 小很多，所以梯度没有爆炸或消失过快。



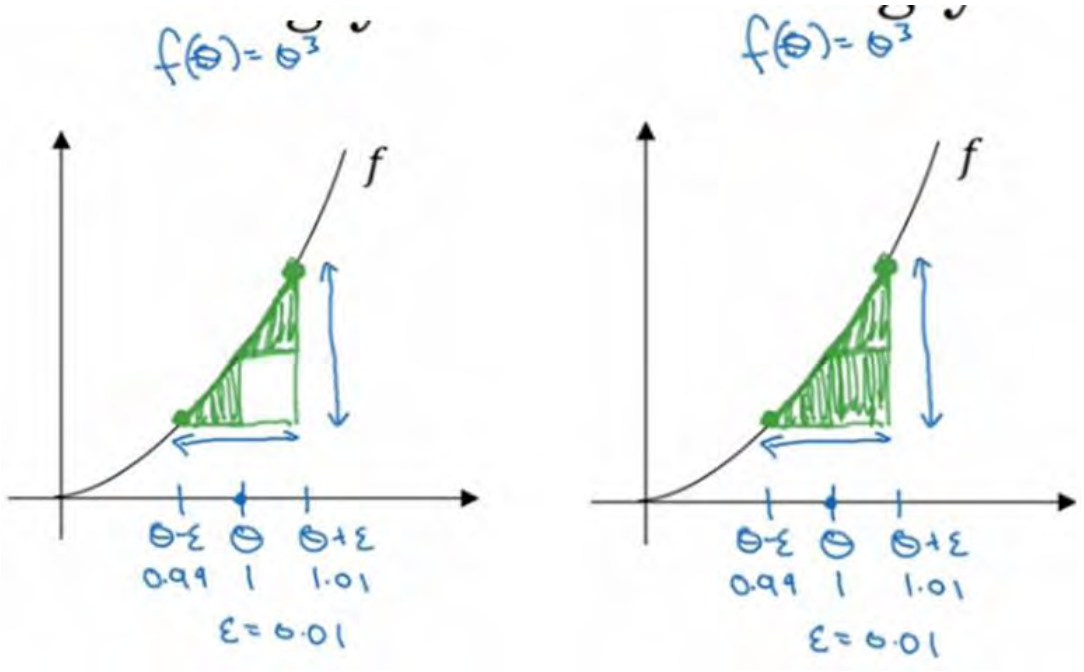
我提到了其它变体函数，刚刚提到的函数是 **Relu** 激活函数，一篇由 **Herd** 等人撰写的论文曾介绍过。对于几个其它变体函数，如 **tanh** 激活函数，有篇论文提到，常量 1 比常量 2 的效率更高，对于 **tanh** 函数来说，它是$\sqrt{\frac{1}{n^{[l-1]}}}$，这里平方根的作用与这个公式作用相同($\text{np.sqrt}(\frac{1}{n^{[l-1]}})$)，它适用于 **tanh** 激活函数，被称为 **Xavier** 初始化。**Yoshua Bengio** 和他的同事还提出另一种方法，你可能在一些论文中看到过，它们使用的是公式$\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$。其它理论已对此证明，但如果你想用 **Relu** 激活函数，也就是最常用的激活函数，我会用这个公式$\text{np.sqrt}(\frac{2}{n^{[l-1]}})$，如果使用 **tanh** 函数，可以用公式$\sqrt{\frac{1}{n^{[l-1]}}}$，有些作者也会使用这个函数。

实际上，我认为所有这些公式只是给你一个起点，它们给出初始化权重矩阵的方差的默认值，如果你想添加方差，方差参数则是另一个你需要调整的超级参数，可以给公式$\text{np.sqrt}(\frac{2}{n^{[l-1]}})$添加一个乘数参数，调优作为超级参数激增一份子的乘子参数。有时调优该超级参数效果一般，这并不是我想调优的首要超级参数，但我发现调优过程中产生的问题，虽然调优该参数能起到一定作用，但考虑到相比调优，其它超级参数的重要性，我通常把它的优先级放得比较低。

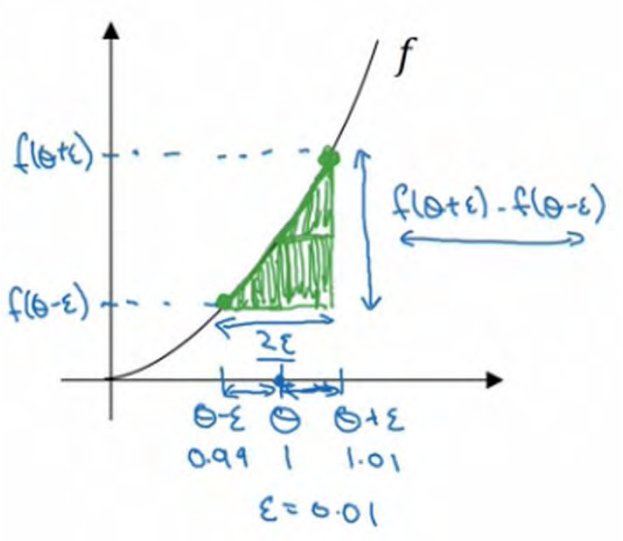
希望你现在对梯度消失或爆炸问题以及如何为权重初始化合理值已经有了一个直观认识，希望你设置的权重矩阵既不会增长过快，也不会太快下降到 0，从而训练出一个权重或梯度不会增长或消失过快的深度网络。我们在训练深度网络时，这也是一个加快训练速度的技巧。

1.12 梯度的数值逼近（Numerical approximation of gradients）

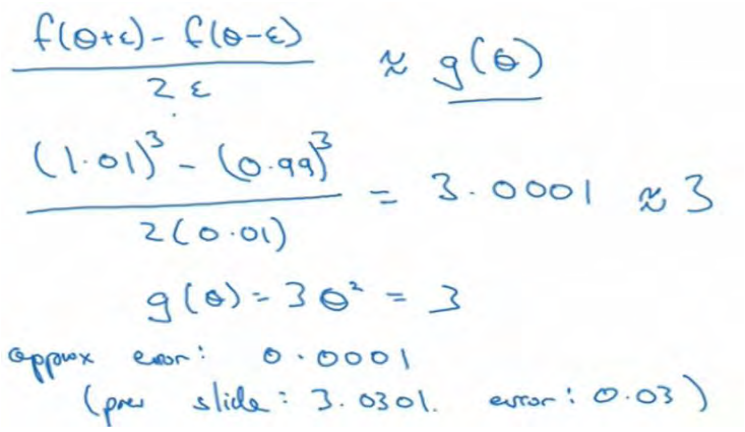
在实施 **backprop** 时，有一个测试叫做梯度检验，它的作用是确保 **backprop** 正确实施。因为有时候，你虽然写下了这些方程式，却不能 100%确定，执行 **backprop** 的所有细节都是正确的。为了逐渐实现梯度检验，我们首先说说如何计算梯度的数值逼近，下节课，我们将讨论如何在 **backprop** 中执行梯度检验，以确保 **backprop** 正确实施。



我们先画出函数f，标记为$f(\theta)$，$f(\theta)=\theta^3$，先看一下θ的值，假设$\theta=1$，不增大θ的值，而是在θ 右侧，设置一个$\theta+\varepsilon$，在θ左侧，设置$\theta-\varepsilon$。因此$\theta=1$，$\theta+\varepsilon=1.01, \theta-\varepsilon=0.99$,，跟以前一样，$\varepsilon$的值为 0.01，看下这个小三角形，计算高和宽的比值，就是更准确的梯度预估，选择f函数在$\theta-\varepsilon$上的这个点，用这个较大三角形的高比上宽，技术上的原因我就不详细解释了，较大三角形的高宽比值更接近于θ的导数，把右上角的三角形下移，好像有了两个三角形，右上角有一个，左下角有一个，我们通过这个绿色大三角形同时考虑了这两个小三角形。所以我们得到的不是一个单边公差而是一个双边公差。



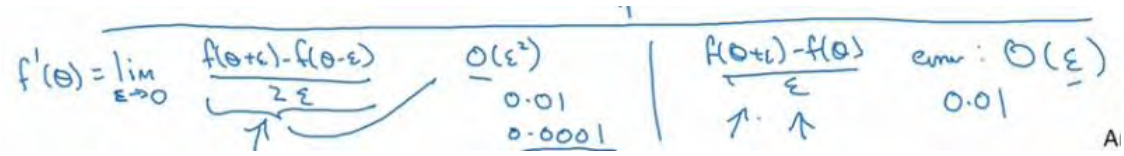
我们写一下数据算式，图中绿色三角形上边的点的值是$f(\theta+\varepsilon)$，下边的点是$f(\theta-\varepsilon)$，这个三角形的高度是$f(\theta+\varepsilon)-f(\theta-\varepsilon)$，这两个宽度都是 ε，所以三角形的宽度是$2\varepsilon$，高宽比值为$\frac{f(\theta+\varepsilon)-(\theta-\varepsilon)}{2\varepsilon}$，它的期望值接近$g(\theta)$，$f(\theta)=\theta^3$传入参数值，$\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon}=\frac{(1.01)^3-(0.99)^3}{2\times0.01}$，大家可以暂停视频，用计算器算算结果，结果应该是 3.0001，而前面一张幻灯片上面是，当$\theta=1$时，$g(\theta)=3\theta^2=3$，所以这两个$g(\theta)$值非常接近，逼近误差为 0.0001，前一张幻灯片，我们只考虑了单边公差，即从θ到$\theta+\varepsilon$之间的误差，$g(\theta)$的值为 3.0301，逼近误差是 0.03，不是 0.0001，所以使用双边误差的方法更逼近导数，其结果接近于 3，现在我们更加确信，$g(\theta)$可能是f导数的正确实现，在梯度检验和反向传播中使用该方法时，最终，它与运行两次单边公差的速度一样，实际上，我认为这种方法还是非常值得使用的，因为它的结果更准确。



这是一些你可能比较熟悉的微积分的理论，如果你不太明白我讲的这些理论也没关系，导数的官方定义是针对值很小的ε，导数的官方定义是$f'(\theta)=\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon}$，如果你上过微

积分课，应该学过无穷尽的定义，我就不在这里讲了。

对于一个非零的ε，它的逼近误差可以写成$O(\varepsilon^2)$，ε 值非常小，如果$\varepsilon = 0.01$，$\varepsilon^2 = 0.0001$，大写符号O的含义是指逼近误差其实是一些常量乘以ε^2，但它的确是很准确的逼近误差，所以大写O的常量有时是 1。然而，如果我们用另外一个公式逼近误差就是$O(\varepsilon)$，当ε小于 1 时，实际上ε比ε^2大很多，所以这个公式近似值远没有左边公式的准确，所以在执行梯度检验时，我们使用双边误差，即$\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon}$，而不使用单边公差，因为它不够准确。



如果你不理解上面两条结论，所有公式都在这儿，不用担心，如果你对微积分和数值逼近有所了解，这些信息已经足够多了，重点是要记住，双边误差公式的结果更准确，下节课我们做梯度检验时就会用到这个方法。

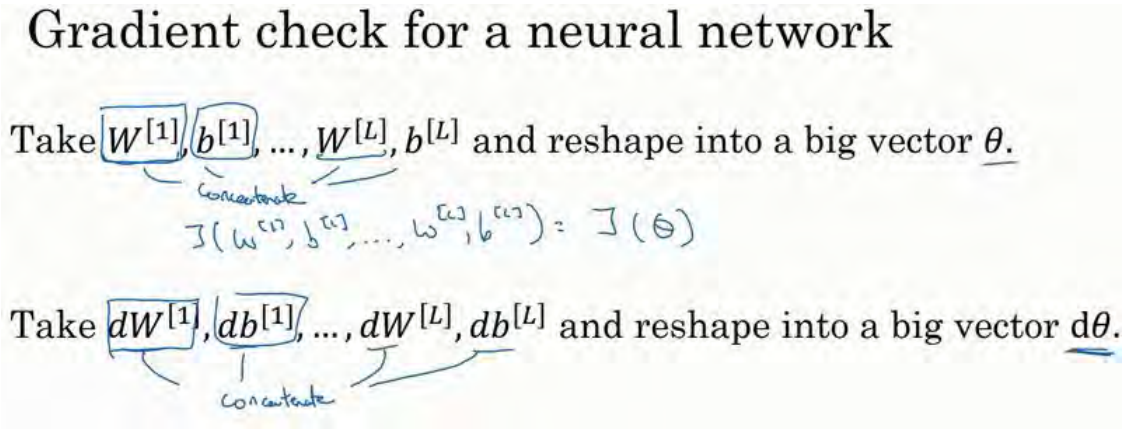
今天我们讲了如何使用双边误差来判断别人给你的函数$g(\theta)$，是否正确实现了函数f的偏导，现在我们可以使用这个方法来检验反向传播是否得以正确实施，如果不正确，它可能有 bug 需要你来解决。

1.13 梯度检验（Gradient checking）

梯度检验帮我们节省了很多时间，也多次帮我发现 **backprop** 实施过程中的 bug，接下来，我们看看如何利用它来调试或检验 **backprop** 的实施是否正确。

假设你的网络中含有下列参数，$W^{[1]}$和$b^{[1]}$……$W^{[l]}$和$b^{[l]}$，为了执行梯度检验，首先要做的就是，把所有参数转换成一个巨大的向量数据，你要做的就是把矩阵W转换成一个向量，把所有 W 矩阵转换成向量之后，做连接运算，得到一个巨型向量θ，该向量表示为参数θ，代价函数J是所有W和b的函数，现在你得到了一个θ的代价函数J（即$J(\theta)$）。接着，你得到与W和b顺序相同的数据，你同样可以把$dW^{[1]}$和$db^{[1]}$……$dW^{[l]}$和$db^{[l]}$转换成一个新的向量，用它们来初始化大向量$d\theta$，它与θ具有相同维度。

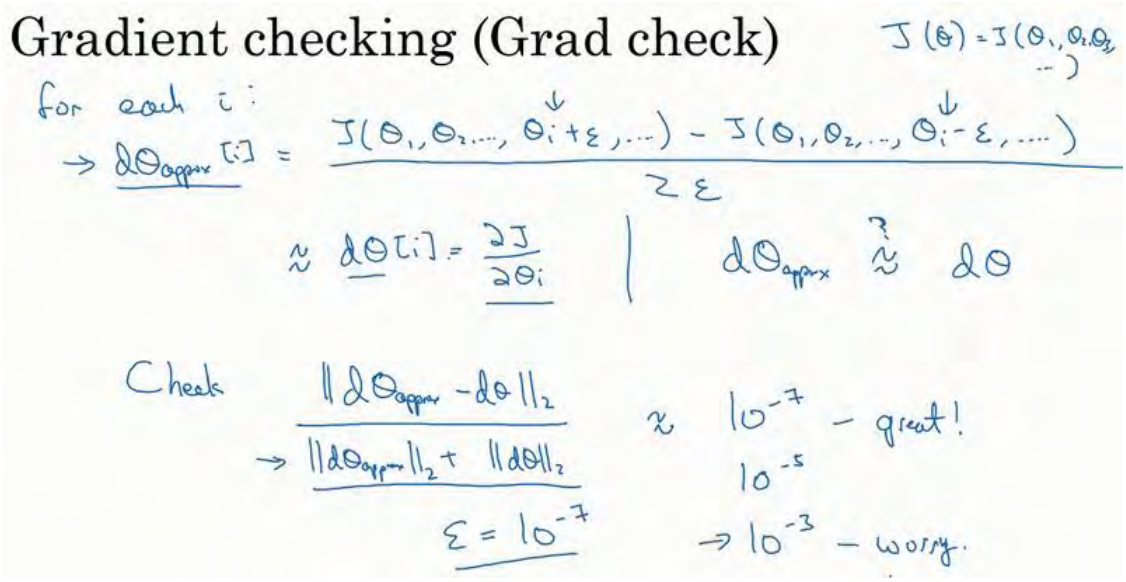
同样的，把$dW^{[1]}$转换成矩阵，$db^{[1]}$已经是一个向量了，直到把$dW^{[l]}$转换成矩阵，这样所有的dW都已经是矩阵，注意$dW^{[1]}$与$W^{[1]}$具有相同维度，$db^{[1]}$与$b^{[1]}$具有相同维度。经过相同的转换和连接运算操作之后，你可以把所有导数转换成一个大向量$d\theta$，它与θ具有相同维度，现在的问题是$d\theta$和代价函数J的梯度或坡度有什么关系？



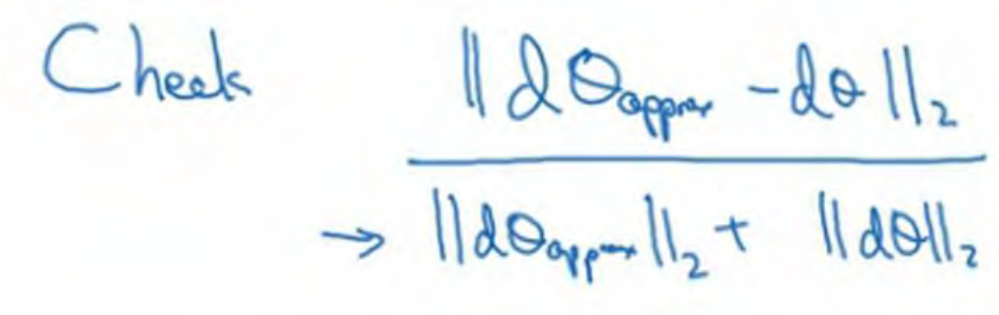
这就是实施梯度检验的过程，英语里通常简称为“**grad check**”，首先，我们要清楚J是超参数θ的一个函数，你也可以将 J 函数展开为$J(\theta_1, \theta_2, \theta_3, \dots\dots)$，不论超级参数向量$\theta$的维度是多少，为了实施梯度检验，你要做的就是循环执行，从而对每个i也就是对每个θ组成元素计算$d\theta_{\text{approx}}[i]$的值，我使用双边误差，也就是

$$d\theta_{\text{approx}}[i] = \frac{J(\theta_1, \theta_2, \dots \theta_i + \varepsilon, \dots) - J(\theta_1, \theta_2, \dots \theta_i - \varepsilon, \dots)}{2\varepsilon}$$

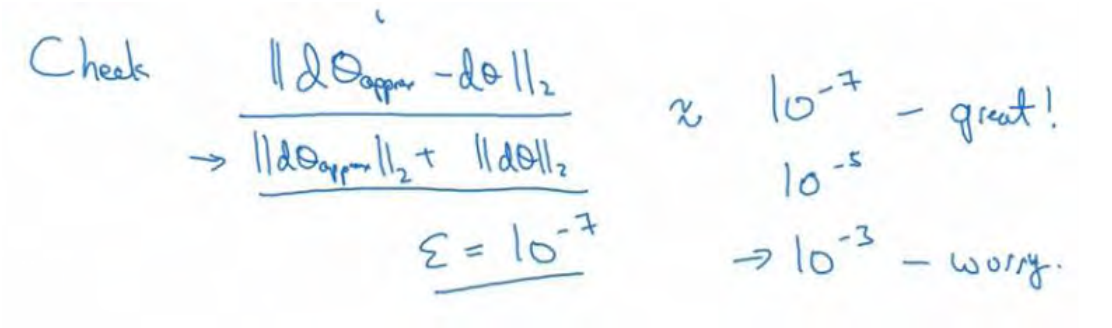
只对θ_i增加ε，其它项保持不变，因为我们使用的是双边误差，对另一边做同样的操作，只不过是减去ε，θ其它项全都保持不变。



从上节课中我们了解到这个值（$d\theta_{\text{approx}}[i]$）应该逼近$d\theta[i]$=$\frac{\partial J}{\partial \theta_i}$，$d\theta[i]$是代价函数的偏导数，然后你需要对 i 的每个值都执行这个运算，最后得到两个向量，得到$d\theta$的逼近值$d\theta_{\text{approx}}$，它与$d\theta$具有相同维度，它们两个与θ具有相同维度，你要做的就是验证这些向量是否彼此接近。



具体来说，如何定义两个向量是否真的接近彼此？我一般做下列运算，计算这两个向量的距离，$d\theta_{\text{approx}}[i]-d\theta[i]$的欧几里得范数，注意这里（$||d\theta_{\text{approx}}-d\theta||_2$）没有平方，它是误差平方之和，然后求平方根，得到欧式距离，然后用向量长度归一化，使用向量长度的欧几里得范数。分母只是用于预防这些向量太小或太大，分母使得这个方程式变成比率，我们实际执行这个方程式，ε可能为10^{-7}，使用这个取值范围内的ε，如果你发现计算方程式得到的值为10^{-7}或更小，这就很好，这就意味着导数逼近很有可能是正确的，它的值非常小。



如果它的值在10^{-5}范围内，我就要小心了，也许这个值没问题，但我会再次检查这个向量的所有项，确保没有一项误差过大，可能这里有 bug。

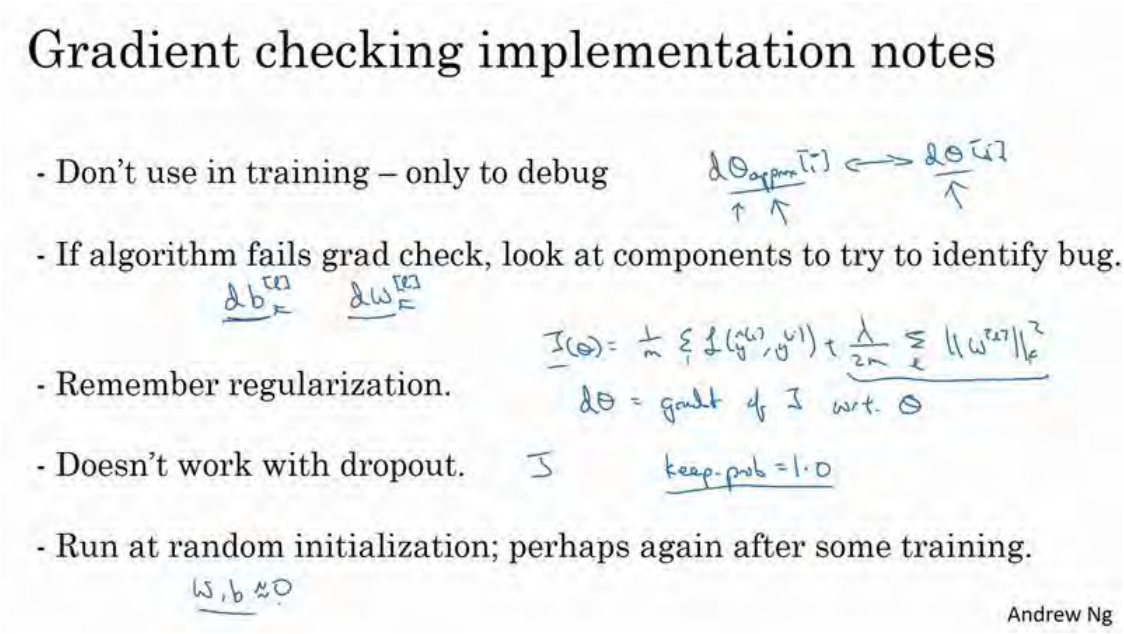
如果左边这个方程式结果是10^{-3}，我就会担心是否存在 bug，计算结果应该比10^{-3}小很多，如果比10^{-3}大很多，我就会很担心，担心是否存在 bug。这时应该仔细检查所有θ项，看是否有一个具体的i值，使得$d\theta_{\text{approx}}[i]$与$d\theta[i]$大不相同，并用它来追踪一些求导计算是否正确，经过一些调试，最终结果会是这种非常小的值（10^{-7}），那么，你的实施可能是正确的。

在实施神经网络时，我经常需要执行 **foreprop** 和 **backprop**，然后我可能发现这个梯度检验有一个相对较大的值，我会怀疑存在 bug，然后开始调试，调试，调试，调试一段时间后，我得到一个很小的梯度检验值，现在我可以很自信的说，神经网络实施是正确的。

现在你已经了解了梯度检验的工作原理，它帮助我在神经网络实施中发现了很多 bug，希望它对你也有所帮助。

1.14 梯度检验应用的注意事项（Gradient Checking Implementation Notes）

这节课，分享一些关于如何在神经网络实施梯度检验的实用技巧和注意事项。



首先，不要在训练中使用梯度检验，它只用于调试。我的意思是，计算所有i值的$d\theta_{\text{approx}}[i]$是一个非常漫长的计算过程，为了实施梯度下降，你必须使用W和b **backprop** 来计算$d\theta$，并使用 **backprop** 来计算导数，只要调试的时候，你才会计算它，来确认数值是否接近$d\theta$。完成后，你会关闭梯度检验，梯度检验的每一个迭代过程都不执行它，因为它太慢了。

第二点，如果算法的梯度检验失败，要检查所有项，检查每一项，并试着找出 bug，也就是说，如果$d\theta_{\text{approx}}[i]$与 dθ[i]的值相差很大，我们要做的就是查找不同的 i 值，看看是哪个导致$d\theta_{\text{approx}}[i]$与$d\theta[i]$的值相差这么多。举个例子，如果你发现，相对某些层或某层的θ或$d\theta$的值相差很大，但是$dw^{[l]}$的各项非常接近，注意θ的各项与b和w的各项都是一一对应的，这时，你可能会发现，在计算参数b的导数db的过程中存在 bug。反过来也是一样，如果你发现它们的值相差很大，$d\theta_{\text{approx}}[i]$的值与$d\theta[i]$的值相差很大，你会发现所有这些项目都来自于$dw$或某层的$dw$，可能帮你定位 bug 的位置，虽然未必能够帮你准确定位 bug 的位置，但它可以帮助你估测需要在哪些地方追踪 bug。

第三点，在实施梯度检验时，如果使用正则化，请注意正则项。如果代价函数$J(\theta) =$

$\frac{1}{m}\sum L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum ||W^{[l]}||^2$，这就是代价函数 J 的定义，$d\theta$等于与$\theta$相关的$J$函数的梯度，包括这个正则项，记住一定要包括这个正则项。

第四点，梯度检验不能与 **dropout** 同时使用，因为每次迭代过程中，**dropout** 会随机消除隐藏层单元的不同子集，难以计算 **dropout** 在梯度下降上的代价函数 J。因此 **dropout** 可作为优化代价函数J的一种方法，但是代价函数 J 被定义为对所有指数极大的节点子集求和。而在任何迭代过程中，这些节点都有可能被消除，所以很难计算代价函数J。你只是对成本函数做抽样，用 **dropout**，每次随机消除不同的子集，所以很难用梯度检验来双重检验 **dropout** 的计算，所以我一般不同时使用梯度检验和 **dropout**。如果你想这样做，可以把 **dropout** 中的 **keepprob** 设置为 1.0，然后打开 **dropout**，并寄希望于 **dropout** 的实施是正确的，你还可以做点别的，比如修改节点丢失模式确定梯度检验是正确的。实际上，我一般不这么做，我建议关闭 **dropout**，用梯度检验进行双重检查，在没有 **dropout** 的情况下，你的算法至少是正确的，然后打开 **dropout**。

最后一点，也是比较微妙的一点，现实中几乎不会出现这种情况。当w和b接近 0 时，梯度下降的实施是正确的，在随机初始化过程中……，但是在运行梯度下降时，w和b变得更大。可能只有在w和b接近 0 时，**backprop** 的实施才是正确的。但是当W和b变大时，它会变得越来越不准确。你需要做一件事，我不经常这么做，就是在随机初始化过程中，运行梯度检验，然后再训练网络，w和b会有一段时间远离 0，如果随机初始化值比较小，反复训练网络之后，再重新运行梯度检验。

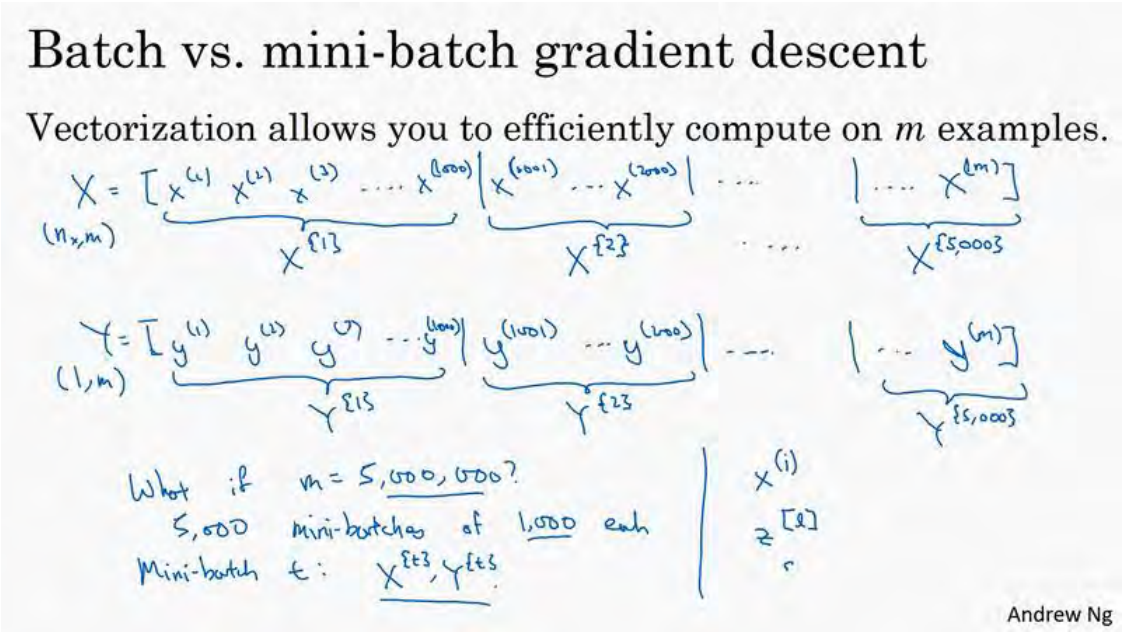
这就是梯度检验，恭喜大家，这是本周最后一课了。回顾这一周，我们讲了如何配置训练集，验证集和测试集，如何分析偏差和方差，如何处理高偏差或高方差以及高偏差和高方差并存的问题，如何在神经网络中应用不同形式的正则化，如$L2$正则化和 **dropout**，还有加快神经网络训练速度的技巧，最后是梯度检验。这一周我们学习了很多内容，你可以在本周编程作业中多多练习这些概念。祝你好运，期待下周再见。

第二周：优化算法 (Optimization algorithms)

2.1 Mini-batch 梯度下降（Mini-batch gradient descent）

本周将学习优化算法，这能让你的神经网络运行得更快。机器学习的应用是一个高度依赖经验的过程，伴随着大量迭代的过程，你需要训练诸多模型，才能找到合适的那一个，所以，优化算法能够帮助你快速训练模型。

其中一个难点在于，深度学习没有在大数据领域发挥最大的效果，我们可以利用一个巨大的数据集来训练神经网络，而在巨大的数据集基础上进行训练速度很慢。因此，你会发现，使用快速的优化算法，使用好用的优化算法能够大大提高你和团队的效率，那么，我们首先来谈谈 **mini-batch** 梯度下降法。

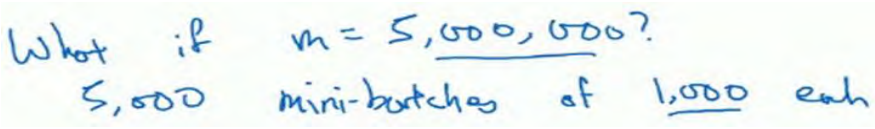


你之前学过，向量化能够让你有效地对所有m个样本进行计算，允许你处理整个训练集，而无需某个明确的公式。所以我们要把训练样本放大巨大的矩阵X当中去，$X = [x^{(1)}\ x^{(2)}\ x^{(3)} \ldots\ldots x^{(m)}]$，$Y$也是如此，$Y = [y^{(1)}\ y^{(2)}\ y^{(3)} \ldots\ldots y^{(m)}]$，所以$X$的维数是$(n_x, m)$，$Y$的维数是$(1, m)$，向量化能够让你相对较快地处理所有$m$个样本。如果$m$很大的话，处理速度仍然缓慢。比如说，如果$m$是 500 万或 5000 万或者更大的一个数，在对整个训练集执行梯度下降法时，你要做的是，你必须处理整个训练集，然后才能进行一步梯度下降法，然后你需要再重新处理 500 万个训练样本，才能进行下一步梯度下降法。所以如果你在处理完整个 500 万个样本的训练集之前，先让梯度下降法处理一部分，你的算法速度会更快，准确地

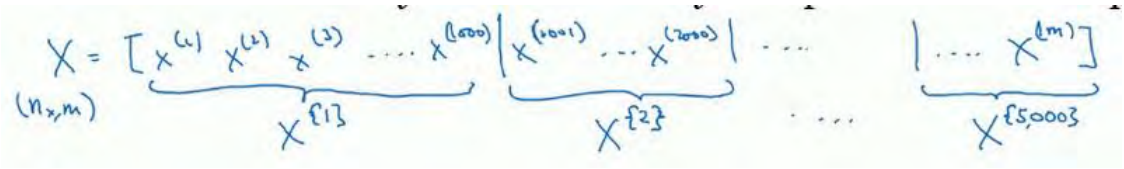
说，这是你可以做的一些事情。

你可以把训练集分割为小一点的子集训练，这些子集被取名为 **mini-batch**，假设每一个子集中只有 1000 个样本，那么把其中的$x^{(1)}$到$x^{(1000)}$取出来，将其称为第一个子训练集，也叫做 **mini-batch**，然后你再取出接下来的 1000 个样本，从$x^{(1001)}$到$x^{(2000)}$，然后再取 1000 个样本，以此类推。

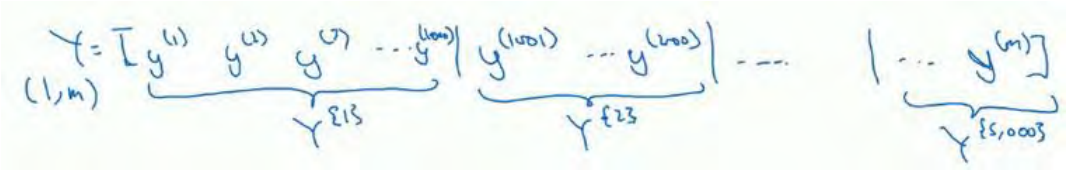
接下来我要说一个新的符号，把$x^{(1)}$到$x^{(1000)}$称为$X^{\{1\}}$，$x^{(1001)}$到$x^{(2000)}$称为$X^{\{2\}}$，如果你的训练样本一共有 500 万个，每个 **mini-batch** 都有 1000 个样本，也就是说，你有 5000 个 **mini-batch**，因为 5000 乘以 1000 就是 5000 万。



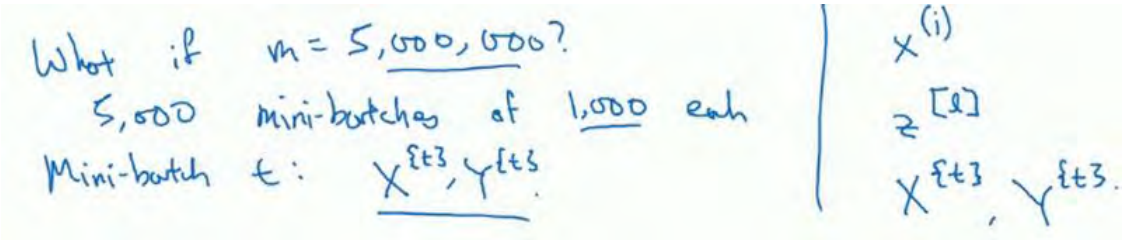
你共有 5000 个 **mini-batch**，所以最后得到是$X^{\{5000\}}$



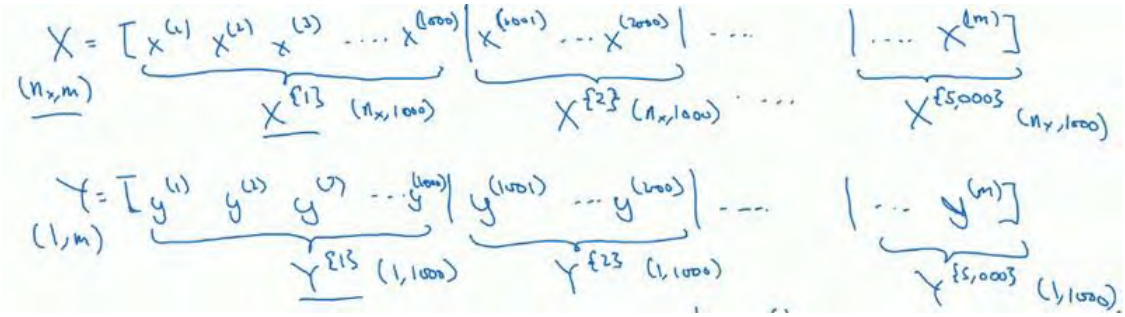
对Y也要进行相同处理，你也要相应地拆分Y的训练集，所以这是$Y^{\{1\}}$，然后从$y^{(1001)}$到$y^{(2000)}$，这个叫$Y^{\{2\}}$，一直到$Y^{\{5000\}}$。



mini-batch 的数量t组成了$X^{\{t\}}$和$Y^{\{t\}}$，这就是 1000 个训练样本，包含相应的输入输出对。



在继续课程之前，先确定一下我的符号，之前我们使用了上角小括号(i)表示训练集里的值，所以$x^{(i)}$是第i个训练样本。我们用了上角中括号$[l]$来表示神经网络的层数，$z^{[l]}$表示神经网络中第 l 层的 z 值，我们现在引入了大括号t来代表不同的 **mini-batch**，所以我们有$X^{\{t\}}$和$Y^{\{t\}}$，检查一下自己是否理解无误。

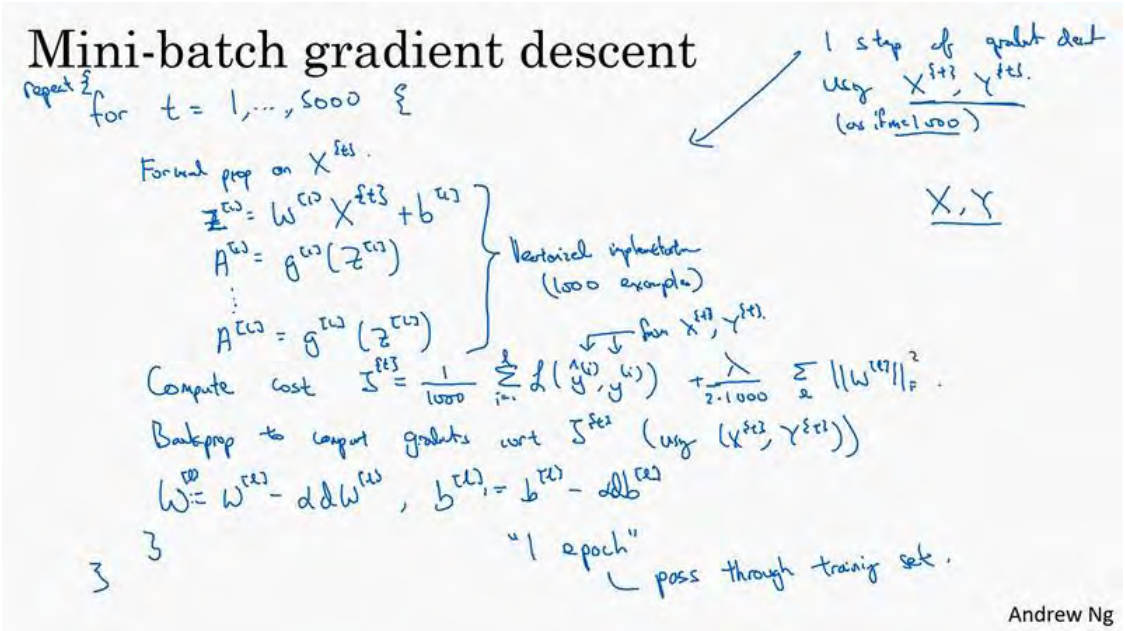


$X^{\{t\}}$和$Y^{\{t\}}$的维数：如果$X^{\{1\}}$是一个有 1000 个样本的训练集，或者说是 1000 个样本的x值，所以维数应该是$(n_x, 1000)$，$X^{\{2\}}$的维数应该是$(n_x, 1000)$，以此类推。因此所有的子集维数都是$(n_x, 1000)$，而这些（$Y^{\{t\}}$）的维数都是$(1,1000)$。

解释一下这个算法的名称，**batch** 梯度下降法指的是我们之前讲过的梯度下降法算法，就是同时处理整个训练集，这个名字就是来源于能够同时看到整个 **batch** 训练集的样本被处理，这个名字不怎么样，但就是这样叫它。

相比之下，**mini-batch** 梯度下降法，指的是我们在下一张幻灯片中会讲到的算法，你每次同时处理的单个的 **mini-batch** $X^{\{t\}}$和$Y^{\{t\}}$，而不是同时处理全部的X和Y训练集。

那么究竟 **mini-batch** 梯度下降法的原理是什么？在训练集上运行 **mini-batch** 梯度下降法，你运行 `for t=1……5000`，因为我们有 5000 个各有 1000 个样本的组，在 **for** 循环里你要做得基本就是对$X^{\{t\}}$和$Y^{\{t\}}$执行一步梯度下降法。假设你有一个拥有 1000 个样本的训练集，而且假设你已经很熟悉一次性处理完的方法，你要用向量化去几乎同时处理 1000 个样本。

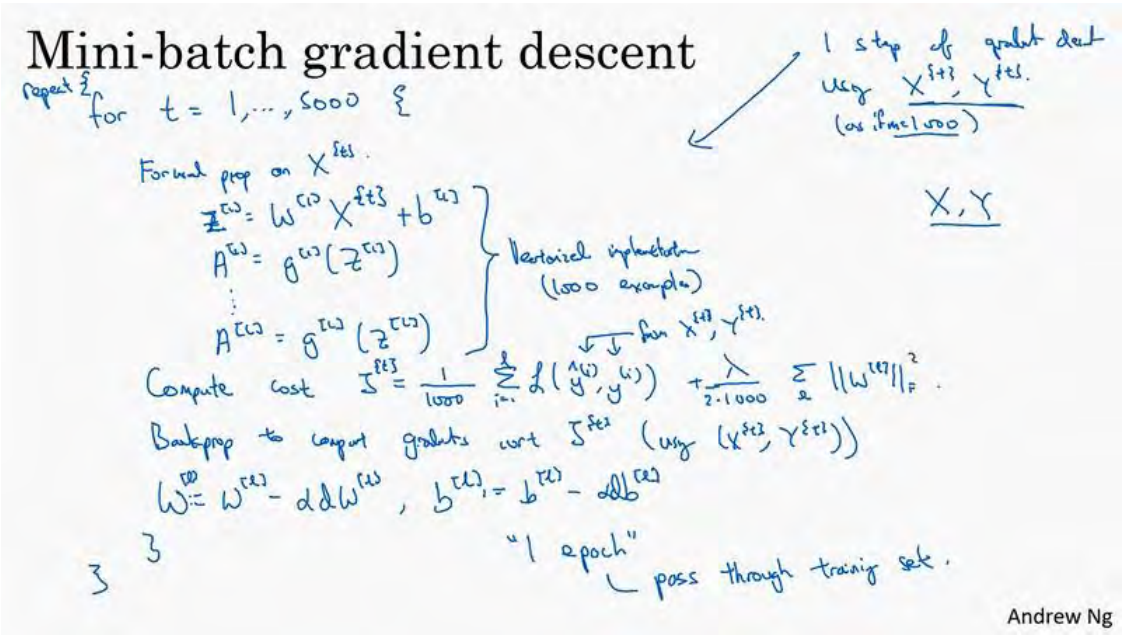


首先对输入也就是$X^{\{t\}}$，执行前向传播，然后执行$z^{[1]} = W^{[1]}X + b^{[1]}$，之前我们这里只

有，但是现在你正在处理整个训练集，你在处理第一个 **mini-batch**，在处理 **mini-batch** 时它变成了$X^{\{t\}}$，即$z^{[1]} = W^{[1]}X^{\{t\}} + b^{[1]}$，然后执行$A^{[1]k} = g^{[1]}(Z^{[1]})$，之所以用大写的$Z$是因为这是一个向量内涵，以此类推，直到$A^{[L]} = g^{[L]}(Z^{[L]})$，这就是你的预测值。注意这里你需要用到一个向量化的执行命令，这个向量化的执行命令，一次性处理 1000 个而不是 500 万个样本。接下来你要计算损失成本函数J，因为子集规模是 1000，$J = \frac{1}{1000}\sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)})$，说明一下，这（$L(\hat{y}^{(i)}, y^{(i)})$）指的是来自于 **mini-batch**$X^{\{t\}}$和$Y^{\{t\}}$中的样本。

如果你用到了正则化，你也可以使用正则化的术语，$J = \frac{1}{1000}\sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{21000}\sum_{l} ||w^{[l]}||_F^2$，因为这是一个 **mini-batch** 的损失，所以我将J损失记为上角标t，放在大括号里（$J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{21000}\sum_{l} ||w^{[l]}||_F^2$）。

你也会注意到，我们做的一切似曾相识，其实跟之前我们执行梯度下降法如出一辙，除了你现在的对象不是X，Y，而是$X^{\{t\}}$和$Y^{\{t\}}$。接下来，你执行反向传播来计算$J^{\{t\}}$的梯度，你只是使用$X^{\{t\}}$和$Y^{\{t\}}$，然后你更新加权值，W实际上是$W^{[l]}$，更新为$W^{[l]} := W^{[l]} - adW^{[l]}$，对$b$做相同处理，$b^{[l]} := b^{[l]} - adb^{[l]}$。这是使用 **mini-batch** 梯度下降法训练样本的一步，我写下的代码也可被称为进行“一代”（**1 epoch**）的训练。一代这个词意味着只是一次遍历了训练集。



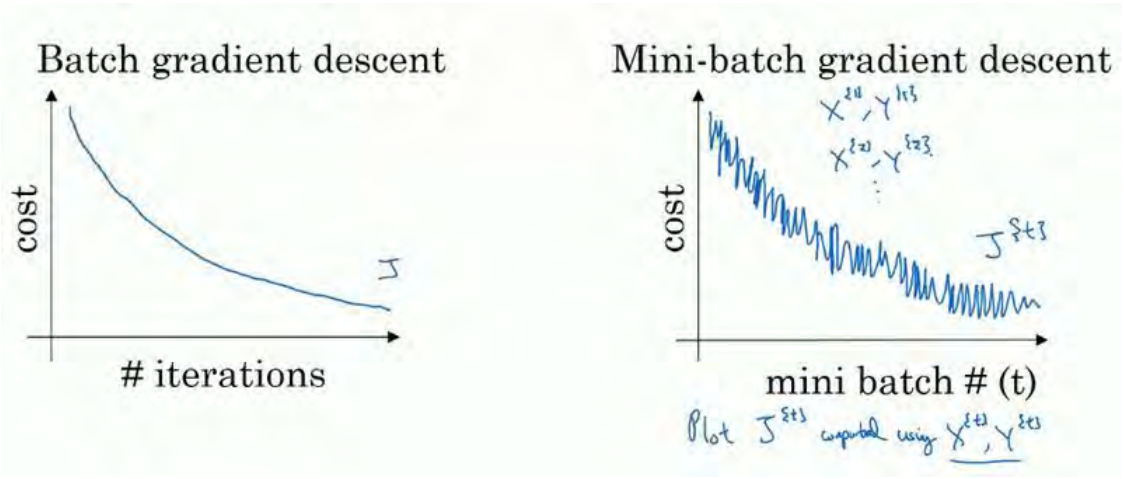
使用 **batch** 梯度下降法，一次遍历训练集只能让你做一个梯度下降，使用 **mini-batch** 梯度下降法，一次遍历训练集，能让你做 5000 个梯度下降。当然正常来说你想要多次遍历训

练集，还需要为另一个 **while** 循环设置另一个 **for** 循环。所以你可以一直处理遍历训练集，直到最后你能收敛到一个合适的精度。

如果你有一个丢失的训练集，**mini-batch** 梯度下降法比 **batch** 梯度下降法运行地更快，所以几乎每个研习深度学习的人在训练巨大的数据集时都会用到，下一个视频中，我们将进一步深度讨论 **mini-batch** 梯度下降法，你也会因此更好地理解它的作用和原理。

2.2 理解 mini-batch 梯度下降法（Understanding mini-batch gradient descent）

在上周视频中，你知道了如何利用 **mini-batch** 梯度下降法来开始处理训练集和开始梯度下降，即使你只处理了部分训练集，即使你是第一次处理，本视频中，我们将进一步学习如何执行梯度下降法，更好地理解其作用和原理。

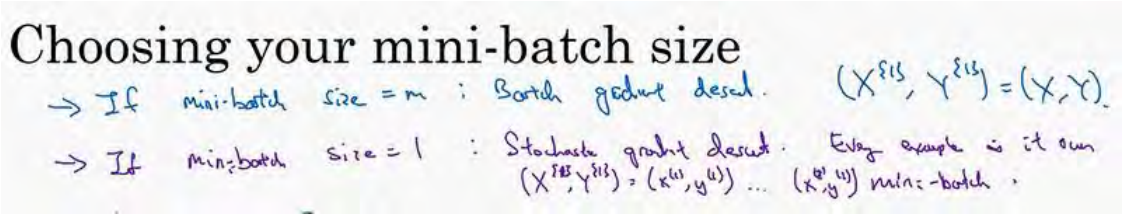


使用 **batch** 梯度下降法时，每次迭代你都需要历遍整个训练集，可以预期每次迭代成本都会下降，所以如果成本函数J是迭代次数的一个函数，它应该会随着每次迭代而减少，如果J在某次迭代中增加了，那肯定出了问题，也许你的学习率太大。

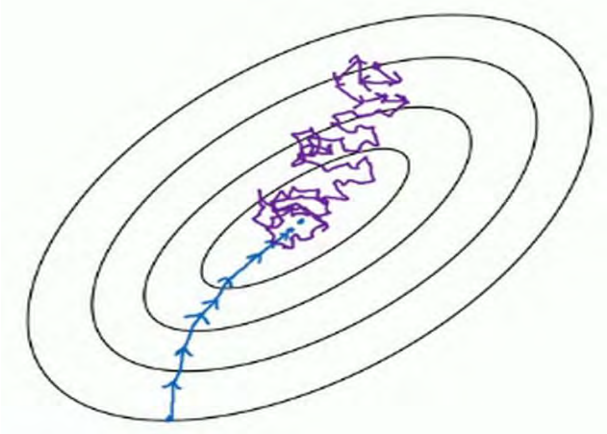
使用 **mini-batch** 梯度下降法，如果你作出成本函数在整个过程中的图，则并不是每次迭代都是下降的，特别是在每次迭代中，你要处理的是$X^{\{t\}}$和$Y^{\{t\}}$，如果要作出成本函数$J^{\{t\}}$的图，而$J^{\{t\}}$只和$X^{\{t\}}$，$Y^{\{t\}}$有关，也就是每次迭代下你都在训练不同的样本集或者说训练不同的 **mini-batch**，如果你要作出成本函数J的图，你很可能会看到这样的结果，走向朝下，但有更多的噪声，所以如果你作出$J^{\{t\}}$的图，因为在训练 **mini-batch** 梯度下降法时，会经过多代，你可能会看到这样的曲线。没有每次迭代都下降是不要紧的，但走势应该向下，噪声产生的原因在于也许$X^{\{1\}}$和$Y^{\{1\}}$是比较容易计算的 **mini-batch**，因此成本会低一些。不过也许出于偶然，$X^{\{2\}}$和$Y^{\{2\}}$是比较难运算的 **mini-batch**，或许你需要一些残缺的样本，这样一来，成本会更高一些，所以才会出现这些摆动，因为你是在运行 **mini-batch** 梯度下降法作出成本函数图。

你需要决定的变量之一是 **mini-batch** 的大小，m就是训练集的大小，极端情况下，如果 **mini-batch** 的大小等于m，其实就是 **batch** 梯度下降法，在这种极端情况下，你就有了 **mini-**

batch $X^{\{1\}}$和$Y^{\{1\}}$，并且该 **mini-batch** 等于整个训练集，所以把 **mini-batch** 大小设为m可以得到 **batch** 梯度下降法。



另一个极端情况，假设 **mini-batch** 大小为 1，就有了新的算法，叫做随机梯度下降法，每个样本都是独立的 **mini-batch**，当你看第一个 **mini-batch**，也就是$X^{\{1\}}$和$Y^{\{1\}}$，如果 **mini-batch** 大小为 1，它就是你的第一个训练样本，这就是你的第一个训练样本。接着再看第二个 **mini-batch**，也就是第二个训练样本，采取梯度下降步骤，然后是第三个训练样本，以此类推，一次只处理一个。

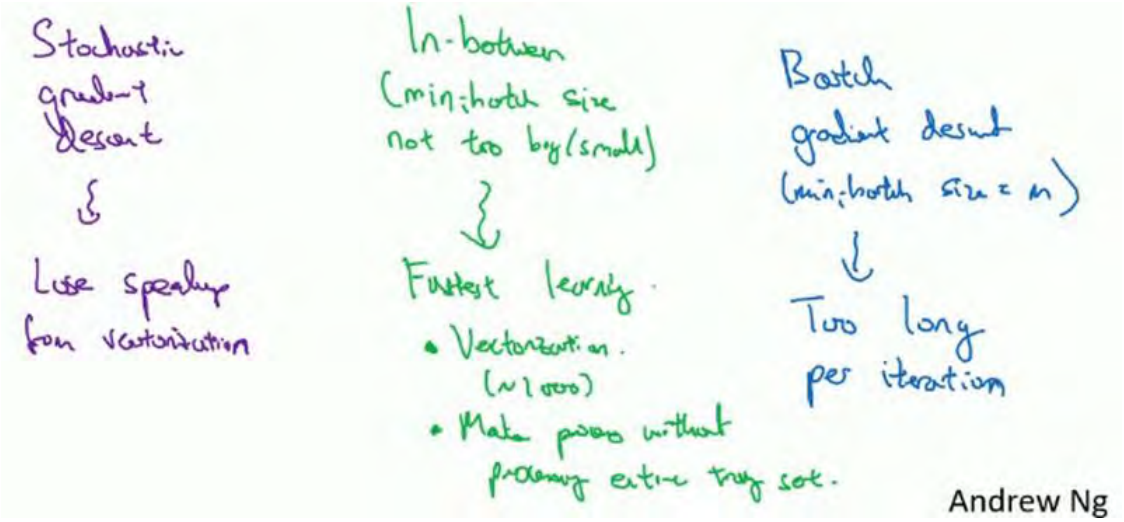


看在两种极端下成本函数的优化情况，如果这是你想要最小化的成本函数的轮廓，最小值在那里，**batch** 梯度下降法从某处开始，相对噪声低些，幅度也大一些，你可以继续找最小值。

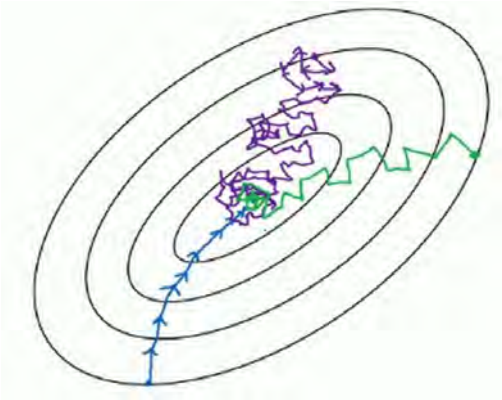
相反，在随机梯度下降法中，从某一点开始，我们重新选取一个起始点，每次迭代，你只对一个样本进行梯度下降，大部分时候你向着全局最小值靠近，有时候你会远离最小值，因为那个样本恰好给你指的方向不对，因此随机梯度下降法是有很多噪声的，平均来看，它最终会靠近最小值，不过有时候也会方向错误，因为随机梯度下降法永远不会收敛，而是会一直在最小值附近波动，但它并不会在达到最小值并停留在此。

实际上你选择的 **mini-batch** 大小在二者之间，大小在 1 和m之间，而 1 太小了，m太大了，原因在于如果使用 **batch** 梯度下降法，**mini-batch** 的大小为m，每个迭代需要处理大量训练样本，该算法的主要弊端在于特别是在训练样本数量巨大的时候，单次迭代耗时太长。

如果训练样本不大，**batch** 梯度下降法运行地很好。



相反，如果使用随机梯度下降法，如果你只要处理一个样本，那这个方法很好，这样做没有问题，通过减小学习率，噪声会被改善或有所减小，但随机梯度下降法的一大缺点是，你会失去所有向量化带给你的加速，因为一次性只处理了一个训练样本，这样效率过于低下，所以实践中最好选择不大不小的 **mini-batch** 尺寸，实际上学习率达到最快。你会发现两个好处，一方面，你得到了大量向量化，上个视频中我们用过的例子中，如果 **mini-batch** 大小为 1000 个样本，你就可以对 1000 个样本向量化，比你一次性处理多个样本快得多。另一方面，你不需要等待整个训练集被处理完就可以开始进行后续工作，再用一下上个视频的数字，每次训练集允许我们采取 5000 个梯度下降步骤，所以实际上一些位于中间的 **mini-batch** 大小效果最好。



用 **mini-batch** 梯度下降法，我们从这里开始，一次迭代这样做，两次，三次，四次，它不会总朝向最小值靠近，但它比随机梯度下降要更持续地靠近最小值的方向，它也不一定在很小的范围内收敛或者波动，如果出现这个问题，可以慢慢减少学习率，我们在下个视频会讲到学习率衰减，也就是如何减小学习率。

如果 **mini-batch** 大小既不是 1 也不是m，应该取中间值，那应该怎么选择呢？其实是有指导原则的。

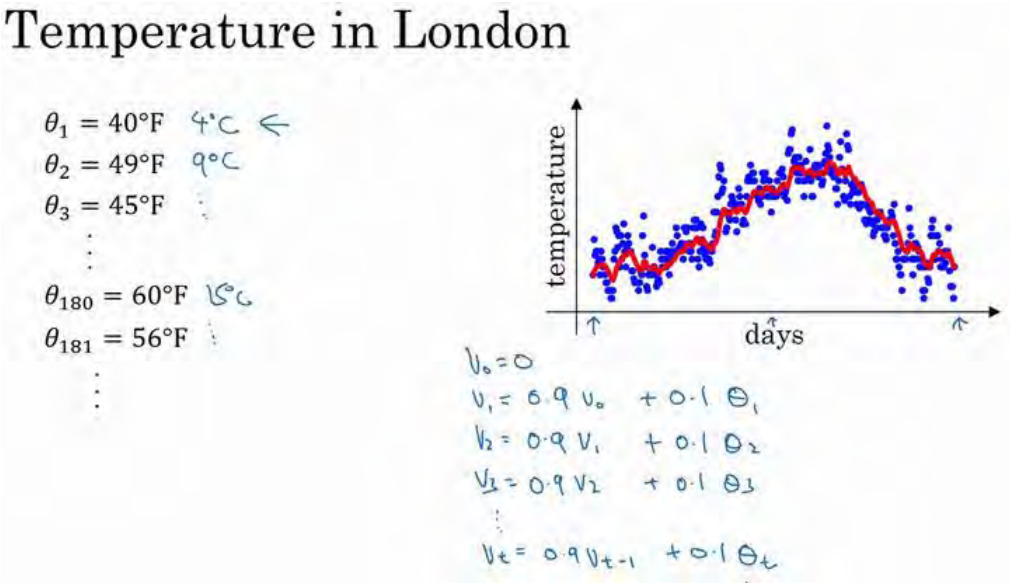
首先，如果训练集较小，直接使用 **batch** 梯度下降法，样本集较小就没必要使用 **mini-batch** 梯度下降法，你可以快速处理整个训练集，所以使用 **batch** 梯度下降法也很好，这里的少是说小于 2000 个样本，这样比较适合使用 **batch** 梯度下降法。不然，样本数目较大的话，一般的 **mini-batch** 大小为 64 到 512，考虑到电脑内存设置和使用的方式，如果 **mini-batch** 大小是 2 的n次方，代码会运行地快一些，64 就是 2 的 6 次方，以此类推，128 是 2 的 7 次方，256 是 2 的 8 次方，512 是 2 的 9 次方。所以我经常把 **mini-batch** 大小设成 2 的次方。在上一个视频里，我的 **mini-batch** 大小设为了 1000，建议你可以试一下 1024，也就是 2 的 10 次方。也有 **mini-batch** 的大小为 1024，不过比较少见，64 到 512 的 **mini-batch** 比较常见。

最后需要注意的是在你的 **mini-batch** 中，要确保$X^{\{t\}}$和$Y^{\{t\}}$要符合 **CPU/GPU** 内存，取决于你的应用方向以及训练集的大小。如果你处理的 **mini-batch** 和 **CPU/GPU** 内存不相符，不管你用什么方法处理数据，你会注意到算法的表现急转直下变得惨不忍睹，所以我希望你对一般人们使用的 **mini-batch** 大小有一个直观了解。事实上 **mini-batch** 大小是另一个重要的变量，你需要做一个快速尝试，才能找到能够最有效地减少成本函数的那个，我一般会尝试几个不同的值，几个不同的 2 次方，然后看能否找到一个让梯度下降优化算法最高效的大小。希望这些能够指导你如何开始找到这一数值。

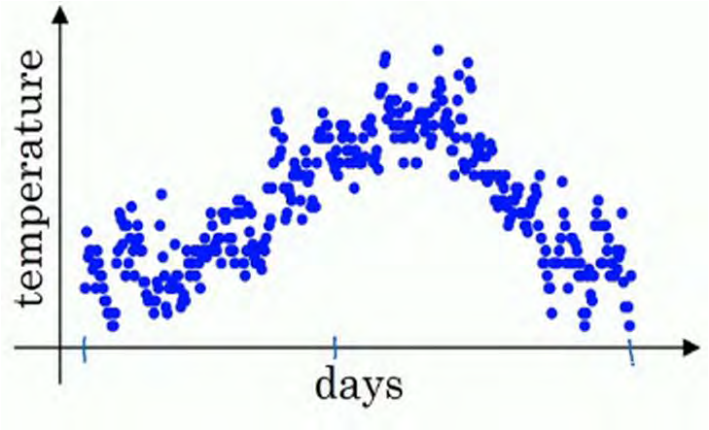
你学会了如何执行 **mini-batch** 梯度下降，令算法运行得更快，特别是在训练样本数目较大的情况下。不过还有个更高效的算法，比梯度下降法和 **mini-batch** 梯度下降法都要高效的多，我们在接下来的视频中将为大家一一讲解。

2.3 指数加权平均数（Exponentially weighted averages）

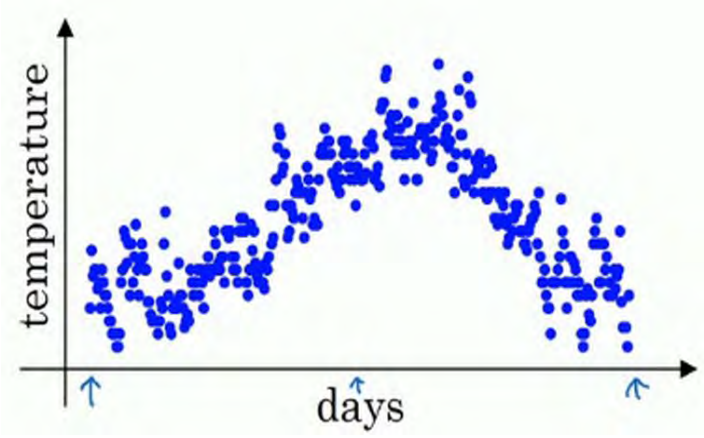
我想向你展示几个优化算法，它们比梯度下降法快，要理解这些算法，你需要用到指数加权平均，在统计中也叫做指数加权移动平均，我们首先讲这个，然后再来讲更复杂的优化算法。



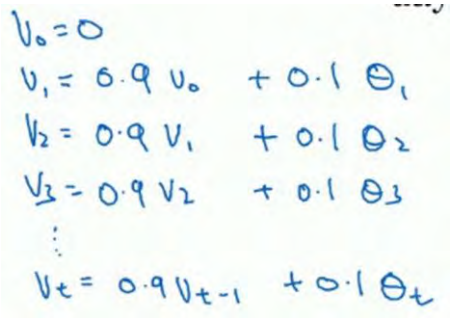
虽然现在我生活在美国，实际上我生于英国伦敦。比如我这儿有去年伦敦的每日温度，所以 1 月 1 号，温度是 40 华氏度，相当于 4 摄氏度。我知道世界上大部分地区使用摄氏度，但是美国使用华氏度。在 1 月 2 号是 9 摄氏度等等。在年中的时候，一年 365 天，年中就是说，大概 180 天的样子，也就是 5 月末，温度是 60 华氏度，也就是 15 摄氏度等等。夏季温度转暖，然后冬季降温。



你用数据作图，可以得到以下结果，起始日在 1 月份，这里是夏季初，这里是年末，相当于 12 月末。



这里是 1 月 1 号，年中接近夏季的时候，随后就是年末的数据，看起来有些杂乱，如果要计算趋势的话，也就是温度的局部平均值，或者说移动平均值。

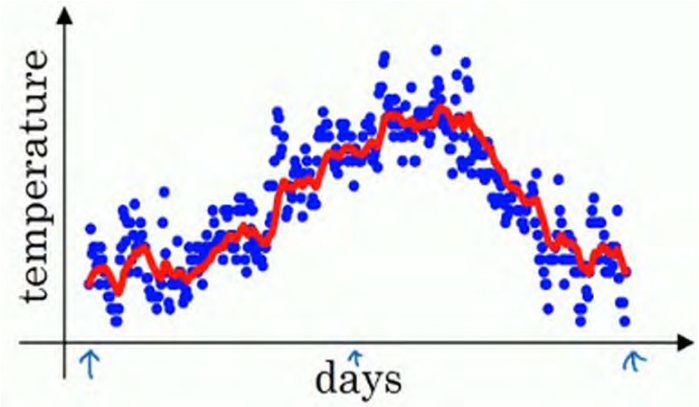


你要做的是，首先使$v_0 = 0$，每天，需要使用 0.9 的加权数之前的数值加上当日温度的 0.1 倍，即$v_1 = 0.9v_0 + 0.1\theta_1$，所以这里是第一天的温度值。

第二天，又可以获得一个加权平均数，0.9 乘以之前的值加上当日的温度 0.1 倍，即$v_2 = 0.9v_1 + 0.1\theta_2$，以此类推。

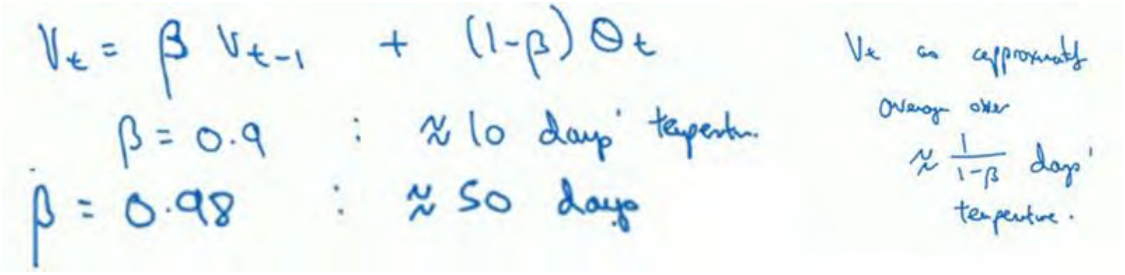
第二天值加上第三日数据的 0.1，如此往下。大体公式就是某天的v等于前一天v值的 0.9 加上当日温度的 0.1。

如此计算，然后用红线作图的话，便得到这样的结果。



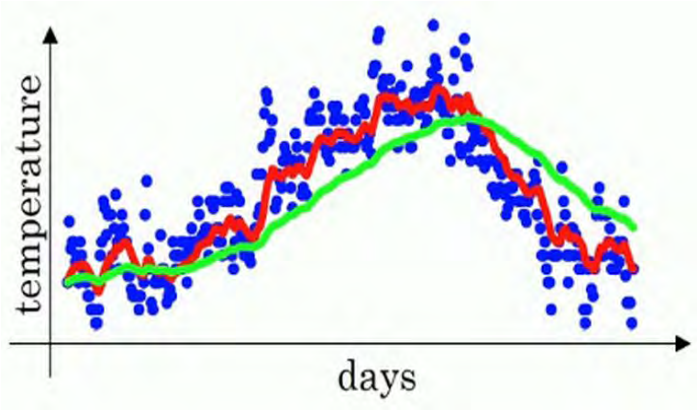
你得到了移动平均值，每日温度的指数加权平均值。

看一下上一张幻灯片里的公式，$v_t = 0.9v_{t-1} + 0.1\theta_t$，我们把 0.9 这个常数变成$\beta$，将之前的 0.1 变成$(1-\beta)$，即$v_t = \beta v_{t-1} + (1-\beta)\theta_t$



由于以后我们要考虑的原因，在计算时可视v_t大概是$\frac{1}{(1-\beta)}$的每日温度，如果β是 0.9，你会想，这是十天的平均值，也就是红线部分。

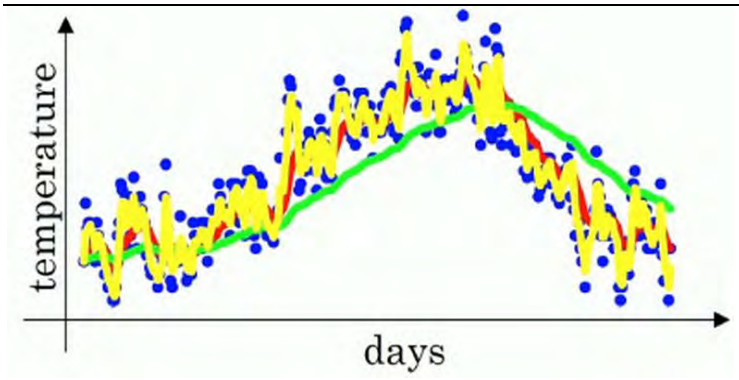
我们来试试别的，将β设置为接近 1 的一个值，比如 0.98，计算$\frac{1}{(1-0.98)} = 50$，这就是粗略平均了一下，过去 50 天的温度，这时作图可以得到绿线。



这个高值β要注意几点，你得到的曲线要平坦一些，原因在于你多平均了几天的温度，所以这个曲线，波动更小，更加平坦，缺点是曲线进一步右移，因为现在平均的温度值更多，要平均更多的值，指数加权平均公式在温度变化时，适应地更缓慢一些，所以会出现一定延迟，因为当$\beta = 0.98$，相当于给前一天的值加了太多权重，只有 0.02 的权重给了当日的值，所以温度变化时，温度上下起伏，当β 较大时，指数加权平均值适应地更缓慢一些。

我们可以再换一个值试一试，如果β是另一个极端值，比如说 0.5，根据右边的公式（$\frac{1}{(1-\beta)}$），这是平均了两天的温度。

作图运行后得到黄线。



由于仅平均了两天的温度，平均的数据太少，所以得到的曲线有更多的噪声，有可能出现异常值，但是这个曲线能够更快适应温度变化。

所以指数加权平均数经常被使用，再说一次，它在统计学中被称为指数加权移动平均值，我们就简称为指数加权平均数。通过调整这个参数（β），或者说后面的算法学习，你会发现这是一个很重要的参数，可以取得稍微不同的效果，往往中间有某个值效果最好，β为中间值时得到的红色曲线，比起绿线和黄线更好地平均了温度。

现在你知道计算指数加权平均数的基本原理，下一个视频中，我们再聊聊它的本质作用。

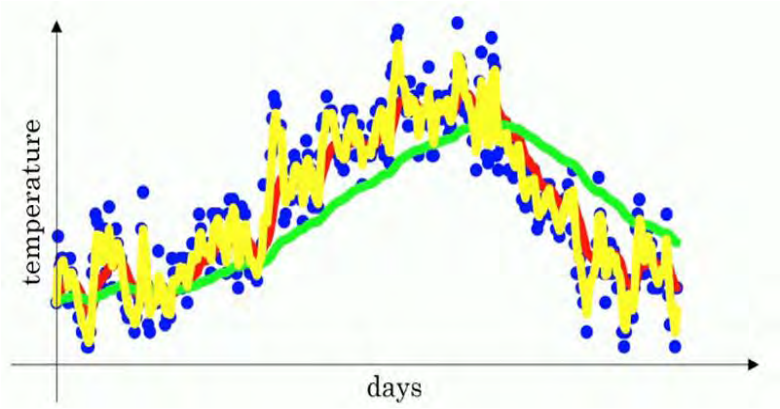
2.4 理解指数加权平均数（Understanding exponentially weighted averages）

上个视频中，我们讲到了指数加权平均数，这是几个优化算法中的关键一环，而这几个优化算法能帮助你训练神经网络。本视频中，我希望进一步探讨算法的本质作用。

回忆一下这个计算指数加权平均数的关键方程。

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$\beta = 0.9$的时候，得到的结果是红线，如果它更接近于 1，比如 0.98，结果就是绿线，如果β小一点，如果是 0.5，结果就是黄线。



我们进一步地分析，来理解如何计算出每日温度的平均值。

同样的公式，$v_t = \beta v_{t-1} + (1-\beta)\theta_t$

使$\beta = 0.9$，写下相应的几个公式，所以在执行的时候，t从 0 到 1 到 2 到 3，t的值在不断增加，为了更好地分析，我写的时候使得t的值不断减小，然后继续往下写。

$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$
$$v_{98} = 0.9v_{97} + 0.1\theta_{98}$$
$$\cdots$$

首先看第一个公式，理解v_{100}是什么？我们调换一下这两项（$0.9v_{99}0.1\theta_{100}$），$v_{100} = 0.1\theta_{100} + 0.9v_{99}$。

那么v_{99}是什么？我们就代入这个公式（$v_{99} = 0.1\theta_{99} + 0.9v_{98}$），所以：

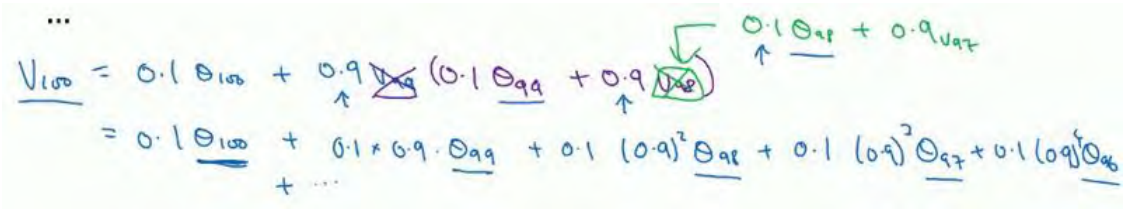
$v_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9v_{98})$。

那么v_{98}是什么？你可以用这个公式计算（$v_{98} = 0.1\theta_{98} + 0.9v_{97}$），把公式代进去，所以：

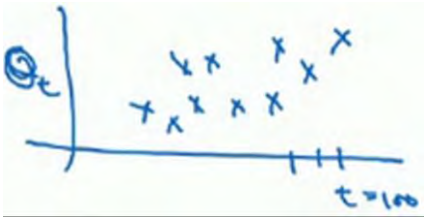
$v_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9v_{97}))$。

以此类推，如果你把这些括号都展开，

$$v_{100} = 0.1\theta_{100} + 0.1 \times 0.9\theta_{99} + 0.1 \times (0.9)^2\theta_{98} + 0.1 \times (0.9)^3\theta_{97} + 0.1 \times (0.9)^4\theta_{96} + \cdots$$

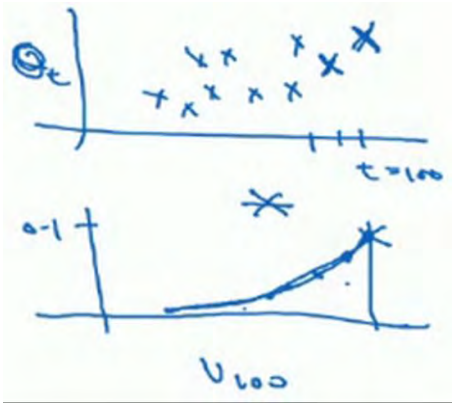


所以这是一个加和并平均，100 号数据，也就是当日温度。我们分析v_{100}的组成，也就是在一年第 100 天计算的数据，但是这个是总和，包括 100 号数据，99 号数据，97 号数据等等。画图的一个办法是，假设我们有一些日期的温度，所以这是数据，这是t，所以 100 号数据有个数值，99 号数据有个数值，98 号数据等等，t为 100，99，98 等等，这就是数日的温度数值。



然后我们构建一个指数衰减函数，从 0.1 开始，到0.1×0.9，到$0.1 \times (0.9)^2$，以此类推，所以就有了这个指数衰减函数。

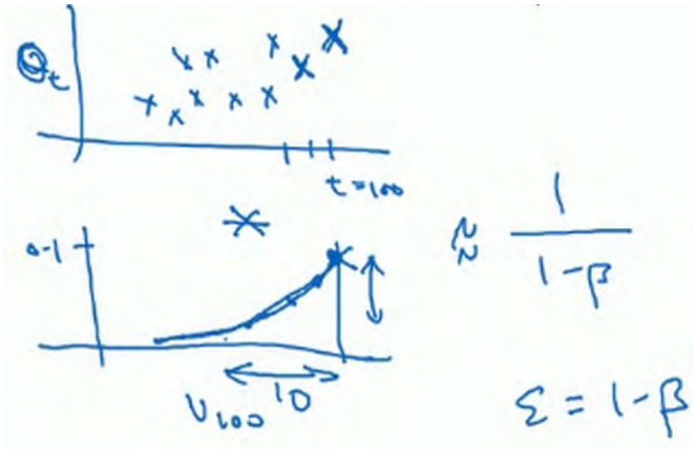
计算v_{100}是通过，把两个函数对应的元素，然后求和，用这个数值 100 号数据值乘以 0.1，99 号数据值乘以 0.1 乘以$(0.9)^2$，这是第二项，以此类推，所以选取的是每日温度，将其与指数衰减函数相乘，然后求和，就得到了v_{100}。



结果是，稍后我们详细讲解，不过所有的这些系数（$0.10.1 \times 0.90.1 \times (0.9)^2 0.1 \times (0.9)^3$ …），相加起来为 1 或者逼近 1，我们称之为偏差修正，下个视频会涉及。

最后也许你会问，到底需要平均多少天的温度。实际上$(0.9)^{10}$大约为 0.35，这大约是$\frac{1}{e}$，e 是自然算法的基础之一。大体上说，如果有$1-\varepsilon$，在这个例子中，$\varepsilon = 0.1$，所以$1-\varepsilon = 0.9$，$(1-\varepsilon)^{(\frac{1}{\varepsilon})}$约等于$\frac{1}{e}$，大约是 0.34，0.35，换句话说，10 天后，曲线的高度下降到$\frac{1}{3}$，相当于在峰值的$\frac{1}{e}$。

又因此当$\beta = 0.9$的时候，我们说仿佛你在计算一个指数加权平均数，只关注了过去 10 天的温度，因为 10 天后，权重下降到不到当日权重的三分之一。



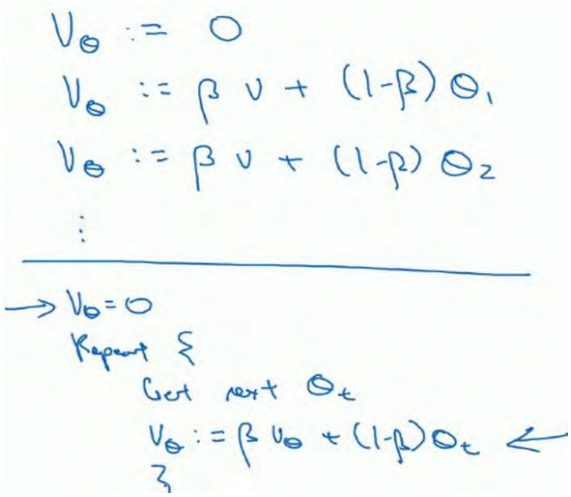
相反，如果，那么 0.98 需要多少次方才能达到这么小的数值？$(0.98)^{50}$大约等于$\frac{1}{e}$，所

以前 50 天这个数值比$\frac{1}{e}$大，数值会快速衰减，所以本质上这是一个下降幅度很大的函数，你可以看作平均了 50 天的温度。因为在例子中，要代入等式的左边，$\varepsilon = 0.02$，所以$\frac{1}{\varepsilon}$为 50，我们由此得到公式，我们平均了大约$\frac{1}{(1-\beta)}$天的温度，这里ε代替了$1-\beta$，也就是说根据一些常数，你能大概知道能够平均多少日的温度，不过这只是思考的大致方向，并不是正式的数学证明。

$$
\begin{aligned}
v_0 &= 0 \\
v_1 &= \beta v_0 + (1-\beta)\,\theta_1 \\
v_2 &= \beta v_1 + (1-\beta)\,\theta_2 \\
v_3 &= \beta v_2 + (1-\beta)\,\theta_3 \\
&\cdots
\end{aligned}
$$

最后讲讲如何在实际中执行，还记得吗？我们一开始将v_0设置为 0，然后计算第一天v_1，然后v_2，以此类推。

现在解释一下算法，可以将v_0，v_1，v_2等等写成明确的变量，不过在实际中执行的话，你要做的是，一开始将v初始化为 0，然后在第一天使$v:=\beta v+(1-\beta)\theta_1$，然后第二天，更新 v 值，$v:=\beta v+(1-\beta)\theta_2$，以此类推，有些人会把$v$加下标，来表示$v$是用来计算数据的指数加权平均数。



再说一次，但是换个说法，$v_\theta = 0$，然后每一天，拿到第t天的数据，把v更新为$v:=\beta v_\theta + (1-\beta)\theta_t$。

指数加权平均数公式的好处之一在于，它占用极少内存，电脑内存中只占用一行数字而已，然后把最新数据代入公式，不断覆盖就可以了，正因为这个原因，其效率，它基本上只占用一行代码，计算指数加权平均数也只占用单行数字的存储和内存，当然它并不是最好的，

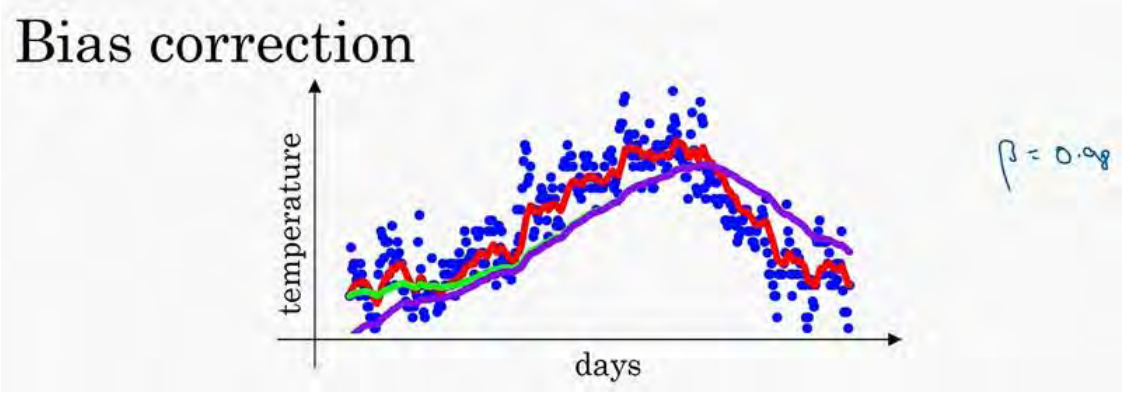
也不是最精准的计算平均数的方法。如果你要计算移动窗，你直接算出过去 10 天的总和，过去 50 天的总和，除以 10 和 50 就好，如此往往会得到更好的估测。但缺点是，如果保存所有最近的温度数据，和过去 10 天的总和，必须占用更多的内存，执行更加复杂，计算成本也更加高昂。

所以在接下来的视频中，我们会计算多个变量的平均值，从计算和内存效率来说，这是一个有效的方法，所以在机器学习中会经常使用，更不用说只要一行代码，这也是一个优势。

现在你学会了计算指数加权平均数，你还需要知道一个专业概念，叫做偏差修正，下一个视频我们会讲到它，接着你就可以用它构建更好的优化算法，而不是简单直接的梯度下降法。

2.5 指数加权平均的偏差修正（Bias correction in exponentially weighted averages）

你学过了如何计算指数加权平均数，有一个技术名词叫做偏差修正，可以让平均数运算更加准确，来看看它是怎么运行的。

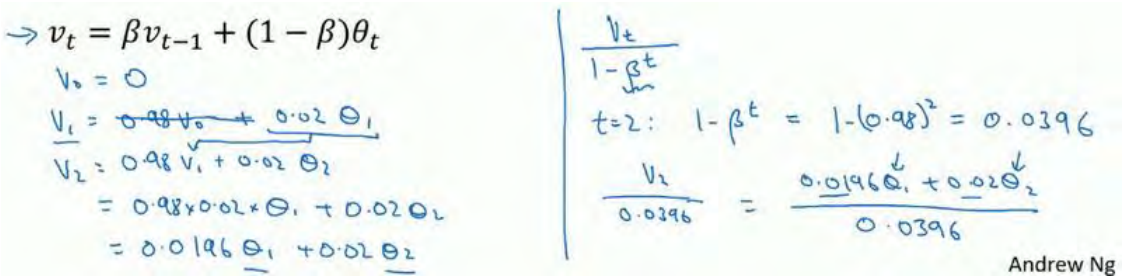


$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

在上一个视频中，这个（红色）曲线对应β的值为 0.9，这个（绿色）曲线对应的β=0.98，如果你执行写在这里的公式，在β等于 0.98 的时候，得到的并不是绿色曲线，而是紫色曲线，你可以注意到紫色曲线的起点较低，我们来看看怎么处理。

计算移动平均数的时候，初始化$v_0 = 0$，$v_1 = 0.98v_0 + 0.02\theta_1$，但是$v_0 = 0$，所以这部分没有了（$0.98v_0$），所以$v_1 = 0.02\theta_1$，所以如果一天温度是 40 华氏度，那么$v_1 = 0.02\theta_1 = 0.02 \times 40 = 8$，因此得到的值会小很多，所以第一天温度的估测不准。

$v_2 = 0.98v_1 + 0.02\theta_2$，如果代入$v_1$，然后相乘，所以$v_2 = 0.98 \times 0.02\theta_1 + 0.02\theta_2 = 0.0196\theta_1 + 0.02\theta_2$，假设$\theta_1$和$\theta_2$都是正数，计算后$v_2$要远小于$\theta_1$和$\theta_2$，所以$v_2$不能很好估测出这一年前两天的温度。



有个办法可以修改这一估测，让估测变得更好，更准确，特别是在估测初期，也就是不用v_t，而是用$\frac{v_t}{1-\beta^t}$，t 就是现在的天数。举个具体例子，当$t = 2$时，$1 - \beta^t = 1 - 0.98^2 = 0.0396$，

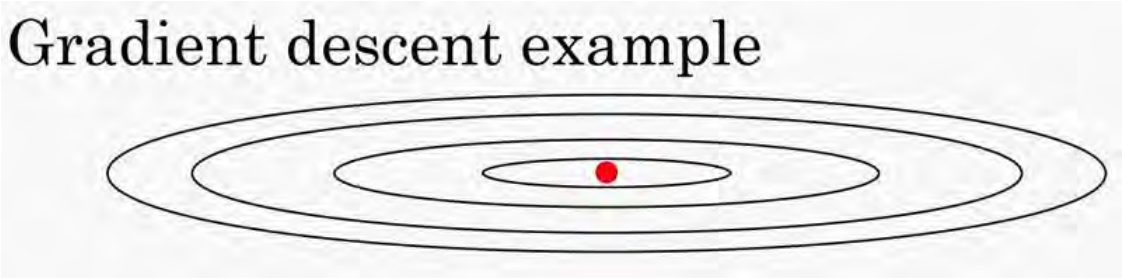
因此对第二天温度的估测变成了$\frac{v_2}{0.0396}=\frac{0.0196\theta_1+0.02\theta_2}{0.0396}$，也就是$\theta_1$和$\theta_2$的加权平均数，并去除了偏差。你会发现随着 t 增加，β^t接近于 0，所以当t很大的时候，偏差修正几乎没有作用，因此当t较大的时候，紫线基本和绿线重合了。不过在开始学习阶段，你才开始预测热身练习，偏差修正可以帮助你更好预测温度，偏差修正可以帮助你使结果从紫线变成绿线。

在机器学习中，在计算指数加权平均数的大部分时候，大家不在乎执行偏差修正，因为大部分人宁愿熬过初始时期，拿到具有偏差的估测，然后继续计算下去。如果你关心初始时期的偏差，在刚开始计算指数加权移动平均数的时候，偏差修正能帮助你在早期获取更好的估测。

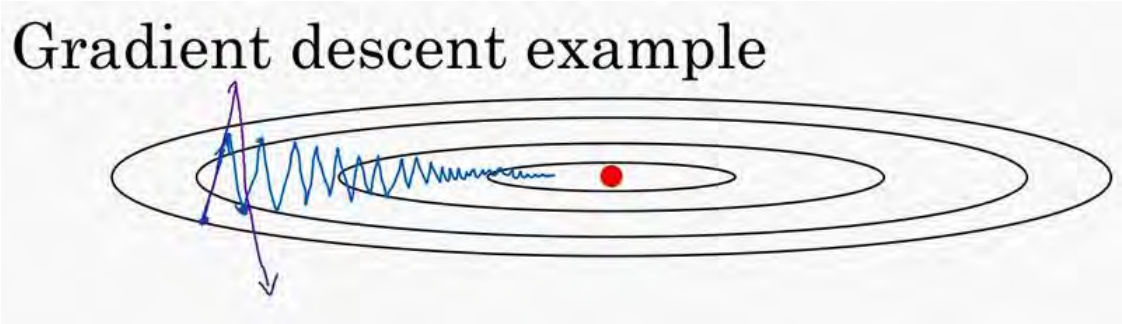
所以你学会了计算指数加权移动平均数，我们接着用它来构建更好的优化算法吧！

2.6 动量梯度下降法（Gradient descent with Momentum）

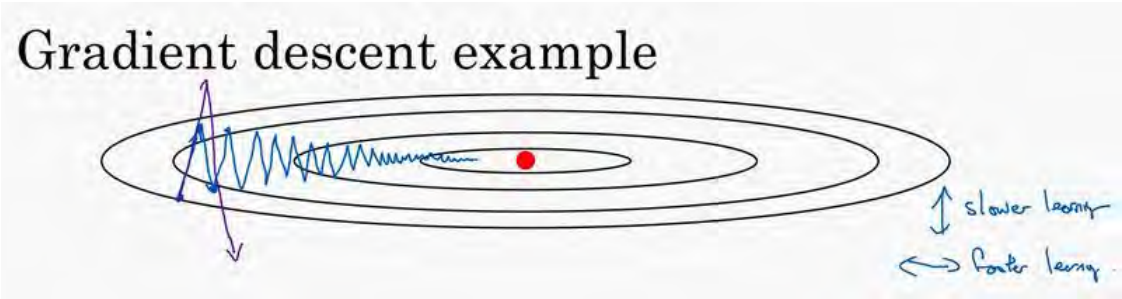
还有一种算法叫做 **Momentum**，或者叫做动量梯度下降法，运行速度几乎总是快于标准的梯度下降算法，简而言之，基本的想法就是计算梯度的指数加权平均数，并利用该梯度更新你的权重，在本视频中，我们呢要一起拆解单句描述，看看你到底如何计算。



例如，如果你要优化成本函数，函数形状如图，红点代表最小值的位置，假设你从这里（蓝色点）开始梯度下降法，如果进行梯度下降法的一次迭代，无论是 **batch** 或 **mini-batch** 下降法，也许会指向这里，现在在椭圆的另一边，计算下一步梯度下降，结果或许如此，然后再计算一步，再一步，计算下去，你会发现梯度下降法要很多计算步骤对吧？



慢慢摆动到最小值，这种上下波动减慢了梯度下降法的速度，你就无法使用更大的学习率，如果你要用较大的学习率（紫色箭头），结果可能会偏离函数的范围，为了避免摆动过大，你要用一个较小的学习率。



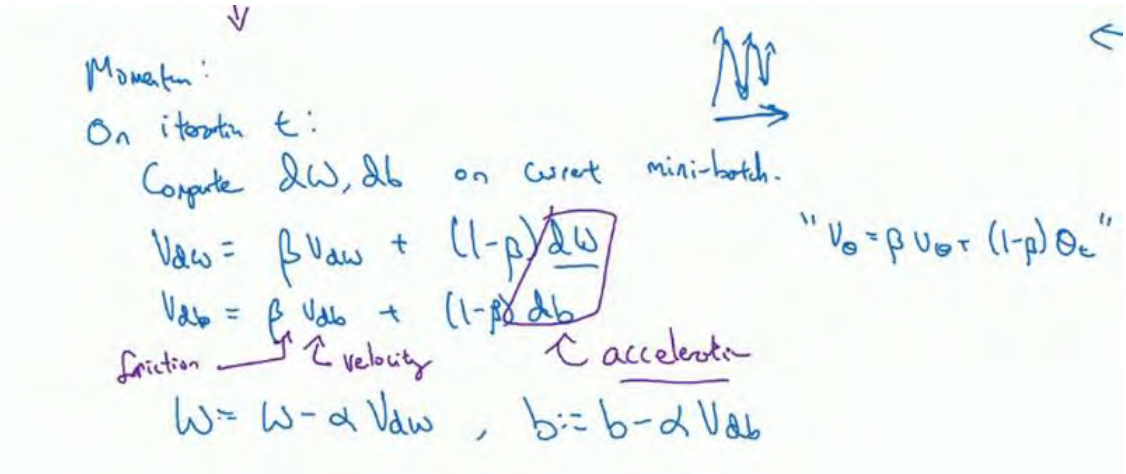
另一个看待问题的角度是，在纵轴上，你希望学习慢一点，因为你不想要这些摆动，但是在横轴上，你希望加快学习，你希望快速从左向右移，移向最小值，移向红点。所以使用

动量梯度下降法，你需要做的是，在每次迭代中，确切来说在第t次迭代的过程中，你会计算微分dW，db，我会省略上标$[l]$，你用现有的 **mini-batch** 计算dW，db。如果你用 **batch** 梯度下降法，现在的 **mini-batch** 就是全部的 **batch**，对于 **batch** 梯度下降法的效果是一样的。如果现有的 **mini-batch** 就是整个训练集，效果也不错，你要做的是计算$v_{dW} = \beta v_{dW} + (1-\beta)dW$，这跟我们之前的计算相似，也就是$v = \beta v + (1-\beta)\theta_t$，$dW$的移动平均数，接着同样地计算$v_{db}$，$v_{db} = \beta v_{db} + (1-\beta)db$，然后重新赋值权重，$W := W - av_{dW}$，同样$b := b - av_{db}$，这样就可以减缓梯度下降的幅度。

例如，在上几个导数中，你会发现这些纵轴上的摆动平均值接近于零，所以在纵轴方向，你希望放慢一点，平均过程中，正负数相互抵消，所以平均值接近于零。但在横轴方向，所有的微分都指向横轴方向，因此横轴方向的平均值仍然较大，因此用算法几次迭代后，你发现动量梯度下降法，最终纵轴方向的摆动变小了，横轴方向运动更快，因此你的算法走了一条更加直接的路径，在抵达最小值的路上减少了摆动。

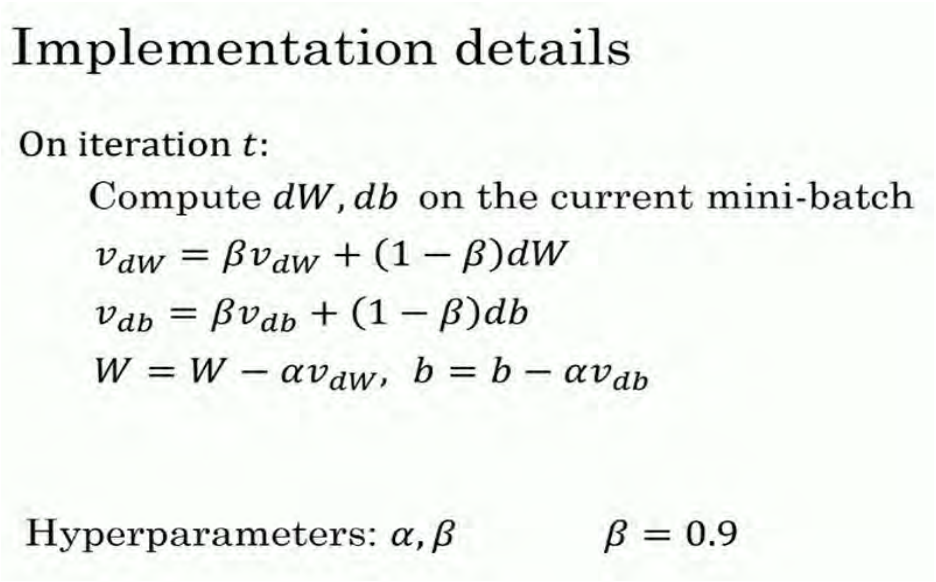
动量梯度下降法的一个本质，这对有些人而不是所有人有效，就是如果你要最小化碗状函数，这是碗的形状，我画的不太好。

它们能够最小化碗状函数，这些微分项，想象它们为你从山上往下滚的一个球，提供了加速度，**Momentum** 项相当于速度。

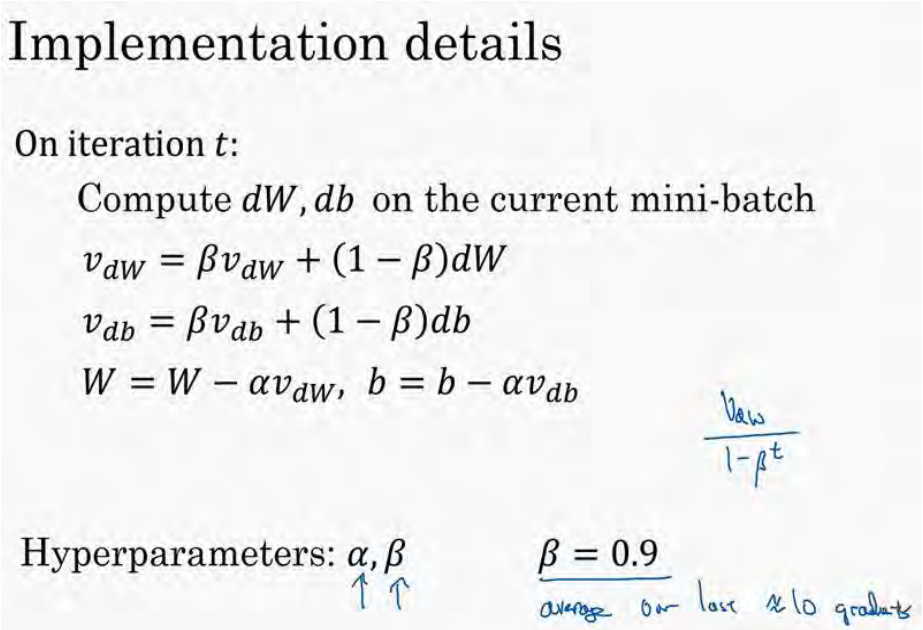


想象你有一个碗，你拿一个球，微分项给了这个球一个加速度，此时球正向山下滚，球因为加速度越滚越快，而因为β 稍小于 1，表现出一些摩擦力，所以球不会无限加速下去，所以不像梯度下降法，每一步都独立于之前的步骤，你的球可以向下滚，获得动量，可以从碗向下加速获得动量。我发现这个球从碗滚下的比喻，物理能力强的人接受得比较好，但不是所有人都能接受，如果球从碗中滚下这个比喻，你理解不了，别担心。

最后我们来看具体如何计算，算法在此。



所以你有两个超参数，学习率a以及参数β，β控制着指数加权平均数。β最常用的值是0.9，我们之前平均了过去十天的温度，所以现在平均了前十次迭代的梯度。实际上β为 0.9 时，效果不错，你可以尝试不同的值，可以做一些超参数的研究，不过 0.9 是很棒的鲁棒数。那么关于偏差修正，所以你要拿v_{dW}和v_{db}除以$1-\beta^t$，实际上人们不这么做，因为 10 次迭代之后，因为你的移动平均已经过了初始阶段。实际中，在使用梯度下降法或动量梯度下降法时，人们不会受到偏差修正的困扰。当然v_{dW}初始值是 0，要注意到这是和dW拥有相同维数的零矩阵，也就是跟W拥有相同的维数，v_{db}的初始值也是向量零，所以和db拥有相同的维数，也就是和b是同一维数。

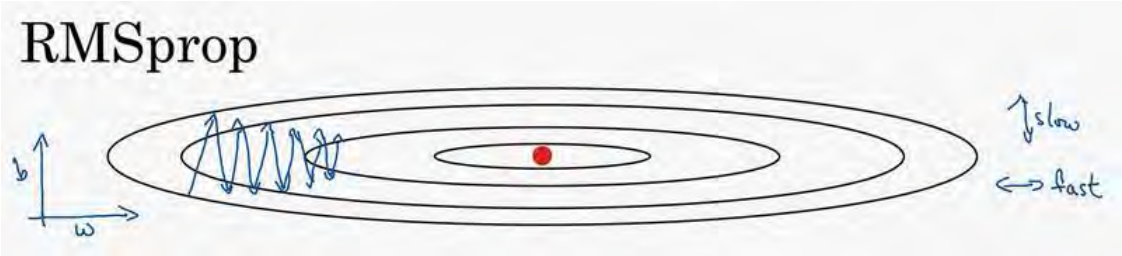


最后要说一点，如果你查阅了动量梯度下降法相关资料，你经常会看到一个被删除了的专业词汇，$1-\beta$被删除了，最后得到的是$v_{dW}=\beta v_{dW}+dW$。用紫色版本的结果就是，所以v_{dW}缩小了$1-\beta$倍，相当于乘以$\frac{1}{1-\beta}$，所以你要用梯度下降最新值的话，a要根据$\frac{1}{1-\beta}$相应变化。实际上，二者效果都不错，只会影响到学习率a的最佳值。我觉得这个公式用起来没有那么自然，因为有一个影响，如果你最后要调整超参数β，就会影响到v_{dW}和v_{db}，你也许还要修改学习率a，所以我更喜欢左边的公式，而不是删去了$1-\beta$的这个公式，所以我更倾向于使用左边的公式，也就是有$1-\beta$的这个公式，但是两个公式都将β设置为 0.9，是超参数的常见选择，只是在这两个公式中，学习率a的调整会有所不同。

所以这就是动量梯度下降法，这个算法肯定要好于没有 **Momentum** 的梯度下降算法，我们还可以做别的事情来加快学习算法，我们将在接下来的视频中探讨这些问题。

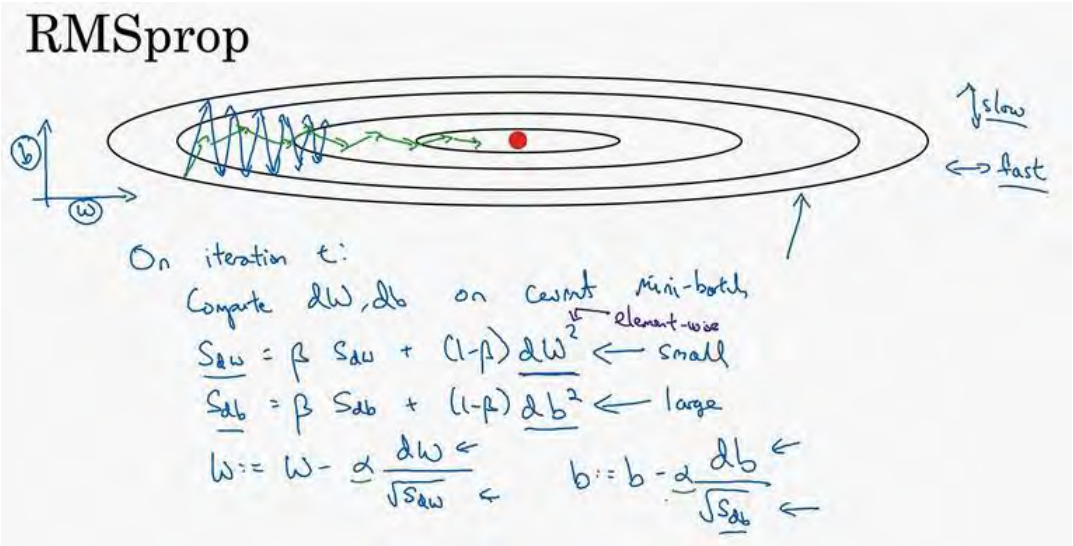
2.7 RMSprop

你们知道了动量（**Momentum**）可以加快梯度下降，还有一个叫做 **RMSprop** 的算法，全称是 **root mean square prop** 算法，它也可以加速梯度下降，我们来看看它是如何运作的。



回忆一下我们之前的例子，如果你执行梯度下降，虽然横轴方向正在推进，但纵轴方向会有大幅度摆动，为了分析这个例子，假设纵轴代表参数b，横轴代表参数W，可能有W_1，W_2或者其它重要的参数，为了便于理解，被称为b和W。

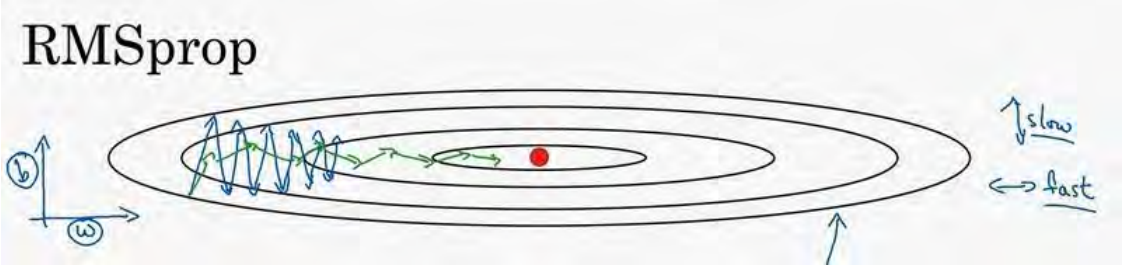
所以，你想减缓b方向的学习，即纵轴方向，同时加快，至少不是减缓横轴方向的学习，**RMSprop** 算法可以实现这一点。



在第t次迭代中，该算法会照常计算当下 **mini-batch** 的微分dW，db，所以我会保留这个指数加权平均数，我们用到新符号S_{dW}，而不是v_{dW}，因此$S_{dW} = \beta S_{dW} + (1-\beta) dW^2$，澄清一下，这个平方的操作是针对这一整个符号的，这样做能够保留微分平方的加权平均数，同样$S_{db} = \beta S_{db} + (1-\beta) db^2$，再说一次，平方是针对整个符号的操作。

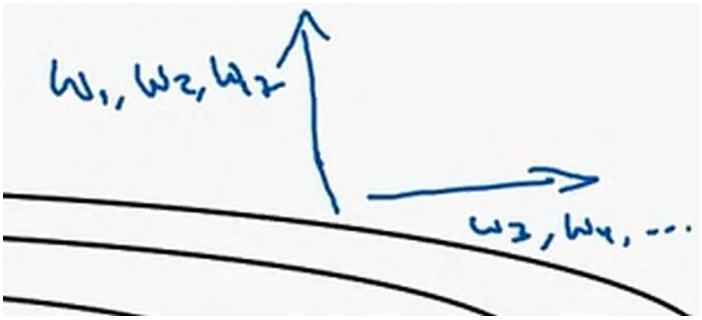
接着 **RMSprop** 会这样更新参数值，$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}}$，$b := b - \alpha \frac{db}{\sqrt{S_{db}}}$，我们来理解一下其原理。记得在横轴方向或者在例子中的W方向，我们希望学习速度快，而在垂直方向，也

就是例子中的b方向，我们希望减缓纵轴上的摆动，所以有了S_{dW}和S_{db}，我们希望S_{dW}会相对较小，所以我们要除以一个较小的数，而希望S_{db}又较大，所以这里我们要除以较大的数字，这样就可以减缓纵轴上的变化。你看这些微分，垂直方向的要比水平方向的大得多，所以斜率在b方向特别大，所以这些微分中，db较大，dW较小，因为函数的倾斜程度，在纵轴上，也就是 b 方向上要大于在横轴上，也就是W方向上。db的平方较大，所以S_{db}也会较大，而相比之下，dW会小一些，亦或dW平方会小一些，因此S_{dW}会小一些，结果就是纵轴上的更新要被一个较大的数相除，就能消除摆动，而水平方向的更新则被较小的数相除。



RMSprop 的影响就是你的更新最后会变成这样（绿色线），纵轴方向上摆动较小，而横轴方向继续推进。还有个影响就是，你可以用一个更大学习率a，然后加快学习，而无须在纵轴上垂直方向偏离。

要说明一点，我一直把纵轴和横轴方向分别称为b和W，只是为了方便展示而已。实际中，你会处于参数的高维度空间，所以需要消除摆动的垂直维度，你需要消除摆动，实际上是参数W_1，W_2等的合集，水平维度可能W_3，W_4等等，因此把W和b分开只是方便说明。实际中dW是一个高维度的参数向量，db也是一个高维度参数向量，但是你的直觉是，在你要消除摆动的维度中，最终你要计算一个更大的和值，这个平方和微分的加权平均值，所以你最后去掉了那些有摆动的方向。所以这就是 **RMSprop**，全称是均方根，因为你将微分进行平方，然后最后使用平方根。



最后再就这个算法说一些细节的东西，然后我们再继续。下一个视频中，我们会将 **RMSprop** 和 **Momentum** 结合起来，我们在 **Momentum** 中采用超参数β，为了避免混淆，我

们现在不用β，而采用超参数β_2以保证在 **Momentum** 和 **RMSprop** 中采用同一超参数。要确保你的算法不会除以 0，如果S_{dW}的平方根趋近于 0 怎么办？得到的答案就非常大，为了确保数值稳定，在实际操练的时候，你要在分母上加上一个很小很小的ε，ε是多少没关系，10^{-8}是个不错的选择，这只是保证数值能稳定一些，无论什么原因，你都不会除以一个很小很小的数。所以 **RMSprop** 跟 **Momentum** 有很相似的一点，可以消除梯度下降中的摆动，包括 **mini-batch** 梯度下降，并允许你使用一个更大的学习率a，从而加快你的算法学习速度。

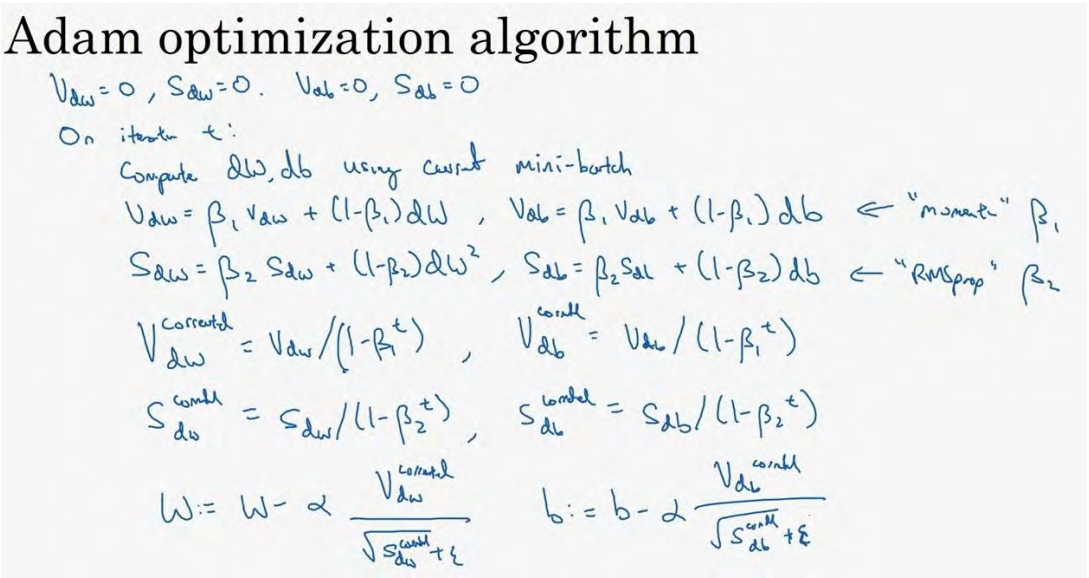
所以你学会了如何运用 **RMSprop**，这是给学习算法加速的另一方法。关于 **RMSprop** 的一个有趣的事是，它首次提出并不是在学术研究论文中，而是在多年前 **Jeff Hinton** 在 **Coursera** 的课程上。我想 **Coursera** 并不是故意打算成为一个传播新兴的学术研究的平台，但是却达到了意想不到的效果。就是从 **Coursera** 课程开始，**RMSprop** 开始被人们广为熟知，并且发展迅猛。

我们讲过了 **Momentum**，我们讲了 **RMSprop**，如果二者结合起来，你会得到一个更好的优化算法，在下个视频中我们再好好讲一讲为什么。

2.8 Adam 优化算法(Adam optimization algorithm)

在深度学习的历史上，包括许多知名研究者在内，提出了优化算法，并很好地解决了一些问题，但随后这些优化算法被指出并不能一般化，并不适用于多种神经网络，时间久了，深度学习圈子里的人开始多少有些质疑全新的优化算法，很多人都觉得动量（**Momentum**）梯度下降法很好用，很难再想出更好的优化算法。所以 **RMSprop** 以及 **Adam** 优化算法（**Adam** 优化算法也是本视频的内容），就是少有的经受住人们考验的两种算法，已被证明适用于不同的深度学习结构，这个算法我会毫不犹豫地推荐给你，因为很多人都试过，并且用它很好地解决了许多问题。

Adam 优化算法基本上就是将 **Momentum** 和 **RMSprop** 结合在一起，那么来看看如何使用 **Adam** 算法。



使用 **Adam** 算法，首先你要初始化，$v_{dW}=0$，$S_{dW}=0$，$v_{db}=0$，$S_{db}=0$，在第 t 次迭代中，你要计算微分，用当前的 **mini-batch** 计算dW，db，一般你会用 **mini-batch** 梯度下降法。接下来计算 **Momentum** 指数加权平均数，所以$v_{dW}=\beta_1 v_{dW}+(1-\beta_1)dW$（使用$\beta_1$，这样就不会跟超参数$\beta_2$混淆，因为后面 **RMSprop** 要用到$\beta_2$），使用 **Momentum** 时我们肯定会用这个公式，但现在不叫它β，而叫它β_1。同样$v_{db}=\beta_1 v_{db}+(1-\beta_1)db$。

接着你用 **RMSprop** 进行更新，即用不同的超参数β_2，$S_{dW}=\beta_2 S_{dW}+(1-\beta_2)(dW)^2$，再说一次，这里是对整个微分$dW$进行平方处理，$S_{db}=\beta_2 S_{db}+(1-\beta_2)(db)^2$。

相当于 **Momentum** 更新了超参数β_1，**RMSprop** 更新了超参数β_2。一般使用 **Adam** 算法的时候，要计算偏差修正，$v_{dW}^{\text{corrected}}$，修正也就是在偏差修正之后，

$v_{dW}^{\text{corrected}} = \frac{v_{dW}}{1-\beta_1^t}$，

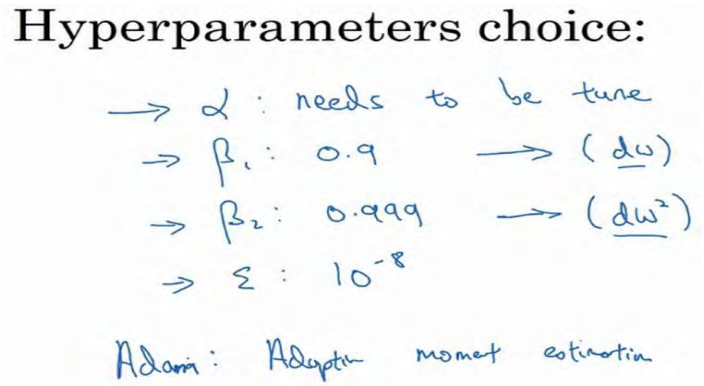
同样$v_{db}^{\text{corrected}} = \frac{v_{db}}{1-\beta_1^t}$，

S也使用偏差修正，也就是$S_{dW}^{\text{corrected}} = \frac{S_{dW}}{1-\beta_2^t}$，$S_{db}^{\text{corrected}} = \frac{S_{db}}{1-\beta_2^t}$。

最后更新权重，所以W更新后是$W := W - \frac{\alpha v_{dW}^{\text{corrected}}}{\sqrt{S_{dW}^{\text{corrected}}}+\varepsilon}$（如果你只是用 **Momentum**，使用$v_{dW}$或者修正后的$v_{dW}$，但现在我们加入了 **RMSprop** 的部分，所以我们要除以修正后S_{dW}的平方根加上ε）。

根据类似的公式更新b值，$b := b - \frac{\alpha v_{\text{db}}^{\text{corrected}}}{\sqrt{S_{\text{db}}^{\text{corrected}}}+\varepsilon}$。

所以 **Adam** 算法结合了 **Momentum** 和 **RMSprop** 梯度下降法，并且是一种极其常用的学习算法，被证明能有效适用于不同神经网络，适用于广泛的结构。



本算法中有很多超参数，超参数学习率α很重要，也经常需要调试，你可以尝试一系列值，然后看哪个有效。β_1常用的缺省值为 0.9，这是 dW 的移动平均数，也就是dW的加权平均数，这是 **Momentum** 涉及的项。至于超参数β_2，**Adam** 论文作者，也就是 **Adam** 算法的发明者，推荐使用 0.999，这是在计算$(dW)^2$以及$(db)^2$的移动加权平均值，关于ε的选择其实没那么重要，**Adam** 论文的作者建议ε为10^{-8}，但你并不需要设置它，因为它并不会影响算法表现。但是在使用 **Adam** 的时候，人们往往使用缺省值即可，β_1，β_2和ε都是如此，我觉得没人会去调整ε，然后尝试不同的α值，看看哪个效果最好。你也可以调整β_1和β_2，但我认识的业内人士很少这么干。

为什么这个算法叫做 **Adam**？**Adam** 代表的是 **Adaptive Moment Estimation**，β_1用于计算这个微分（dW），叫做第一矩，β_2用来计算平方数的指数加权平均数（$(dW)^2$），叫做第二矩，所以 **Adam** 的名字由此而来，但是大家都简称 **Adam** 权威算法。

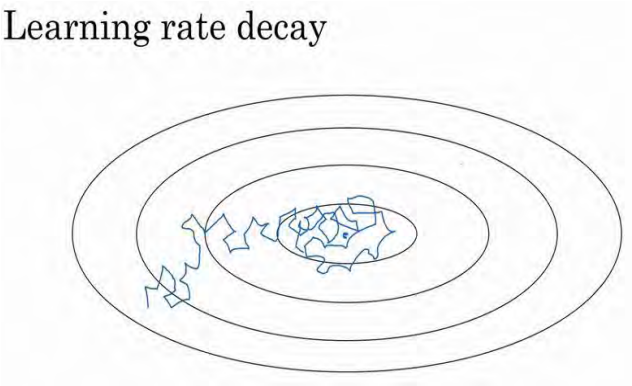
顺便提一下，我有一个老朋友兼合作伙伴叫做 **Adam Coates**。据我所知，他跟 **Adam** 算

法没有任何关系，不过我觉得他偶尔会用到这个算法，不过有时有人会问我这个问题，我想你可能也有相同的疑惑。

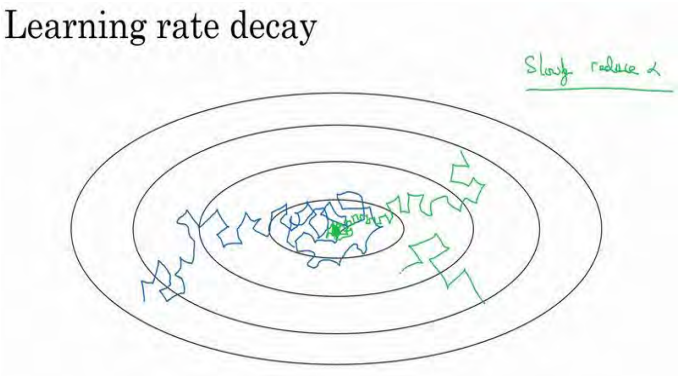
这就是关于 **Adam** 优化算法的全部内容，有了它，你可以更加快速地训练神经网络，在结束本周课程之前，我们还要讲一下超参数调整，以及更好地理解神经网络的优化问题有哪些。下个视频中，我们将讲讲学习率衰减。

2.9 学习率衰减(Learning rate decay)

加快学习算法的一个办法就是随时间慢慢减少学习率，我们将之称为学习率衰减，我们来看看如何做到，首先通过一个例子看看，为什么要计算学习率衰减。



假设你要使用 **mini-batch** 梯度下降法，**mini-batch** 数量不大，大概 64 或者 128 个样本，在迭代过程中会有噪音（蓝色线），下降朝向这里的最小值，但是不会精确地收敛，所以你的算法最后在附近摆动，并不会真正收敛，因为你用的a是固定值，不同的 **mini-batch** 中有噪音。

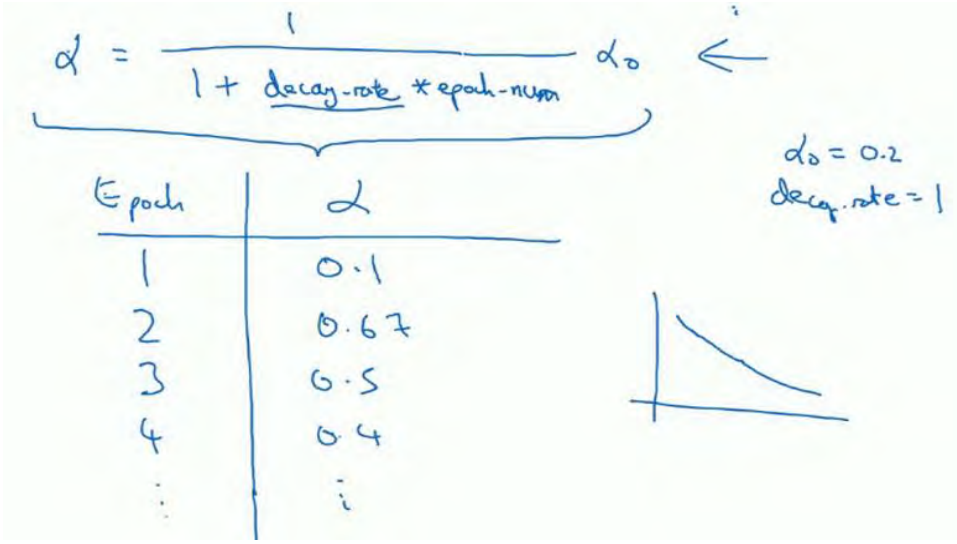


但要慢慢减少学习率a的话，在初期的时候，a学习率还较大，你的学习还是相对较快，但随着a变小，你的步伐也会变慢变小，所以最后你的曲线（绿色线）会在最小值附近的一小块区域里摆动，而不是在训练过程中，大幅度在最小值附近摆动。

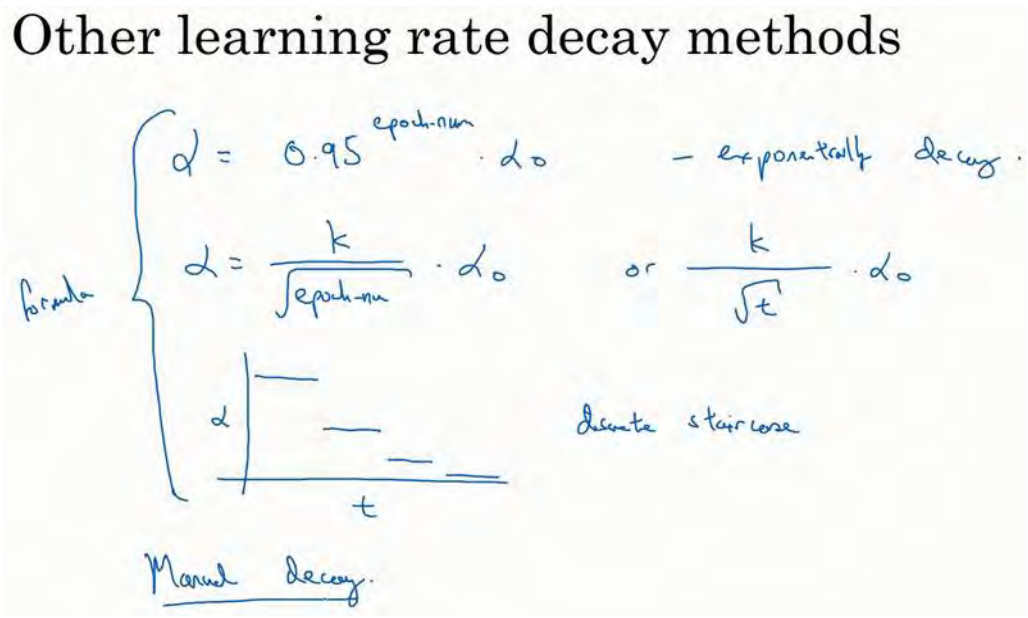
所以慢慢减少a的本质在于，在学习初期，你能承受较大的步伐，但当开始收敛的时候，小一些的学习率能让你步伐小一些。

你可以这样做到学习率衰减，记得一代要遍历一次数据，如果你有以下这样的训练集，

你应该拆分成不同的 **mini-batch**，第一次遍历训练集叫做第一代。第二次就是第二代，依此类推，你可以将a学习率设为$a=\frac{1}{1+decayrate*\text{epoch}-\text{num}}a_0$（**decay-rate** 称为衰减率，**epoch-num** 为代数，α_0为初始学习率），注意这个衰减率是另一个你需要调整的超参数。



这里有一个具体例子，如果你计算了几代，也就是历遍了几次，如果a_0为 0.2，衰减率 **decay-rate** 为 1，那么在第一代中，$a=\frac{1}{1+1}a_0=0.1$，这是在代入这个公式计算（$a=\frac{1}{1+decayrate*\text{epoch}-\text{num}}a_0$），此时衰减率是 1 而代数是 1。在第二代学习率为 0.67，第三代变成 0.5，第四代为 0.4 等等，你可以自己多计算几个数据。要理解，作为代数函数，根据上述公式，你的学习率呈递减趋势。如果你想用学习率衰减，要做的是要去尝试不同的值，包括超参数a_0，以及超参数衰退率，找到合适的值，除了这个学习率衰减的公式，人们还会用其它的公式。



比如，这个叫做指数衰减，其中a相当于一个小于 1 的值，如$a = 0.95^{\text{epoch-num}}a_0$，所以你的学习率呈指数下降。

人们用到的其它公式有$a = \frac{k}{\sqrt{\text{epoch-num}}}a_0$或者$a = \frac{k}{\sqrt{t}}a_0$（$t$为 **mini-batch** 的数字）。

有时人们也会用一个离散下降的学习率，也就是某个步骤有某个学习率，一会之后，学习率减少了一半，一会儿减少一半，一会儿又一半，这就是离散下降（**discrete stair cease**）的意思。

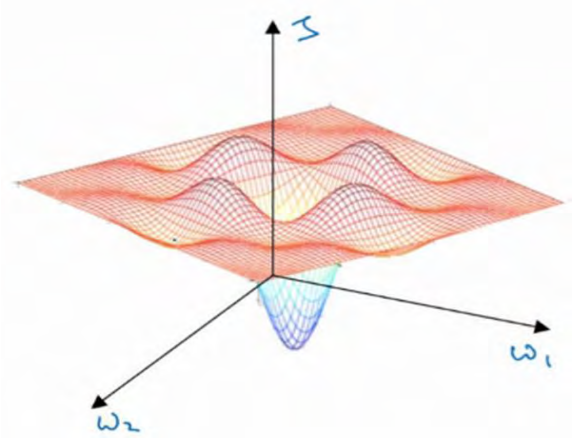
到现在，我们讲了一些公式，看学习率a究竟如何随时间变化。人们有时候还会做一件事，手动衰减。如果你一次只训练一个模型，如果你要花上数小时或数天来训练，有些人的确会这么做，看看自己的模型训练，耗上数日，然后他们觉得，学习速率变慢了，我把a调小一点。手动控制a当然有用，时复一时，日复一日地手动调整a，只有模型数量小的时候有用，但有时候人们也会这么做。

所以现在你有了多个选择来控制学习率a。你可能会想，好多超参数，究竟我应该做哪一个选择，我觉得，现在担心为时过早。下一周，我们会讲到，如何系统选择超参数。对我而言，学习率衰减并不是我尝试的要点，设定一个固定的a，然后好好调整，会有很大的影响，学习率衰减的确大有裨益，有时候可以加快训练，但它并不是我会率先尝试的内容，但下周我们将涉及超参数调整，你能学到更多系统的办法来管理所有的超参数，以及如何高效搜索超参数。

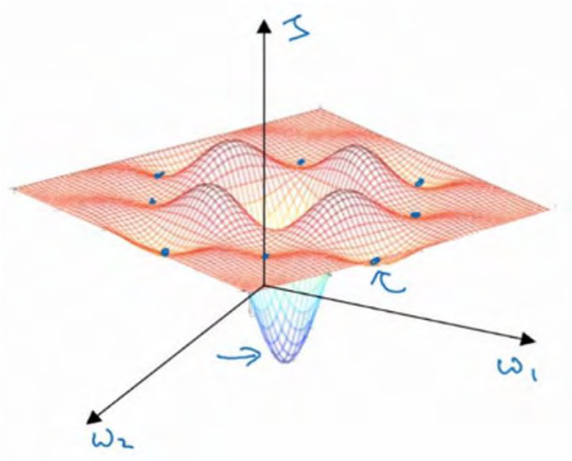
这就是学习率衰减，最后我还要讲讲神经网络中的局部最优以及鞍点，所以能更好理解在训练神经网络过程中，你的算法正在解决的优化问题，下个视频我们就好好聊聊这些问题。

2.10 局部最优的问题(The problem of local optima)

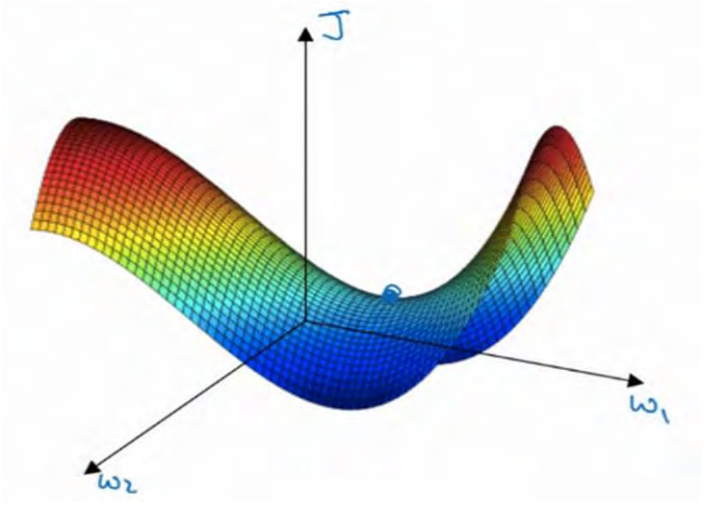
在深度学习研究早期，人们总是担心优化算法会困在极差的局部最优，不过随着深度学习理论不断发展，我们对局部最优的理解也发生了改变。我向你展示一下现在我们怎么看待局部最优以及深度学习中的优化问题。



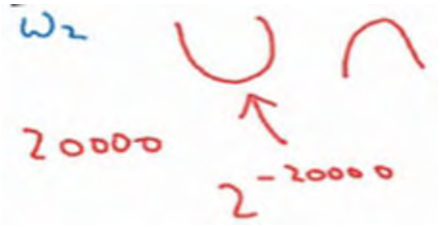
这是曾经人们在想到局部最优时脑海里会出现的图，也许你想优化一些参数，我们把它们称之为W_1和W_2，平面的高度就是损失函数。在图中似乎各处都分布着局部最优。梯度下降法或者某个算法可能困在一个局部最优中，而不会抵达全局最优。如果你要作图计算一个数字，比如说这两个维度，就容易出现有多个不同局部最优的图，而这些低维的图曾经影响了我们的理解，但是这些理解并不正确。事实上，如果你要创建一个神经网络，通常梯度为零的点并不是这个图中的局部最优点，实际上成本函数的零梯度点，通常是鞍点。



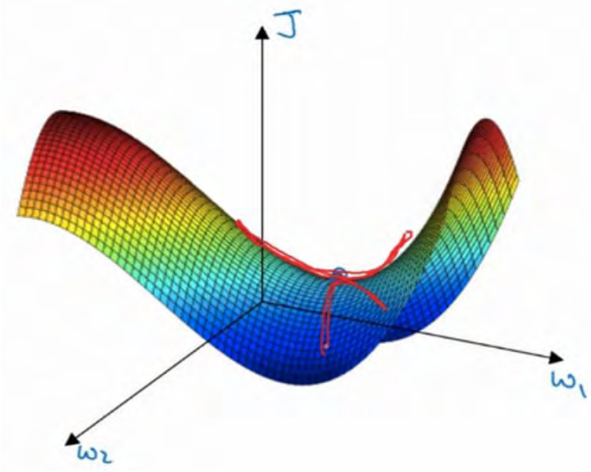
也就是在这个点，这里是W_1和W_2，高度即成本函数 J 的值。



但是一个具有高维度空间的函数，如果梯度为 0，那么在每个方向，它可能是凸函数，也可能是凹函数。如果你在 2 万维空间中，那么想要得到局部最优，所有的 2 万个方向都需要是这样，但发生的机率也许很小，也许是2^{-20000}，你更有可能遇到有些方向的曲线会这样向上弯曲，另一些方向曲线向下弯，而不是所有的都向上弯曲，因此在高维度空间，你更可能碰到鞍点。

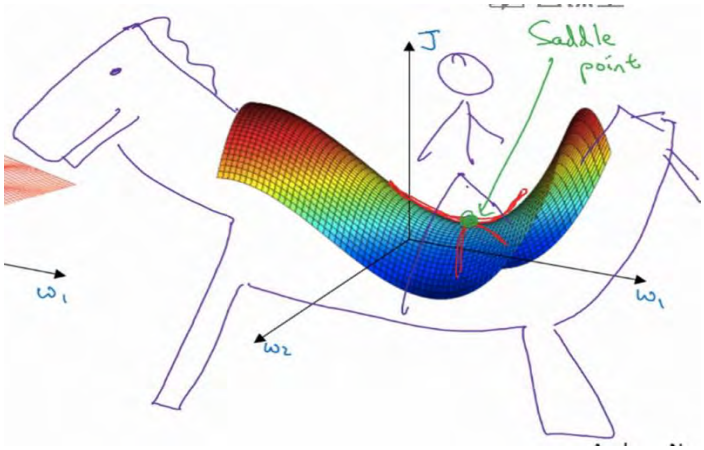


就像下面的这种：



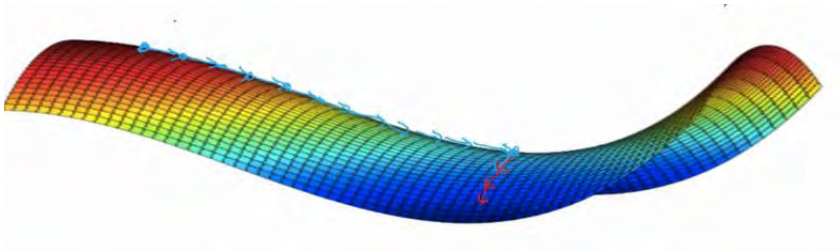
而不会碰到局部最优。至于为什么会把一个曲面叫做鞍点，你想象一下，就像是放在马背上的马鞍一样，如果这是马，这是马的头，这就是马的眼睛，画得不好请多包涵，然后你就是骑马的人，要坐在马鞍上，因此这里的这个点，导数为 0 的点，这个点叫做鞍点。我想

那确实是你坐在马鞍上的那个点，而这里导数为 0。

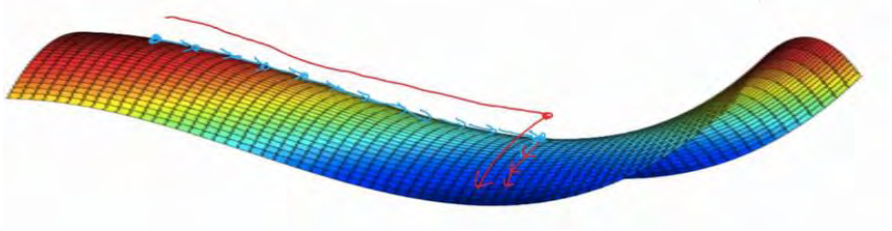


所以我们从深度学习历史中学到的一课就是，我们对低维度空间的大部分直觉，比如你可以画出上面的图，并不能应用到高维度空间中。适用于其它算法，因为如果你有 2 万个参数，那么J函数有 2 万个维度向量，你更可能遇到鞍点，而不是局部最优点。

如果局部最优不是问题，那么问题是什么？结果是平稳段会减缓学习，平稳段是一块区域，其中导数长时间接近于 0，如果你在此处，梯度会从曲面从从上向下下降，因为梯度等于或接近 0，曲面很平坦，你得花上很长时间慢慢抵达平稳段的这个点，因为左边或右边的随机扰动，我换个笔墨颜色，大家看得清楚一些，然后你的算法能够走出平稳段（红色笔）。



我们可以沿着这段长坡走，直到这里，然后走出平稳段。



所以此次视频的要点是，首先，你不太可能困在极差的局部最优中，条件是你在训练较大的神经网络，存在大量参数，并且成本函数J被定义在较高的维度空间。

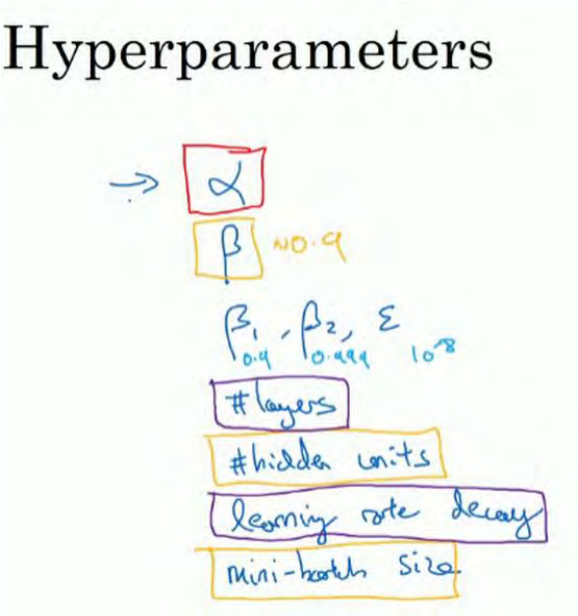
第二点，平稳段是一个问题，这样使得学习十分缓慢，这也是像 **Momentum** 或是 **RMSprop**，**Adam** 这样的算法，能够加速学习算法的地方。在这些情况下，更成熟的优化算法，如 **Adam** 算法，能够加快速度，让你尽早往下走出平稳段。

因为你的网络要解决优化问题，说实话，要面临如此之高的维度空间，我觉得没有人有那么好的直觉，知道这些空间长什么样，而且我们对它们的理解还在不断发展，不过我希望这一点能够让你更好地理解优化算法所面临的问题。

第三周 超参数调试、Batch 正则化和程序框架（Hyperparameter tuning）

3.1 调试处理（Tuning process）

大家好，欢迎回来，目前为止，你已经了解到，神经网络的改变会涉及到许多不同超参数的设置。现在，对于超参数而言，你要如何找到一套好的设定呢？在此视频中，我想和你分享一些指导原则，一些关于如何系统地组织超参调试过程的技巧，希望这些能够让你更有效的聚焦到合适的超参设定中。



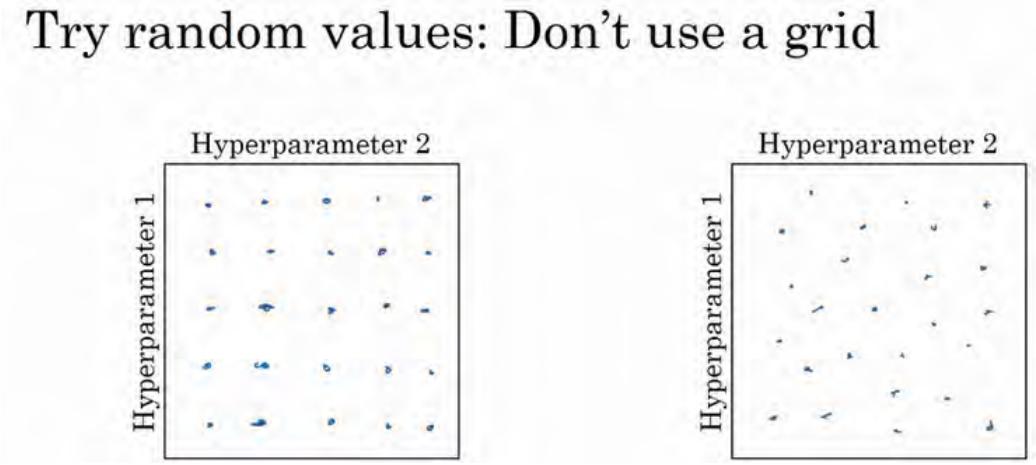
关于训练深度最难的事情之一是你要处理的参数的数量，从学习速率a到 **Momentum**（动量梯度下降法）的参数β。如果使用 **Momentum** 或 **Adam** 优化算法的参数，β_1，β_2和ε，也许你还得选择层数，也许你还得选择不同层中隐藏单元的数量，也许你还想使用学习率衰减。所以，你使用的不是单一的学习率a。接着，当然你可能还需要选择 **mini-batch** 的大小。

结果证实一些超参数比其它的更为重要，我认为，最为广泛的学习应用是a，学习速率是需要调试的最重要的超参数。

除了a，还有一些参数需要调试，例如 **Momentum** 参数β，0.9 就是个很好的默认值。我还会调试 **mini-batch** 的大小，以确保最优算法运行有效。我还会经常调试隐藏单元，我用橙色圈住的这些，这三个是我觉得其次比较重要的，相对于a而言。重要性排第三位的是其他因素，层数有时会产生很大的影响，学习率衰减也是如此。当应用 **Adam** 算法时，事实上，

我从不调试β_1，β_2和ε，我总是选定其分别为 0.9，0.999 和10^{-8}，如果你想的话也可以调试它们。

但希望你粗略了解到哪些超参数较为重要，a无疑是最重要的，接下来是我用橙色圈住的那些，然后是我用紫色圈住的那些，但这不是严格且快速的标准，我认为，其它深度学习的研究者可能会很不同意我的观点或有着不同的直觉。

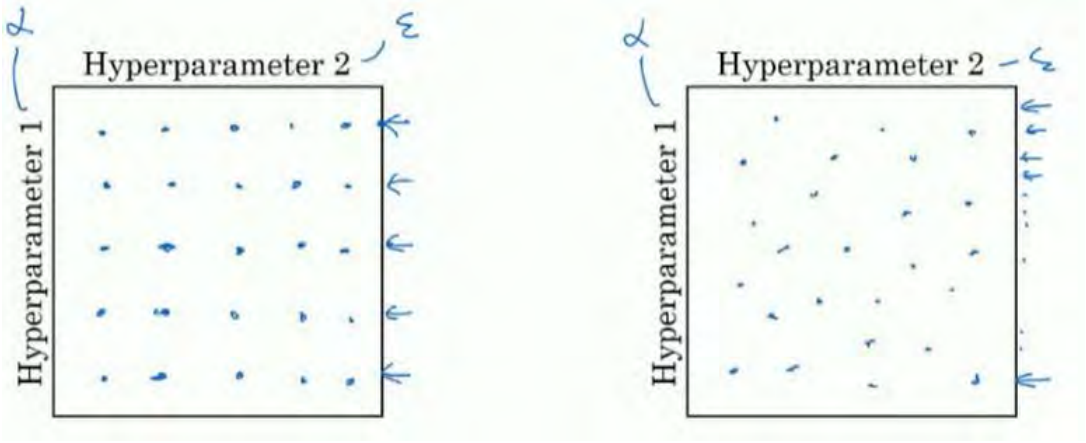


现在，如果你尝试调整一些超参数，该如何选择调试值呢？在早一代的机器学习算法中，如果你有两个超参数，这里我会称之为超参 1，超参 2，常见的做法是在网格中取样点，像这样，然后系统的研究这些数值。这里我放置的是 5×5 的网格，实践证明，网格可以是 5×5，也可多可少，但对于这个例子，你可以尝试这所有的 25 个点，然后选择哪个参数效果最好。当参数的数量相对较少时，这个方法很实用。

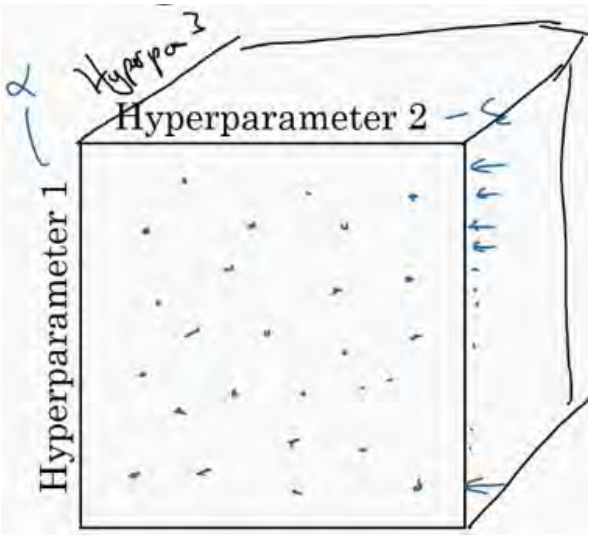
在深度学习领域，我们常做的，我推荐你采用下面的做法，随机选择点，所以你可以选择同等数量的点，对吗？25 个点，接着，用这些随机取的点试验超参数的效果。之所以这么做是因为，对于你要解决的问题而言，你很难提前知道哪个超参数最重要，正如你之前看到的，一些超参数的确要比其它的更重要。

举个例子，假设超参数 1 是a（学习速率），取一个极端的例子，假设超参数 2 是 **Adam** 算法中，分母中的ε。在这种情况下，a的取值很重要，而ε取值则无关紧要。如果你在网格中取点，接着，你试验了a的 5 个取值，那你会发现，无论ε取何值，结果基本上都是一样的。所以，你知道共有 25 种模型，但进行试验的a值只有 5 个，我认为这是很重要的。

对比而言，如果你随机取值，你会试验 25 个独立的a，似乎你更有可能发现效果做好的那个。

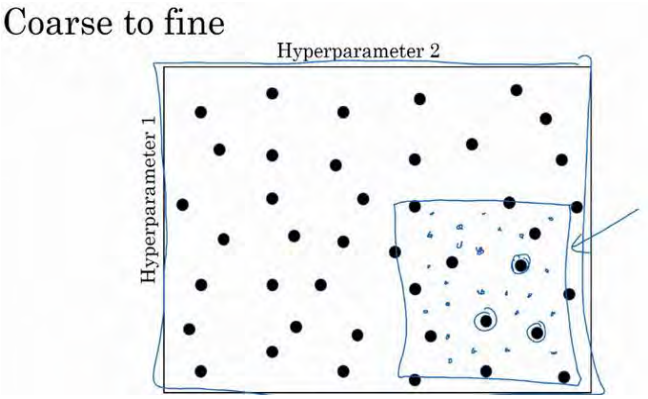


我已经解释了两个参数的情况，实践中，你搜索的超参数可能不止两个。假如，你有三个超参数，这时你搜索的不是一个方格，而是一个立方体，超参数 3 代表第三维，接着，在三维立方体中取值，你会试验大量的更多的值，三个超参数中每个都是。



实践中，你搜索的可能不止三个超参数有时很难预知，哪个是最重要的超参数，对于你的具体应用而言，随机取值而不是网格取值表明，你探究了更多重要超参数的潜在值，无论结果是什么。

当你给超参数取值时，另一个惯例是采用由粗糙到精细的策略。



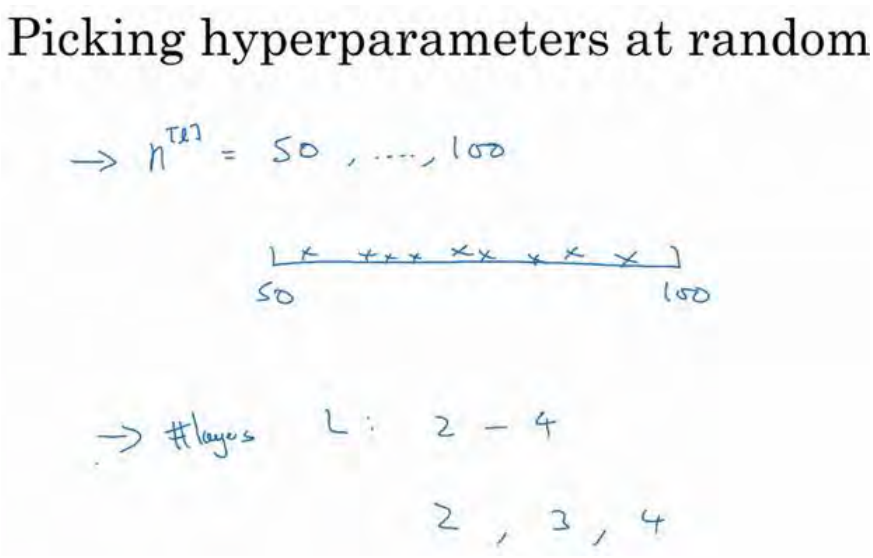
比如在二维的那个例子中，你进行了取值，也许你会发现效果最好的某个点，也许这个点周围的其他一些点效果也很好，那在接下来要做的是放大这块小区域（小蓝色方框内），然后在其中更密集得取值或随机取值，聚集更多的资源，在这个蓝色的方格中搜索，如果你怀疑这些超参数在这个区域的最优结果，那在整个的方格中进行粗略搜索后，你会知道接下来应该聚焦到更小的方格中。在更小的方格中，你可以更密集得取点。所以这种从粗到细的搜索也经常使用。

通过试验超参数的不同取值，你可以选择对训练集目标而言的最优值，或对于开发集而言的最优值，或在超参搜索过程中你最想优化的东西。

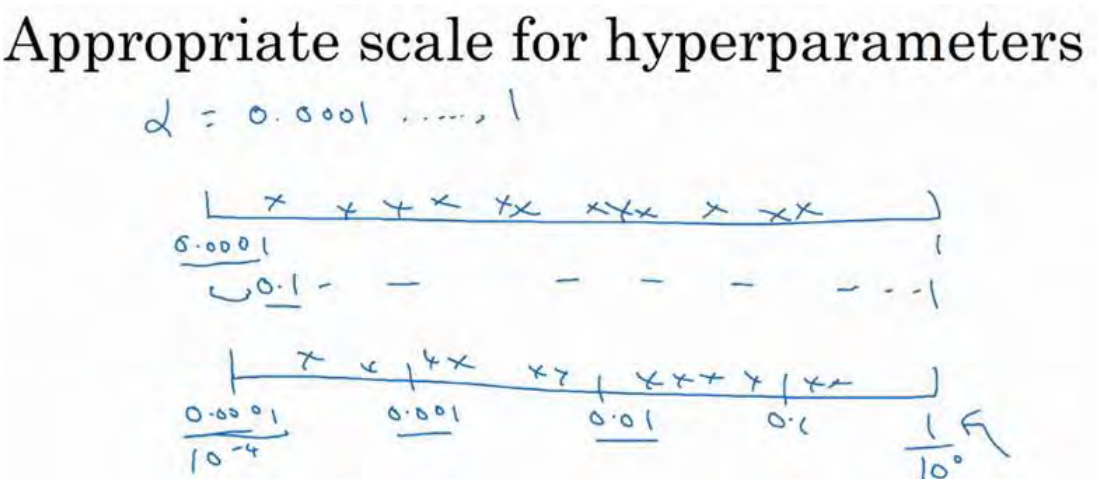
我希望，这能给你提供一种方法去系统地组织超参数搜索过程。另一个关键点是随机取值和精确搜索，考虑使用由粗糙到精细的搜索过程。但超参数的搜索内容还不止这些，在下一个视频中，我会继续讲解关于如何选择超参数取值的合理范围。

3.2 为超参数选择合适的范围（Using an appropriate scale to pick hyperparameters）

在上一个视频中，你已经看到了在超参数范围中，随机取值可以提升你的搜索效率。但随机取值并不是在有效范围内的随机均匀取值，而是选择合适的标尺，用于探究这些超参数，这很重要。在这个视频中，我会教你怎么做。

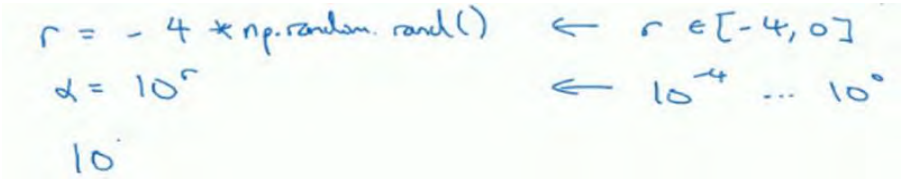


假设你要选取隐藏单元的数量$n^{[l]}$，假设，你选取的取值范围是从 50 到 100 中某点，这种情况下，看到这条从 50-100 的数轴，你可以随机在其取点，这是一个搜索特定超参数的很直观的方式。或者，如果你要选取神经网络的层数，我们称之为字母L，你也许会选择层数为 2 到 4 中的某个值，接着顺着 2，3，4 随机均匀取样才比较合理，你还可以应用网格搜索，你会觉得 2，3，4，这三个数值是合理的，这是在几个在你考虑范围内随机均匀取值的例子，这些取值还蛮合理的，但对某些超参数而言不适用。

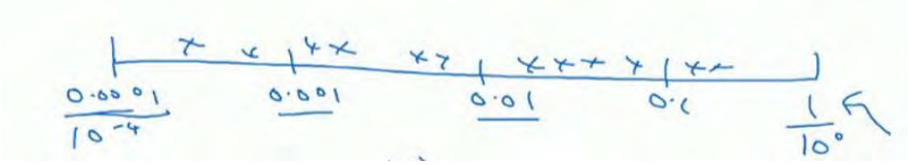


看看这个例子，假设你在搜索超参数a（学习速率），假设你怀疑其值最小是 0.0001 或最大是 1。如果你画一条从 0.0001 到 1 的数轴，沿其随机均匀取值，那 90%的数值将会落在 0.1 到 1 之间，结果就是，在 0.1 到 1 之间，应用了 90%的资源，而在 0.0001 到 0.1 之间，只有 10%的搜索资源，这看上去不太对。

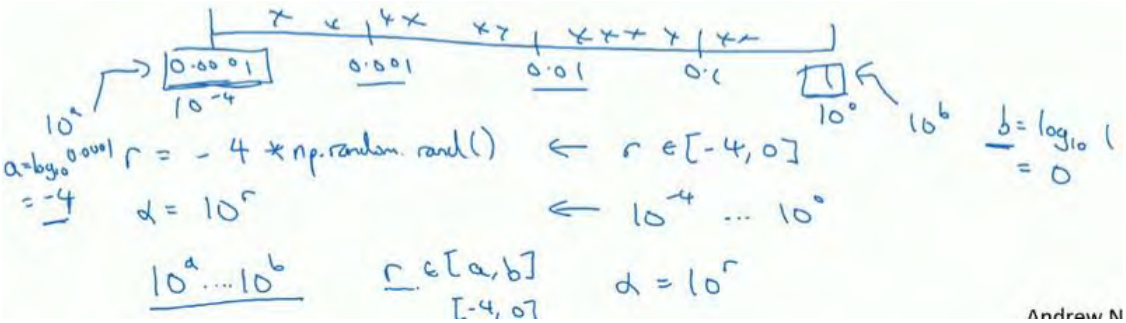
反而，用对数标尺搜索超参数的方式会更合理，因此这里不使用线性轴，分别依次取 0.0001，0.001，0.01，0.1，1，在对数轴上均匀随机取点，这样，在 0.0001 到 0.001 之间，就会有更多的搜索资源可用，还有在 0.001 到 0.01 之间等等。



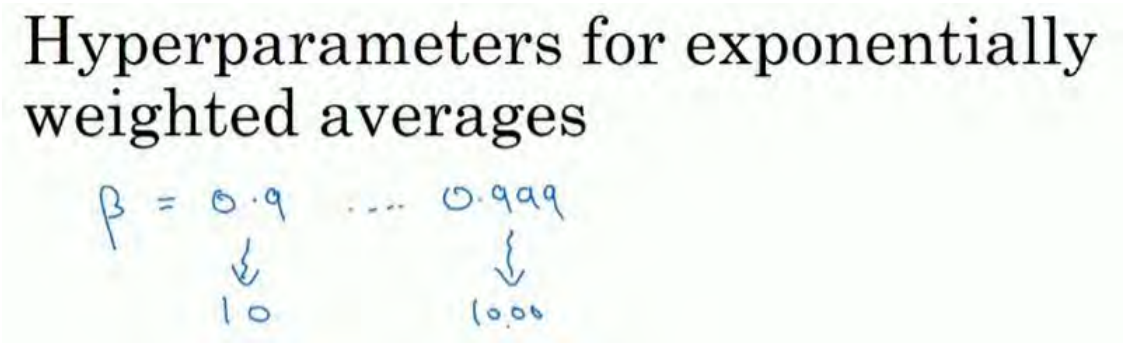
所以在 **Python** 中，你可以这样做，使 `r=-4*np.random.rand()`，然后a随机取值，$a = 10^r$，所以，第一行可以得出$r \in [4,0]$，那么$a \in [10^{-4}, 10^0]$，所以最左边的数字是10^{-4}，最右边是10^0。



更常见的情况是，如果你在10^a和10^b之间取值，在此例中，这是10^a（0.0001），你可以通过0.0001算出a的值，即-4，在右边的值是10^b，你可以算出b的值1，即 0。你要做的就是在$[a, b]$区间随机均匀地给r取值，这个例子中$r \in [-4,0]$，然后你可以设置a的值，基于随机取样的超参数$a = 10^r$。

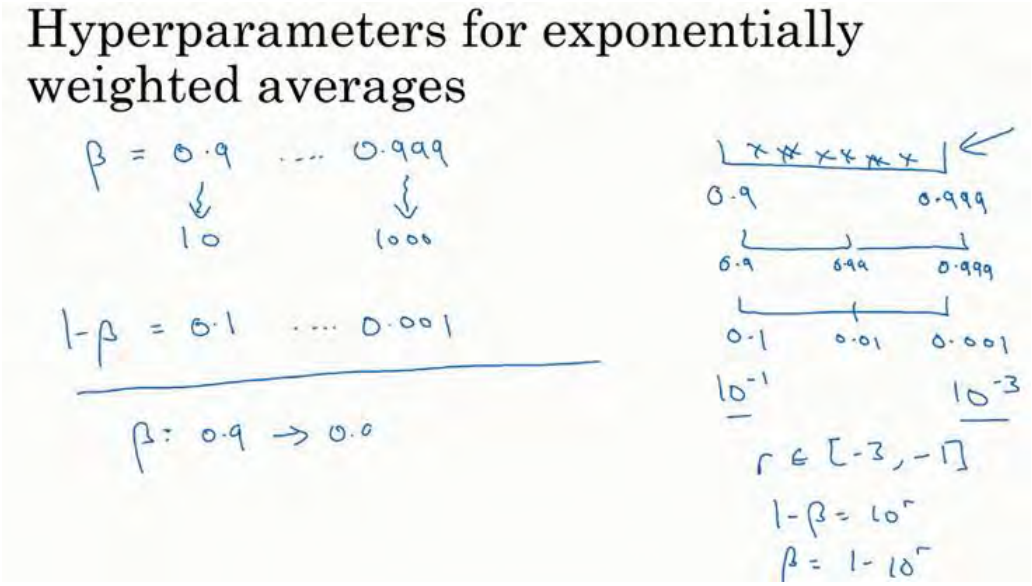


所以总结一下，在对数坐标下取值，取最小值的对数就得到a的值，取最大值的对数就得到b值，所以现在你在对数轴上的10^a到10^b区间取值，在a，b间随意均匀的选取r值，将超参数设置为10^r，这就是在对数轴上取值的过程。



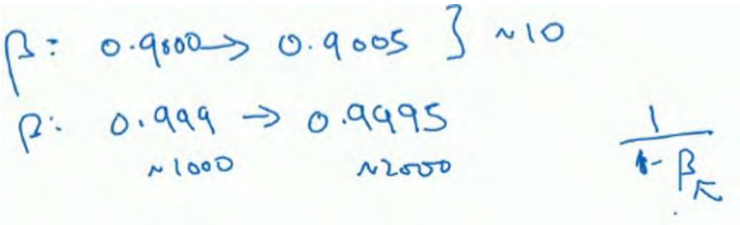
最后，另一个棘手的例子是给 β 取值，用于计算指数的加权平均值。假设你认为β是 0.9 到 0.999 之间的某个值，也许这就是你想搜索的范围。记住这一点，当计算指数的加权平均值时，取 0.9 就像在 10 个值中计算平均值，有点类似于计算 10 天的温度平均值，而取 0.999 就是在 1000 个值中取平均。

所以和上张幻灯片上的内容类似，如果你想在 0.9 到 0.999 区间搜索，那就不能用线性轴取值，对吧？不要随机均匀在此区间取值，所以考虑这个问题最好的方法就是，我们要探究的是$1-\beta$，此值在 0.1 到 0.001 区间内，所以我们会给$1-\beta$取值，大概是从 0.1 到 0.001，应用之前幻灯片中介绍的方法，这是10^{-1}，这是10^{-3}，值得注意的是，在之前的幻灯片里，我们把最小值写在左边，最大值写在右边，但在这里，我们颠倒了大小。这里，左边的是最大值，右边的是最小值。所以你要做的就是在$[-3,-1]$里随机均匀的给 r 取值。你设定了$1-\beta=10^{r}$，所以$\beta=1-10^{r}$，然后这就变成了在特定的选择范围内超参数随机取值。希望用这种方式得到想要的结果，你在 0.9 到 0.99 区间探究的资源，和在 0.99 到 0.999 区间探究的一样多。



所以，如果你想研究更多正式的数学证明，关于为什么我们要这样做，为什么用线性轴

取值不是个好办法，这是因为当β 接近 1 时，所得结果的灵敏度会变化，即使β有微小的变化。所以β 在 0.9 到 0.9005 之间取值，无关紧要，你的结果几乎不会变化。



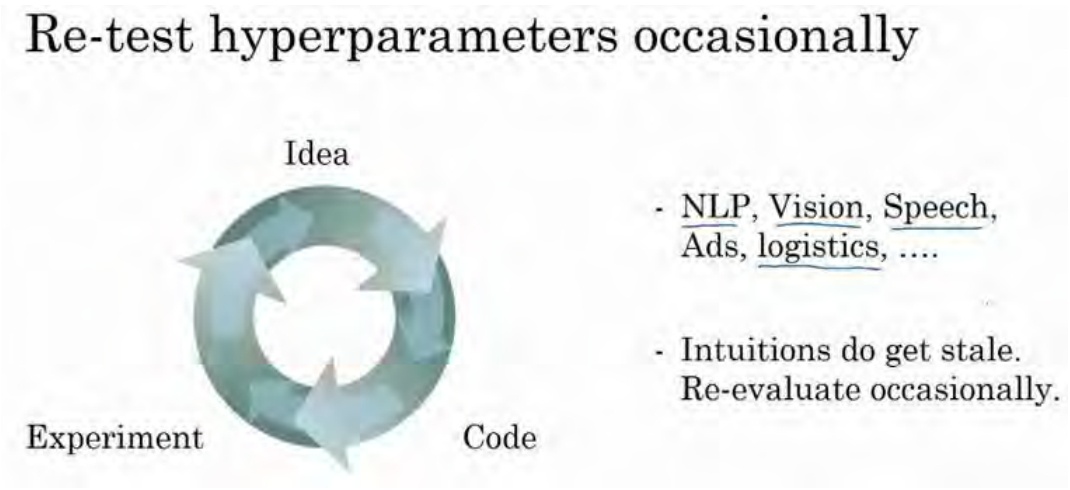
但β值如果在 0.999 到 0.9995 之间，这会对你的算法产生巨大影响，对吧？在这两种情况下，是根据大概 10 个值取平均。但这里，它是指数的加权平均值，基于 1000 个值，现在是 2000 个值，因为这个公式$\frac{1}{1-\beta}$，当β接近 1 时，β就会对细微的变化变得很敏感。所以整个取值过程中，你需要更加密集地取值，在β 接近 1 的区间内，或者说，当$1-\beta$ 接近于 0 时，这样，你就可以更加有效的分布取样点，更有效率的探究可能的结果。

希望能帮助你选择合适的标尺，来给超参数取值。如果你没有在超参数选择中作出正确的标尺决定，别担心，即使你在均匀的标尺上取值，如果数值总量较多的话，你也会得到还不错的结果，尤其是应用从粗到细的搜索方法，在之后的迭代中，你还是会聚焦到有用的超参数取值范围上。

希望这会对你的超参数搜索有帮助，下一个视频中，我们将会分享一些关于如何组建搜索过程的思考，希望它能使你的工作更高效。

3.3 超参数训练的实践：Pandas VS Caviar（Hyperparameters tuning in practice: Pandas vs. Caviar）

到现在为止，你已经听了许多关于如何搜索最优超参数的内容，在结束我们关于超参数搜索的讨论之前，我想最后和你分享一些建议和技巧，关于如何组织你的超参数搜索过程。



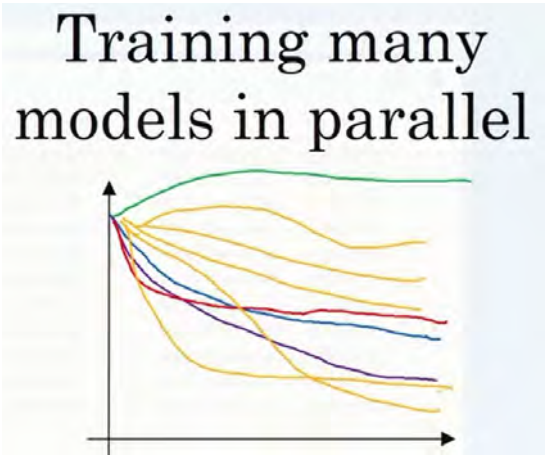
如今的深度学习已经应用到许多不同的领域，某个应用领域的超参数设定，有可能通用于另一领域，不同的应用领域出现相互交融。比如，我曾经看到过计算机视觉领域中涌现的巧妙方法，比如说 **Confonets** 或 **ResNets**，这我们会在后续课程中讲到。它还成功应用于语音识别，我还看到过最初起源于语音识别的想法成功应用于 **NLP** 等等。

深度学习领域中，发展很好的一点是，不同应用领域的人们会阅读越来越多其它研究领域的文章，跨领域去寻找灵感。

就超参数的设定而言，我见到过有些直觉想法变得很缺乏新意，所以，即使你只研究一个问题，比如说逻辑学，你也许已经找到一组很好的参数设置，并继续发展算法，或许在几个月的过程中，观察到你的数据会逐渐改变，或也许只是在你的数据中心更新了服务器，正因为有了这些变化，你原来的超参数的设定不再好用，所以我建议，或许只是重新测试或评估你的超参数，至少每隔几个月一次，以确保你对数值依然很满意。

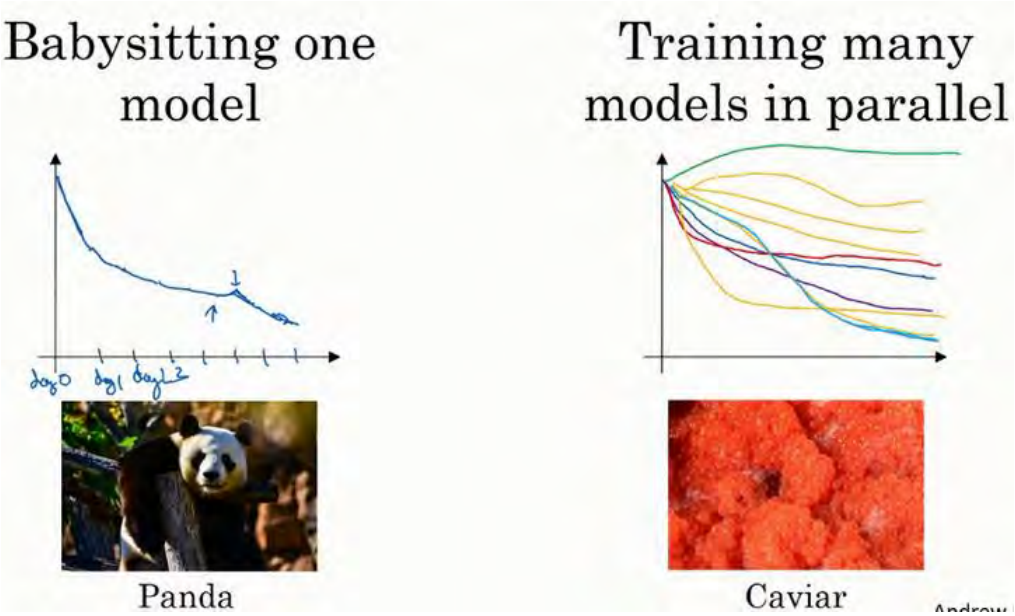
最后，关于如何搜索超参数的问题，我见过大概两种重要的思想流派或人们通常采用的两种重要但不同的方式。

一种是你照看一个模型，通常是有庞大的数据组，但没有许多计算资源或足够的 **CPU** 和 **GPU** 的前提下，基本而言，你只可以一次负担起试验一个模型或一小批模型，在这种情况下，即使当它在试验时，你也可以逐渐改良。比如，第 0 天，你将随机参数初始化，然后开始试验，然后你逐渐观察自己的学习曲线，也许是损失函数 J，或者数据设置误差或其它的东西，在第 1 天内逐渐减少，那这一天末的时候，你可能会说，看，它学习得真不错。我试着增加一点学习速率，看看它会怎样，也许结果证明它做得更好，那是你第二天的表现。两天后，你会说，它依旧做得不错，也许我现在可以填充下 **Momentum** 或减少变量。然后进入第三天，每天，你都会观察它，不断调整你的参数。也许有一天，你会发现你的学习率太大了，所以你可能又回归之前的模型，像这样，但你可以说是在每天花时间照看此模型，即使是它在许多天或许多星期的试验过程中。所以这是一个人们照料一个模型的方法，观察它的表现，耐心地调试学习率，但那通常是因为你没有足够的计算能力，不能在同一时间试验大量模型时才采取的办法。



另一种方法则是同时试验多种模型，你设置了一些超参数，尽管让它自己运行，或者是一天甚至多天，然后你会获得像这样的学习曲线，这可以是损失函数 J 或实验误差或损失或

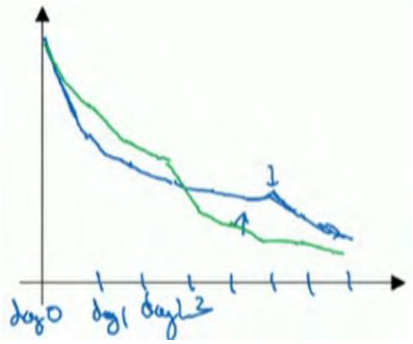
数据误差的损失，但都是你曲线轨迹的度量。同时你可以开始一个有着不同超参数设定的不同模型，所以，你的第二个模型会生成一个不同的学习曲线，也许是像这样的一条（紫色曲线），我会说这条看起来更好些。与此同时，你可以试验第三种模型，其可能产生一条像这样的学习曲线（红色曲线），还有另一条（绿色曲线），也许这条有所偏离，像这样，等等。或者你可以同时平行试验许多不同的模型，橙色的线就是不同的模型。用这种方式你可以试验许多不同的参数设定，然后只是最后快速选择工作效果最好的那个。在这个例子中，也许这条看起来是最好的（下方绿色曲线）。



打个比方，我把左边的方法称为熊猫方式。当熊猫有了孩子，他们的孩子非常少，一次通常只有一个，然后他们花费很多精力抚养熊猫宝宝以确保其能成活，所以，这的确是一种照料，一种模型类似于一只熊猫宝宝。对比而言，右边的方式更像鱼类的行为，我称之为鱼子酱方式。在交配季节，有些鱼类会产下一亿颗卵，但鱼类繁殖的方式是，它们会产生很多卵，但不对其中任何一个多加照料，只是希望其中一个，或其中一群，能够表现出色。我猜，这就是哺乳动物繁衍和鱼类，很多爬虫类动物繁衍的区别。我将称之为熊猫方式与鱼子酱方式，因为这很有趣，更容易记住。

所以这两种方式的选择，是由你拥有的计算资源决定的，如果你拥有足够的计算机去平行试验许多模型，那绝对采用鱼子酱方式，尝试许多不同的超参数，看效果怎么样。但在一些应用领域，比如在线广告设置和计算机视觉应用领域，那里的数据太多了，你需要试验大量的模型，所以同时试验大量的模型是很困难的，它的确是依赖于应用的过程。但我看到那些应用熊猫方式多一些的组织，那里，你会像对婴儿一样照看一个模型，调试参数，试着让

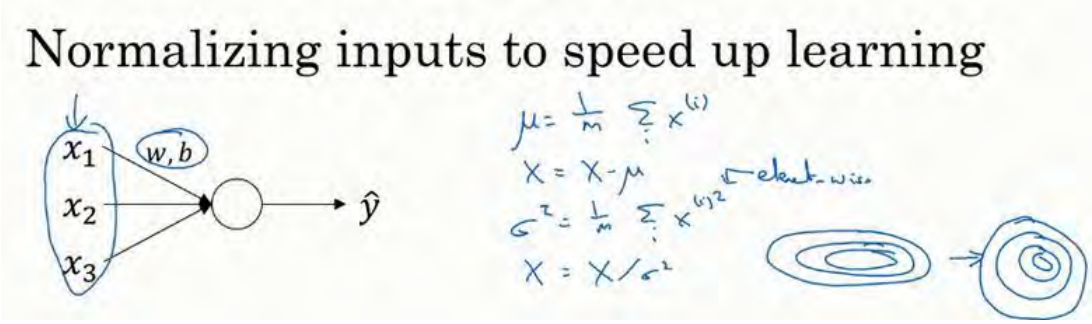
它工作运转。尽管，当然，甚至是在熊猫方式中，试验一个模型，观察它工作与否，也许第二或第三个星期后，也许我应该建立一个不同的模型（绿色曲线），像熊猫那样照料它，我猜，这样一生中可以培育几个孩子，即使它们一次只有一个孩子或孩子的数量很少。



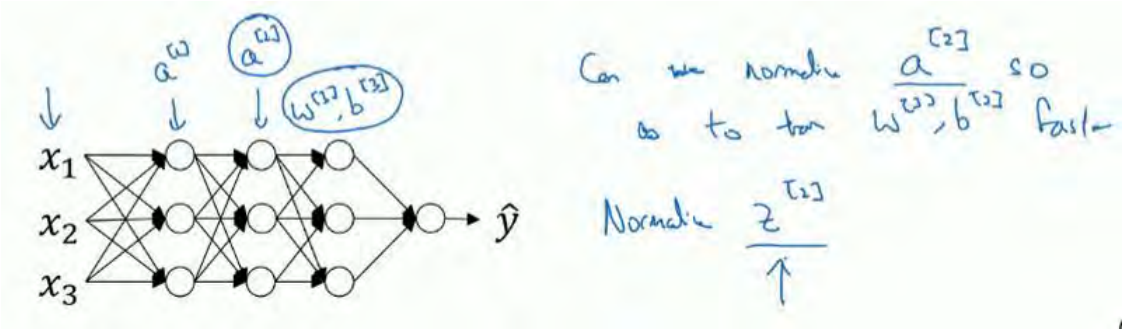
所以希望你能学会如何进行超参数的搜索过程，现在，还有另一种技巧，能使你的神经网络变得更加坚实，它并不是对所有的神经网络都适用，但当适用时，它可以使超参数搜索变得容易许多并加速试验过程，我们在下个视频中再讲解这个技巧。

3.4 归一化网络的激活函数（Normalizing activations in a network）

在深度学习兴起后，最重要的一个思想是它的一种算法，叫做 **Batch** 归一化，由 **Sergey Ioffe** 和 **Christian Szegedy** 两位研究者创造。**Batch** 归一化会使你的参数搜索问题变得很容易，使神经网络对超参数的选择更加稳定，超参数的范围会更加庞大，工作效果也很好，也会是你的训练更加容易，甚至是深层网络。让我们来看看 **Batch** 归一化是怎么起作用的吧。



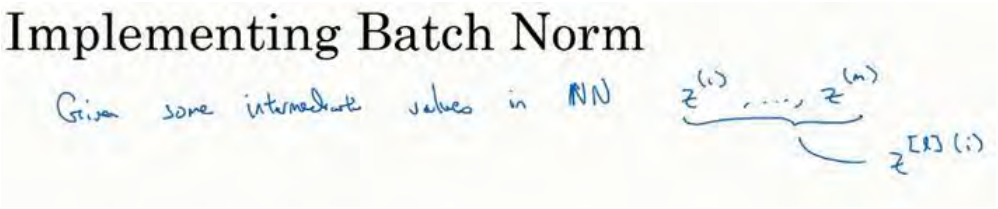
当训练一个模型，比如 **logistic** 回归时，你也许会记得，归一化输入特征可以加快学习过程。你计算了平均值，从训练集中减去平均值，计算了方差，接着根据方差归一化你的数据集，在之前的视频中我们看到，这是如何把学习问题的轮廓，从很长的东西，变成更圆的东西，更易于算法优化。所以这是有效的，对 **logistic** 回归和神经网络的归一化输入特征值而言。



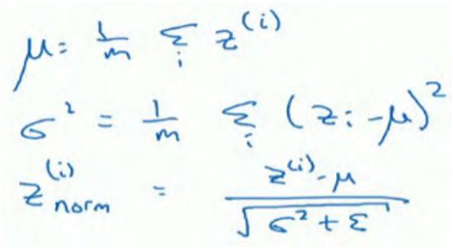
那么更深的模型呢？你不仅输入了特征值 x，而且这层有激活值$a^{[1]}$，这层有激活值$a^{[2]}$等等。如果你想训练这些参数，比如$w^{[3]}$，$b^{[3]}$，那归一化$a^{[2]}$的平均值和方差岂不是很好？以便使$w^{[3]}$，$b^{[3]}$的训练更有效率。在 **logistic** 回归的例子中，我们看到了如何归一化x_1，x_2，x_3，会帮助你更有效的训练w和b。

所以问题来了，对任何一个隐藏层而言，我们能否归一化a值，在此例中，比如说$a^{[2]}$的值，但可以是任何隐藏层的，以更快的速度训练$w^{[3]}$，$b^{[3]}$，因为$a^{[2]}$是下一层的输入值，所

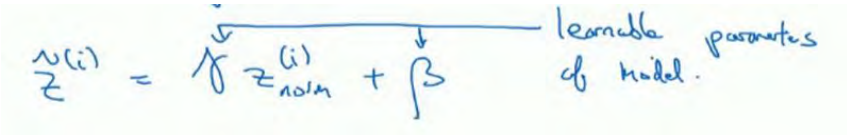
以就会影响$w^{[3]}$，$b^{[3]}$的训练。简单来说，这就是 **Batch** 归一化的作用。尽管严格来说，我们真正归一化的不是$a^{[2]}$，而是$z^{[2]}$，深度学习文献中有一些争论，关于在激活函数之前是否应该将值$z^{[2]}$归一化，或是否应该在应用激活函数$a^{[2]}$后再规范值。实践中，经常做的是归一化$z^{[2]}$，所以这就是我介绍的版本，我推荐其为默认选择，那下面就是 **Batch** 归一化的使用方法。



在神经网络中，已知一些中间值，假设你有一些隐藏单元值，从$z^{(1)}$到$z^{(m)}$，这些来源于隐藏层，所以这样写会更准确，即$z^{[l](i)}$为隐藏层，i从 1 到m，但这样书写，我要省略l及方括号，以便简化这一行的符号。所以已知这些值，如下，你要计算平均值，强调一下，所有这些都是针对l层，但我省略l及方括号，然后用正如你常用的那个公式计算方差，接着，你会取每个$z^{(i)}$值，使其规范化，方法如下，减去均值再除以标准偏差，为了使数值稳定，通常将ε作为分母，以防$\sigma = 0$的情况。



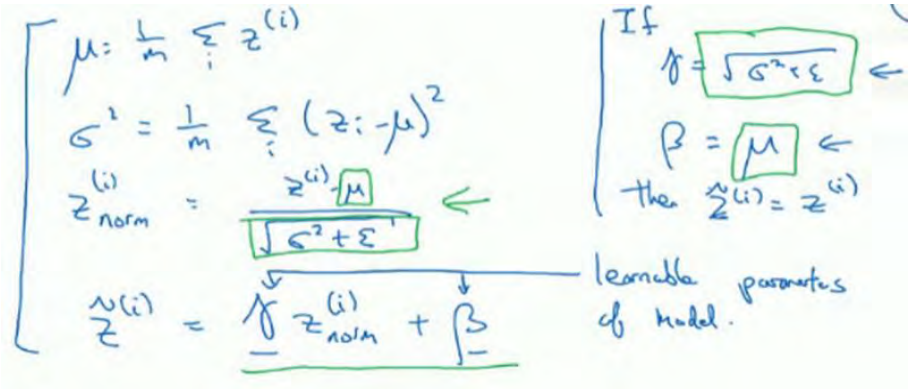
所以现在我们已把这些 z 值标准化，化为含平均值 0 和标准单位方差，所以 z 的每一个分量都含有平均值 0 和方差 1，但我们不想让隐藏单元总是含有平均值 0 和方差 1，也许隐藏单元有了不同的分布会有意义，所以我们所要做的就是计算，我们称之为$\tilde{z}^{(i)}$，$\tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta$，这里$\gamma$和$\beta$是你模型的学习参数，所以我们使用梯度下降或一些其它类似梯度下降的算法，比如 **Momentum** 或者 **Nesterov**，**Adam**，你会更新γ和β，正如更新神经网络的权重一样。



请注意γ和β的作用是，你可以随意设置$\tilde{z}^{(i)}$的平均值，事实上，如果$\gamma = \sqrt{\sigma^2 + \varepsilon}$，如果$\gamma$等于这个分母项（$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$中的分母），$\beta$等于$\mu$，这里的这个值是$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$中的

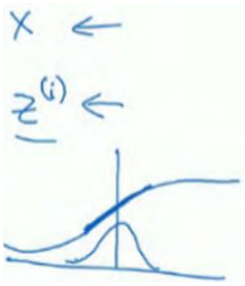
μ，那么$\gamma z_{\text{norm}}^{(i)} + \beta$的作用在于，它会精确转化这个方程，如果这些成立（$\gamma = \sqrt{\sigma^2 + \varepsilon}, \beta = \mu$），那么$\tilde{z}^{(i)} = z^{(i)}$。

通过对γ和β合理设定，规范化过程，即这四个等式，从根本来说，只是计算恒等函数，通过赋予γ和β其它值，可以使你构造含其它平均值和方差的隐藏单元值。



所以，在网络匹配这个单元的方式，之前可能是用$z^{(1)}$，$z^{(2)}$等等，现在则会用$\tilde{z}^{(i)}$取代$z^{(i)}$，方便神经网络中的后续计算。如果你想放回$[l]$，以清楚的表明它位于哪层，你可以把它放这。

所以我希望你学到的是，归一化输入特征X是怎样有助于神经网络中的学习，**Batch** 归一化的作用是它适用的归一化过程，不只是输入层，甚至同样适用于神经网络中的深度隐藏层。你应用 **Batch** 归一化了一些隐藏单元值中的平均值和方差，不过训练输入和这些隐藏单元值的一个区别是，你也许不想隐藏单元值必须是平均值 0 和方差 1。



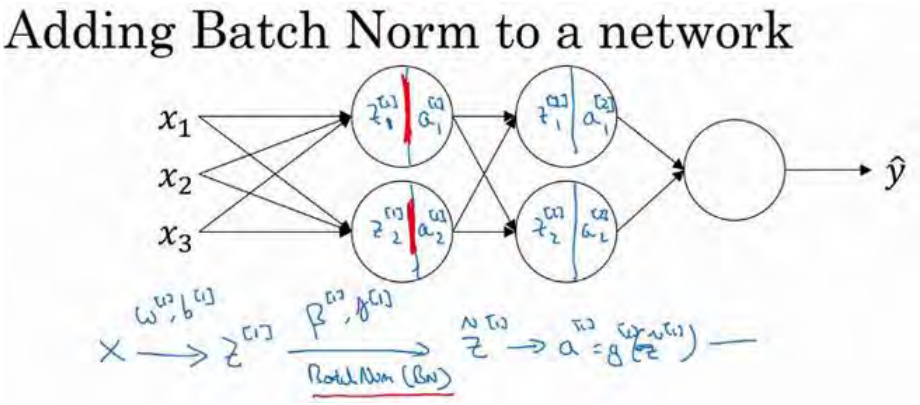
比如，如果你有 **sigmoid** 激活函数，你不想让你的值总是全部集中在这里，你想使它们有更大的方差，或不是 0 的平均值，以便更好的利用非线性的 **sigmoid** 函数，而不是使所有的值都集中于这个线性版本中，这就是为什么有了γ和β两个参数后，你可以确保所有的$z^{(i)}$值可以是你想赋予的任意值，或者它的作用是保证隐藏的单元已使均值和方差标准化。那里，均值和方差由两参数控制，即γ和β，学习算法可以设置为任何值，所以它真正的作用是，使

隐藏单元值的均值和方差标准化，即$z^{(i)}$有固定的均值和方差，均值和方差可以是 0 和 1，也可以是其它值，它是由γ和β两参数控制的。

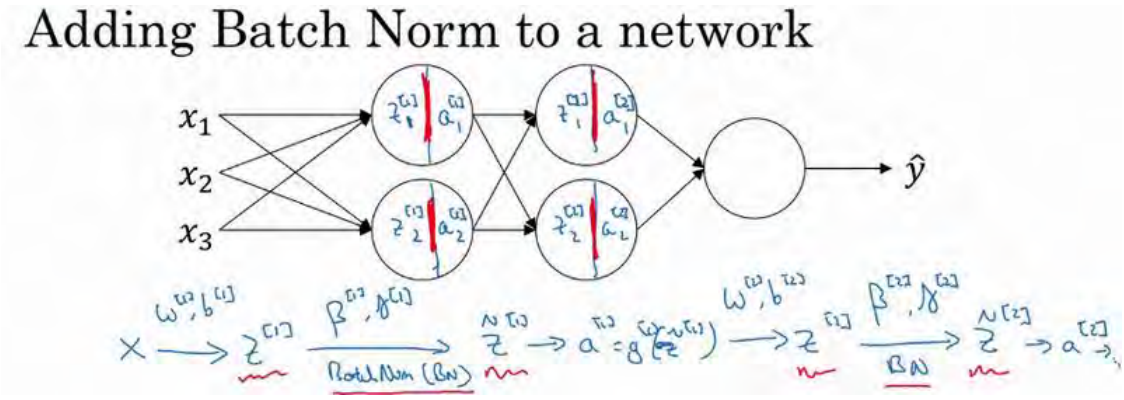
我希望你能学会怎样使用 **Batch** 归一化，至少就神经网络的单一层而言，在下一个视频中，我会教你如何将 **Batch** 归一化与神经网络甚至是深度神经网络相匹配。对于神经网络许多不同层而言，又该如何使它适用，之后，我会告诉你，**Batch** 归一化有助于训练神经网络的原因。所以如果觉得 **Batch** 归一化起作用的原因还显得有点神秘，那跟着我走，在接下来的两个视频中，我们会弄清楚。

3.5 将 Batch Norm 拟合进神经网络（Fitting Batch Norm into a neural network）

你已经看到那些等式，它可以在单一隐藏层进行 **Batch** 归一化，接下来，让我们看看它是怎样在深度网络训练中拟合的吧。

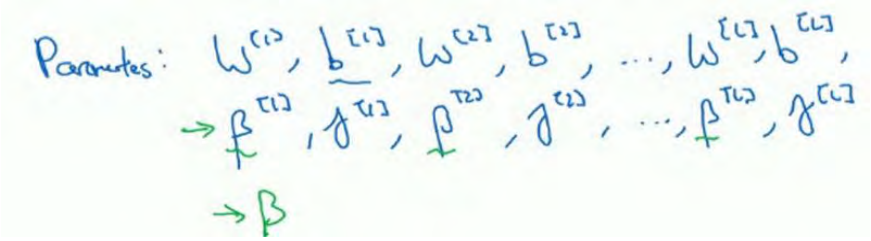


假设你有一个这样的神经网络，我之前说过，你可以认为每个单元负责计算两件事。第一，它先计算 z，然后应用其到激活函数中再计算 a，所以我可以认为，每个圆圈代表着两步的计算过程。同样的，对于下一层而言，那就是$z_1^{[2]}$和$a_1^{[2]}$等。所以如果你没有应用 **Batch** 归一化，你会把输入X拟合到第一隐藏层，然后首先计算$z^{[1]}$，这是由$w^{[1]}$和$b^{[1]}$两个参数控制的。接着，通常而言，你会把$z^{[1]}$拟合到激活函数以计算$a^{[1]}$。但 **Batch** 归一化的做法是将$z^{[1]}$值进行 **Batch** 归一化，简称 **BN**，此过程将由$\beta^{[1]}$和$\gamma^{[1]}$两参数控制，这一操作会给你一个新的规范化的$z^{[1]}$值（$\tilde{z}^{[1]}$），然后将其输入激活函数中得到$a^{[1]}$，即$a^{[1]} = g^{[1]}(\tilde{z}^{[l]})$。



现在，你已在第一层进行了计算，此时 **Batch** 归一化发生在 z 的计算和a之间，接下来，你需要应用$a^{[1]}$值来计算$z^{[2]}$，此过程是由$w^{[2]}$和$b^{[2]}$控制的。与你在第一层所做的类似，你会将$z^{[2]}$进行 **Batch** 归一化，现在我们简称 **BN**，这是由下一层的 **Batch** 归一化参数所管制的，即$\beta^{[2]}$和$\gamma^{[2]}$，现在你得到$\tilde{z}^{[2]}$，再通过激活函数计算出$a^{[2]}$等等。

所以需要强调的是 **Batch** 归一化是发生在计算z和a之间的。直觉就是，与其应用没有归一化的z值，不如用归一过的$\tilde{z}$，这是第一层（$\tilde{z}^{[1]}$）。第二层同理，与其应用没有规范过的$z^{[2]}$值，不如用经过方差和均值归一后的$\tilde{z}^{[2]}$。所以，你网络的参数就会是$w^{[1]}$，$b^{[1]}$，$w^{[2]}$和$b^{[2]}$等等，我们将要去掉这些参数。但现在，想象参数$w^{[1]}$，$b^{[1]}$到$w^{[l]}$，$b^{[l]}$，我们将另一些参数加入到此新网络中$\beta^{[1]}$，$\beta^{[2]}$，$\gamma^{[1]}$，$\gamma^{[2]}$等等。对于应用 **Batch** 归一化的每一层而言。需要澄清的是，请注意，这里的这些β（$\beta^{[1]}$，$\beta^{[2]}$等等）和超参数β没有任何关系，下一张幻灯片中会解释原因，后者是用于 **Momentum** 或计算各个指数的加权平均值。**Adam** 论文的作者，在论文里用β代表超参数。**Batch** 归一化论文的作者，则使用β代表此参数（$\beta^{[1]}$，$\beta^{[2]}$等等），但这是两个完全不同的β。我在两种情况下都决定使用β，以便你阅读那些原创的论文，但 **Batch** 归一化学习参数$\beta^{[1]}$，$\beta^{[2]}$等等和用于 **Momentum**、**Adam**、**RMSprop** 算法中的β不同。

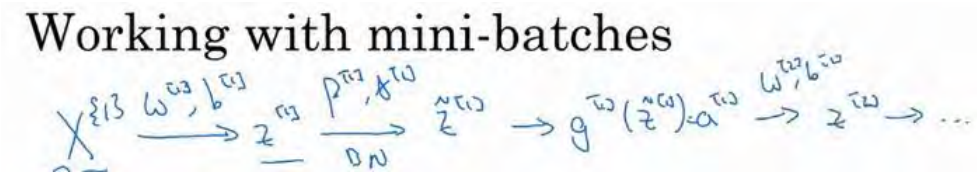


所以现在，这是你算法的新参数，接下来你可以使用想用的任何一种优化算法，比如使用梯度下降法来执行它。

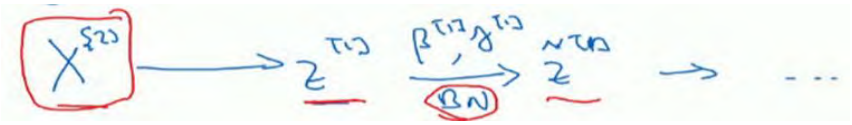
举个例子，对于给定层，你会计算$d\beta^{[l]}$，接着更新参数β为$\beta^{[l]} = \beta^{[l]} - \alpha d\beta^{[l]}$。你也可以使用 **Adam** 或 **RMSprop** 或 **Momentum**，以更新参数β和γ，并不是只应用梯度下降法。

即使在之前的视频中，我已经解释过 **Batch** 归一化是怎么操作的，计算均值和方差，减去均值，再除以方差，如果它们使用的是深度学习编程框架，通常你不必自己把 **Batch** 归一化步骤应用于 **Batch** 归一化层。因此，探究框架，可写成一行代码，比如说，在 **TensorFlow** 框架中，你可以用这个函数（`tf.nn.batch_normalization`）来实现 **Batch** 归一化，我们稍后讲解，但实践中，你不必自己操作所有这些具体的细节，但知道它是如何作用的，你可以更好的理解代码的作用。但在深度学习框架中，**Batch** 归一化的过程，经常是类似一行代码的东西。

所以，到目前为止，我们已经讲了 **Batch** 归一化，就像你在整个训练站点上训练一样，或就像你正在使用 **Batch** 梯度下降法。



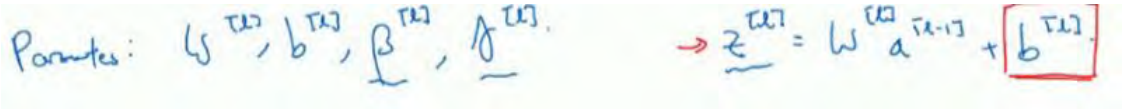
实践中，**Batch** 归一化通常和训练集的 **mini-batch** 一起使用。你应用 **Batch** 归一化的方式就是，你用第一个 **mini-batch**($X^{\{1\}}$)，然后计算$z^{[1]}$，这和上张幻灯片上我们所做的一样，应用参数$w^{[1]}$和$b^{[1]}$，使用这个 **mini-batch**($X^{\{1\}}$)。接着，继续第二个 **mini-batch**($X^{\{2\}}$)，接着 **Batch** 归一化会减去均值，除以标准差，由$\beta^{[1]}$和$\gamma^{[1]}$重新缩放，这样就得到了$\tilde{z}^{[1]}$，而所有的这些都是在第一个 **mini-batch** 的基础上，你再应用激活函数得到$a^{[1]}$。然后用$w^{[2]}$和$b^{[2]}$计算$z^{[2]}$，等等，所以你做的这一切都是为了在第一个 **mini-batch**($X^{\{1\}}$)上进行一步梯度下降法。



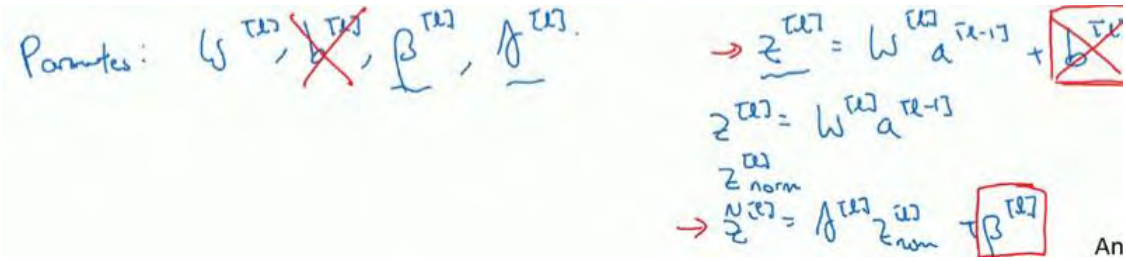
类似的工作，你会在第二个 **mini-batch**（$X^{\{2\}}$）上计算$z^{[1]}$，然后用 **Batch** 归一化来计算$\tilde{z}^{[1]}$，所以 **Batch** 归一化的此步中，你用第二个 **mini-batch**（$X^{\{2\}}$）中的数据使$\tilde{z}^{[1]}$归一化，这里的 **Batch** 归一化步骤也是如此，让我们来看看在第二个 **mini-batch**（$X^{\{2\}}$）中的例子，在 **mini-batch** 上计算$z^{[1]}$的均值和方差，重新缩放的β和γ得到$z^{[1]}$，等等。

然后在第三个 **mini-batch**（$X^{\{3\}}$）上同样这样做，继续训练。

现在，我想澄清此参数的一个细节。先前我说过每层的参数是$w^{[l]}$和$b^{[l]}$，还有$\beta^{[l]}$和$\gamma^{[l]}$，请注意计算 z 的方式如下，$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$，但 **Batch** 归一化做的是，它要看这个 **mini-batch**，先将$z^{[l]}$归一化，结果为均值 0 和标准方差，再由β和γ重缩放，但这意味着，无论$b^{[l]}$的值是多少，都是要被减去的，因为在 **Batch** 归一化的过程中，你要计算$z^{[l]}$的均值，再减去平均值，在此例中的 **mini-batch** 中增加任何常数，数值都不会改变，因为加上的任何常数都将会被均值减去所抵消。

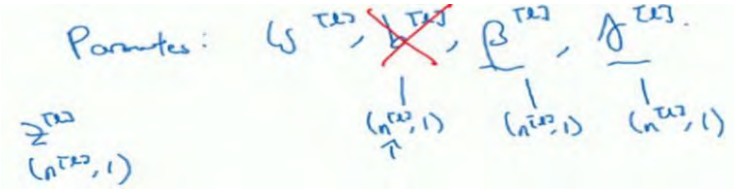


所以，如果你在使用 **Batch** 归一化，其实你可以消除这个参数（$b^{[l]}$），或者你也可以，暂时把它设置为 0，那么，参数变成$z^{[l]} = w^{[l]}a^{[l-1]}$，然后你计算归一化的$z^{[l]}$，$\tilde{z}^{[l]} = \gamma^{[l]}z^{[l]} + \beta^{[l]}$，你最后会用参数$\beta^{[l]}$，以便决定$\tilde{z}^{[l]}$的取值，这就是原因。



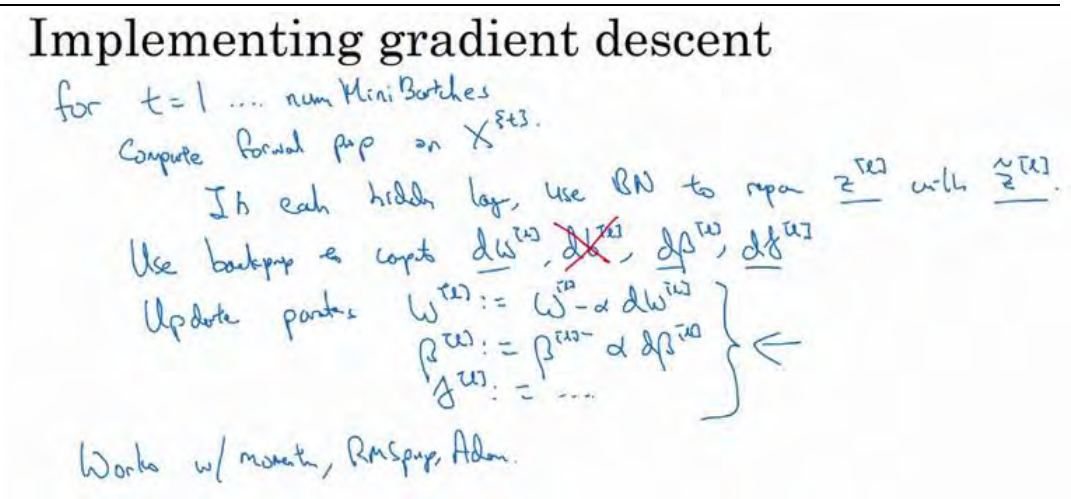
所以总结一下，因为 **Batch** 归一化超过了此层$z^{[l]}$的均值，$b^{[l]}$这个参数没有意义，所以，你必须去掉它，由$\beta^{[l]}$代替，这是个控制参数，会影响转移或偏置条件。

最后，请记住$z^{[l]}$的维数，因为在这个例子中，维数会是$(n^{[l]},1)$，$b^{[l]}$的尺寸为$(n^{[l]},1)$，如果是 l 层隐藏单元的数量，那$\beta^{[l]}$和$\gamma^{[l]}$的维度也是$(n^{[l]},1)$，因为这是你隐藏层的数量，你有$n^{[l]}$隐藏单元，所以$\beta^{[l]}$和$\gamma^{[l]}$用来将每个隐藏层的均值和方差缩放为网络想要的值。



让我们总结一下关于如何用 **Batch** 归一化来应用梯度下降法，假设你在使用 **mini-batch** 梯度下降法，你运行$t=1$到 **batch** 数量的 **for** 循环，你会在 **mini-batch** $X^{\{t\}}$上应用正向 **prop**，每个隐藏层都应用正向 **prop**，用 **Batch** 归一化代替$z^{[l]}$为$\tilde{z}^{[l]}$。接下来，它确保在这个 **mini-batch** 中，z值有归一化的均值和方差，归一化均值和方差后是$\tilde{z}^{[l]}$，然后，你用反向 **prop** 计算$dw^{[l]}$和$db^{[l]}$，及所有 l 层所有的参数，$d\beta^{[l]}$和$d\gamma^{[l]}$。尽管严格来说，因为你要去掉b，这部分其实已经去掉了。最后，你更新这些参数：$w^{[l]}=w^{[l]}-\alpha dw^{[l]}$，和以前一样，$\beta^{[l]}=\beta^{[l]}-\alpha d\beta^{[l]}$，对于$\gamma$也是如此$\gamma^{[l]}=\gamma^{[l]}-\alpha d\gamma^{[l]}$。

如果你已将梯度计算如下，你就可以使用梯度下降法了，这就是我写到这里的，但也适用于有 **Momentum**、**RMSprop**、**Adam** 的梯度下降法。与其使用梯度下降法更新 **mini-batch**，你可以使用这些其它算法来更新，我们在之前几个星期中的视频中讨论过的，也可以应用其它的一些优化算法来更新由 **Batch** 归一化添加到算法中的β 和γ 参数。



我希望，你能学会如何从头开始应用 **Batch** 归一化，如果你想的话。如果你使用深度学习编程框架之一，我们之后会谈。，希望，你可以直接调用别人的编程框架，这会使 **Batch** 归一化的使用变得很容易。

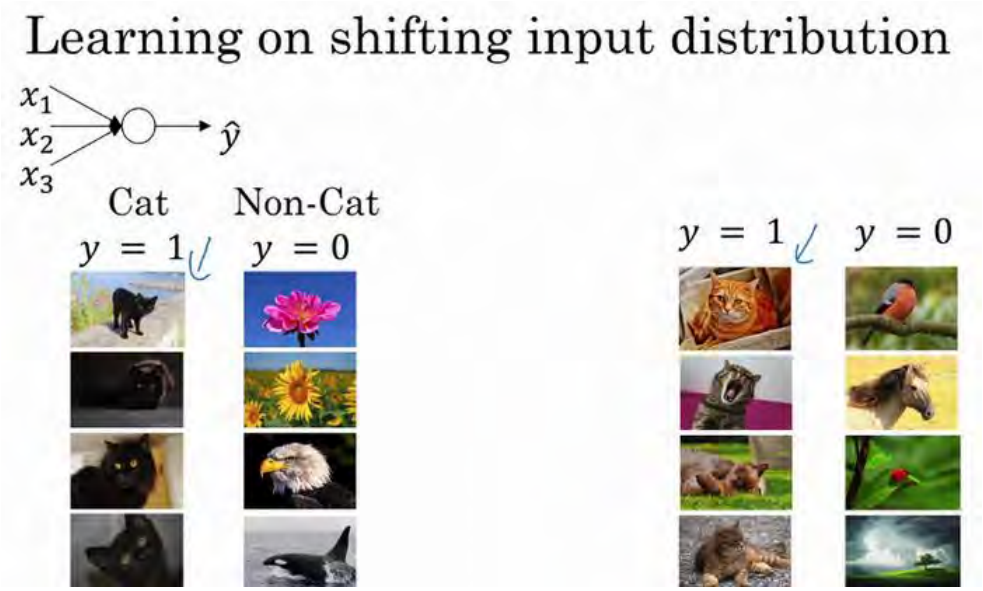
现在，以防 **Batch** 归一化仍然看起来有些神秘，尤其是你还不清楚为什么其能如此显著的加速训练，我们进入下一个视频，详细讨论 **Batch** 归一化为何效果如此显著，它到底在做什么。

3.6 Batch Norm 为什么奏效？（Why does Batch Norm work?）

为什么 **Batch** 归一化会起作用呢？

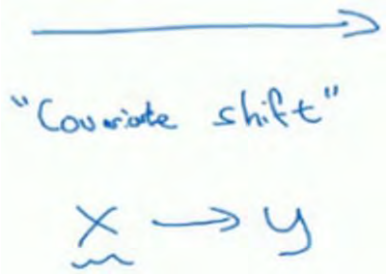
一个原因是，你已经看到如何归一化输入特征值x，使其均值为 0，方差 1，它又是怎样加速学习的，有一些从 0 到 1 而不是从 1 到 1000 的特征值，通过归一化所有的输入特征值x，以获得类似范围的值，可以加速学习。所以 **Batch** 归一化起的作用的原因，直观的一点就是，它在做类似的工作，但不仅仅对于这里的输入值，还有隐藏单元的值，这只是 **Batch** 归一化作用的冰山一角，还有些深层的原理，它会有助于你对 **Batch** 归一化的作用有更深的理解，让我们一起来看看吧。

Batch 归一化有效的第二个原因是，它可以使权重比你的网络更滞后或更深层，比如，第 10 层的权重更能经受得住变化，相比于神经网络中前层的权重，比如第 1 层，为了解释我的意思，让我们来看看这个最生动形象的例子。

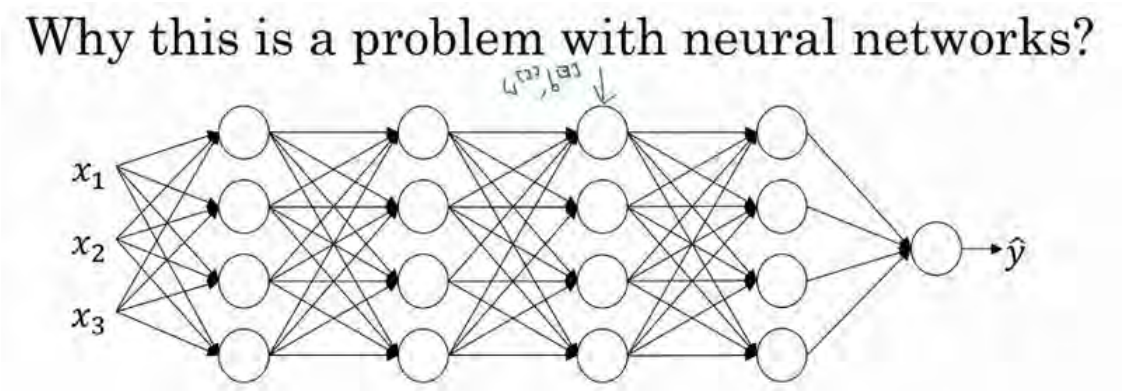


这是一个网络的训练，也许是个浅层网络，比如 **logistic** 回归或是一个神经网络，也许是个浅层网络，像这个回归函数。或一个深层网络，建立在我们著名的猫脸识别检测上，但假设你已经在所有黑猫的图像上训练了数据集，如果现在你要把此网络应用于有色猫，这种情况下，正面的例子不只是左边的黑猫，还有右边其它颜色的猫，那么你的 **cosfa** 可能适用的不会很好。

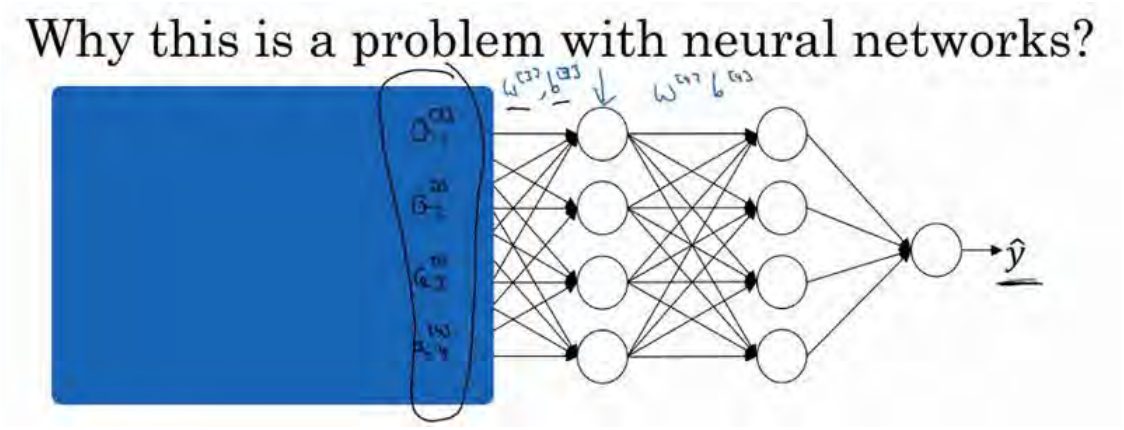
如果图像中，你的训练集是这个样子的，你的正面例子在这儿，反面例子在那儿（左图），但你试图把它们都统一于一个数据集，也许正面例子在这，反面例子在那儿（右图）。你也许无法期待，在左边训练得很好的模块，同样在右边也运行得很好，即使存在运行都很好的同一个函数，但你不会希望你的学习算法去发现绿色的决策边界，如果只看左边数据的话。



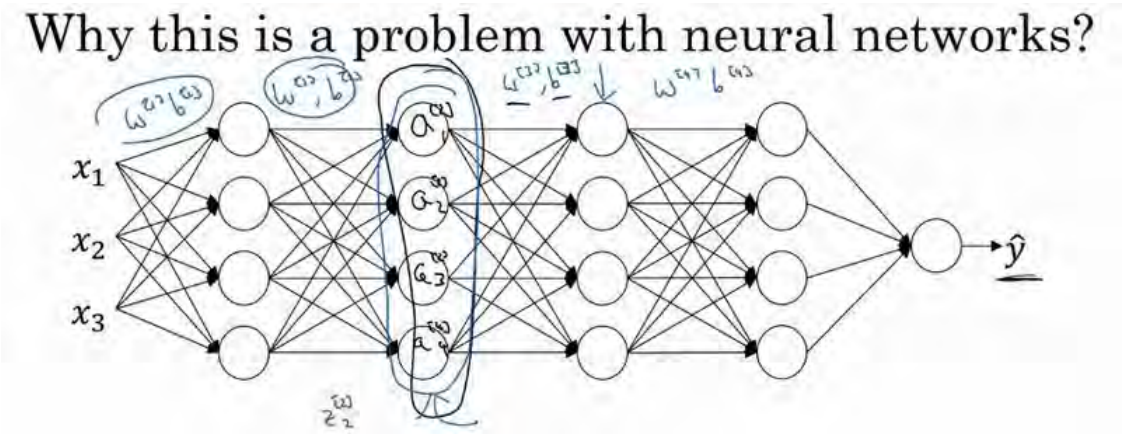
所以使你数据改变分布的这个想法，有个有点怪的名字“**Covariate shift**”，想法是这样的，如果你已经学习了x到y 的映射，如果x 的分布改变了，那么你可能需要重新训练你的学习算法。这种做法同样适用于，如果真实函数由x 到y 映射保持不变，正如此例中，因为真实函数是此图片是否是一只猫，训练你的函数的需要变得更加迫切，如果真实函数也改变，情况就更糟了。



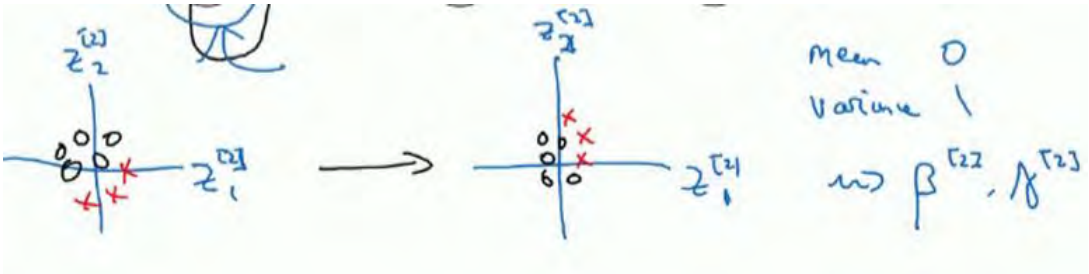
“**Covariate shift**”的问题怎么应用于神经网络呢？试想一个像这样的深度网络，让我们从这层（第三层）来看看学习过程。此网络已经学习了参数$w^{[3]}$和$b^{[3]}$，从第三隐藏层的角度来看，它从前层中取得一些值，接着它需要做些什么，使希望输出值$\hat{y}$接近真实值y。



让我先遮住左边的部分，从第三隐藏层的角度来看，它得到一些值，称为$a_1^{[2]}$，$a_2^{[2]}$，$a_3^{[2]}$，$a_4^{[2]}$，但这些值也可以是特征值x_1，x_2，x_3，x_4，第三层隐藏层的工作是找到一种方式，使这些值映射到$\hat{y}$，你可以想象做一些截断，所以这些参数$w^{[3]}$和$b^{[3]}$或$w^{[4]}$和$b^{[4]}$或$w^{[5]}$和$b^{[5]}$，也许是学习这些参数，所以网络做的不错，从左边我用黑色笔写的映射到输出值$\hat{y}$。



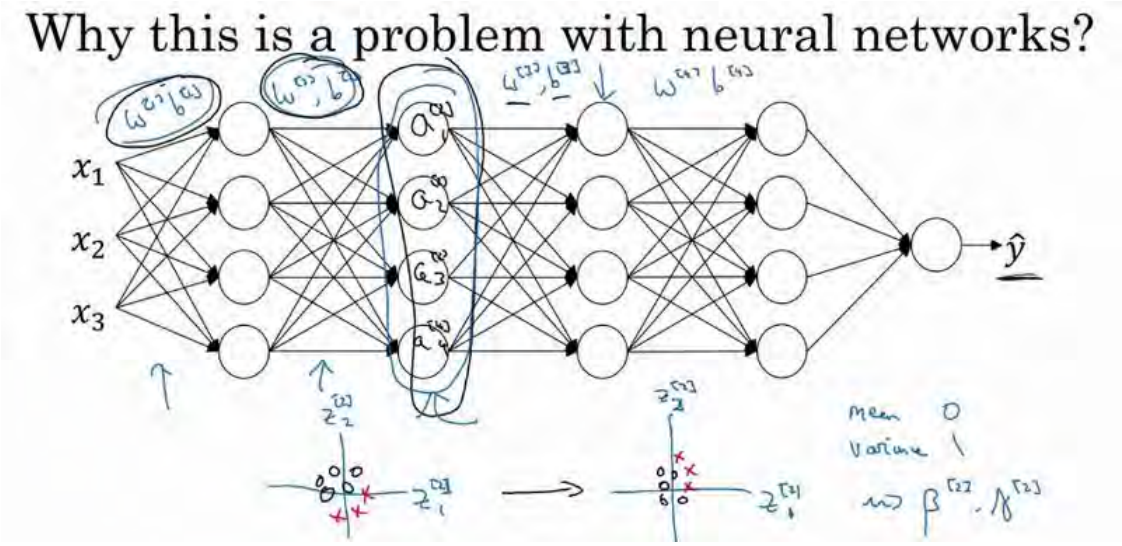
现在我们把网络的左边揭开，这个网络还有参数$w^{[2]}$，$b^{[2]}$和$w^{[1]}$，$b^{[1]}$，如果这些参数改变，这些$a^{[2]}$的值也会改变。所以从第三层隐藏层的角度来看，这些隐藏单元的值在不断地改变，所以它就有了**“Covariate shift”**的问题，上张幻灯片中我们讲过的。



Batch 归一化做的，是它减少了这些隐藏值分布变化的数量。如果是绘制这些隐藏的单元值的分布，也许这是重整值 z ，这其实是$z_1^{[2]}$，$z_2^{[2]}$，我要绘制两个值而不是四个值，以便我们设想为 **2D**，**Batch** 归一化讲的是$z_1^{[2]}$，$z_2^{[2]}$的值可以改变，它们的确会改变，当神经网络

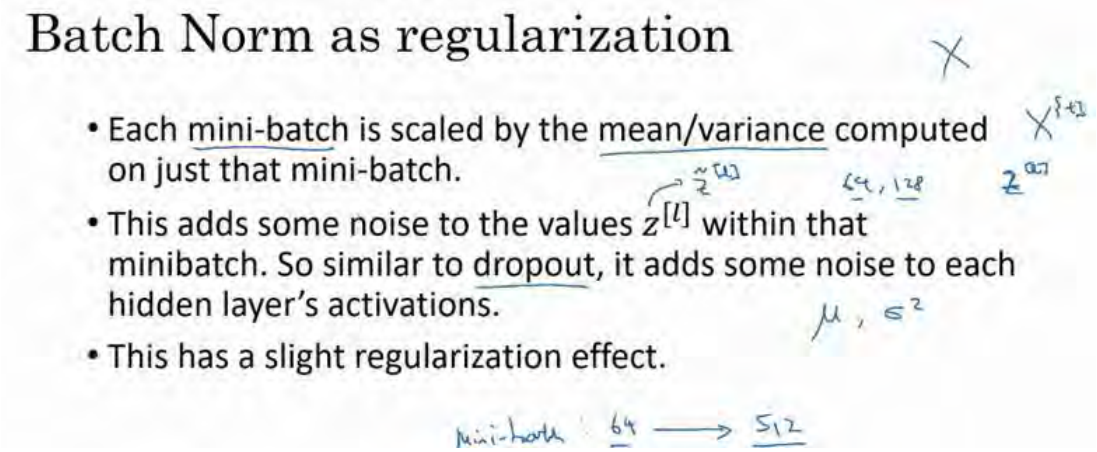
在之前层中更新参数，**Batch** 归一化可以确保无论其怎样变化$z_1^{[2]}$，$z_2^{[2]}$的均值和方差保持不变，所以即使$z_1^{[2]}$，$z_2^{[2]}$的值改变，至少他们的均值和方差也会是均值 0，方差 1，或不一定必须是均值 0，方差 1，而是由$\beta^{[2]}$和$\gamma^{[2]}$决定的值。如果神经网络选择的话，可强制其为均值 0，方差 1，或其他任何均值和方差。但它做的是，它限制了在前层的参数更新，会影响数值分布的程度，第三层看到的这种情况，因此得到学习。

Batch 归一化减少了输入值改变的问题，它的确使这些值变得更稳定，神经网络的之后层就会有更坚实的基础。即使使输入分布改变了一些，它会改变得更少。它做的是当前层保持学习，当改变时，迫使后层适应的程度减小了，你可以这样想，它减弱了前层参数的作用与后层参数的作用之间的联系，它使得网络每层都可以自己学习，稍稍独立于其它层，这有助于加速整个网络的学习。



所以，希望这能带给你更好的直觉，重点是 **Batch** 归一化的意思是，尤其从神经网络后层之一的角度而言，前层不会左右移动的吧么多，因为它们被同样的均值和方差所限制，所以，这会使得后层的学习工作变得更容易些。

Batch 归一化还有一个作用，它有轻微的正则化效果，**Batch** 归一化中非直观的一件事是，每个 **mini-batch**，我会说 **mini-batch**$X^{\{t\}}$的值为$z^{[l]}$，$z^{[l]}$，在 **mini-batch** 计算中，由均值和方差缩放的，因为在 **mini-batch** 上计算的均值和方差，而不是在整个数据集上，均值和方差有一些小的噪声，因为它只在你的 **mini-batch** 上计算，比如 64 或 128 或 256 或更大的训练例子。因为均值和方差有一点小噪音，因为它只是由一小部分数据估计得出的。缩放过程从$z^{[l]}$到$\tilde{z}^{[l]}$，过程也有一些噪音，因为它是用有些噪音的均值和方差计算得出的。



所以和 **dropout** 相似，它往每个隐藏层的激活值上增加了噪音，**dropout** 有增加噪音的方式，它使一个隐藏的单元，以一定的概率乘以 0，以一定的概率乘以 1，所以你的 **dropout** 含几重噪音，因为它乘以 0 或 1。

对比而言，**Batch** 归一化含几重噪音，因为标准偏差的缩放和减去均值带来的额外噪音。这里的均值和标准差的估计值也是有噪音的，所以类似于 **dropout**，**Batch** 归一化有轻微的正则化效果，因为给隐藏单元添加了噪音，这迫使后部单元不过分依赖任何一个隐藏单元，类似于 **dropout**，它给隐藏层增加了噪音，因此有轻微的正则化效果。因为添加的噪音很微小，所以并不是巨大的正则化效果，你可以将 **Batch** 归一化和 **dropout** 一起使用，如果你想得到 **dropout** 更强大的正则化效果。

也许另一个轻微非直观的效果是，如果你应用了较大的 **mini-batch**，对，比如说，你用了 512 而不是 64，通过应用较大的 **min-batch**，你减少了噪音，因此减少了正则化效果，这是 **dropout** 的一个奇怪的性质，就是应用较大的 **mini-batch** 可以减少正则化效果。

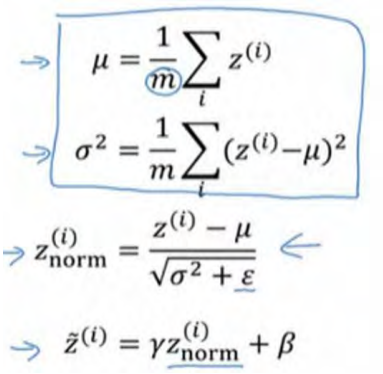
说到这儿，我会把 **Batch** 归一化当成一种正则化，这确实不是其目的，但有时它会对你的算法有额外的期望效应或非期望效应。但是不要把 **Batch** 归一化当作正则化，把它当作将你归一化隐藏单元激活值并加速学习的方式，我认为正则化几乎是一个意想不到的副作用。

所以希望这能让你更理解 **Batch** 归一化的工作，在我们结束 **Batch** 归一化的讨论之前，我想确保你还知道一个细节。**Batch** 归一化一次只能处理一个 **mini-batch** 数据，它在 **mini-batch** 上计算均值和方差。所以测试时，你试图做出预测，试着评估神经网络，你也许没有 **mini-batch** 的例子，你也许一次只能进行一个简单的例子，所以测试时，你需要做一些不同的东西以确保你的预测有意义。

在下一个也就是最后一个 **Batch** 归一化视频中，让我们详细谈谈你需要注意的一些细节，来让你的神经网络应用 **Batch** 归一化来做出预测。

3.7 测试时的 Batch Norm（Batch Norm at test time）

Batch 归一化将你的数据以 **mini-batch** 的形式逐一处理，但在测试时，你可能需要对每个样本逐一处理，我们来看一下怎样调整你的网络来做到这一点。



$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

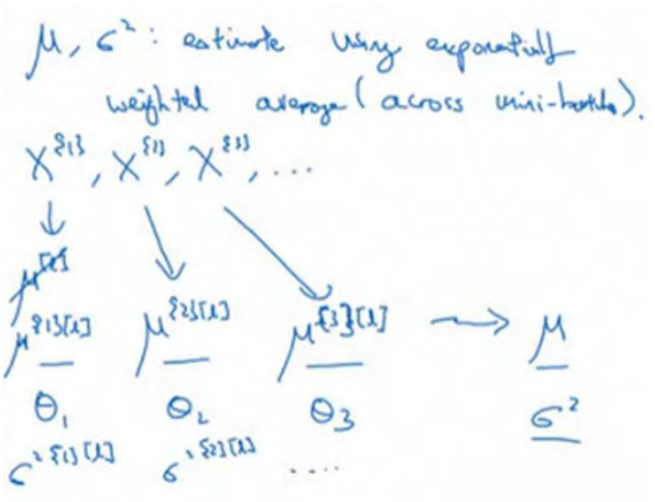
$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)}-\mu)^2$$

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)}-\mu}{\sqrt{\sigma^2+\varepsilon}}$$

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{\text{norm}} + \beta$$

回想一下，在训练时，这些就是用来执行 **Batch** 归一化的等式。在一个 **mini-batch** 中，你将 **mini-batch** 的$z^{(i)}$值求和，计算均值，所以这里你只把一个 **mini-batch** 中的样本都加起来，我用 m 来表示这个 **mini-batch** 中的样本数量，而不是整个训练集。然后计算方差，再算$z^{(i)}_{\text{norm}}$，即用均值和标准差来调整，加上ε是为了数值稳定性。$\tilde{z}$是用γ和β再次调整z_{norm}得到的。

请注意用于调节计算的μ和σ^2是在整个 **mini-batch** 上进行计算，但是在测试时，你可能不能将一个 **mini-batch** 中的 6428 或 2056 个样本同时处理，因此你需要用其它方式来得到μ和σ^2，而且如果你只有一个样本，一个样本的均值和方差没有意义。那么实际上，为了将你的神经网络运用于测试，就需要单独估算μ和σ^2，在典型的 **Batch** 归一化运用中，你需要用一个指数加权平均来估算，这个平均数涵盖了所有 **mini-batch**，接下来我会具体解释。



我们选择l层，假设我们有 **mini-batch**，$X^{[1]}$，$X^{[2]}$，$X^{[3]}$……以及对应的y值等等，那么在

为l层训练$X^{\{1\}}$时，你就得到了$\mu^{[l]}$，我还是把它写做第一个 **mini-batch** 和这一层的μ吧，（$\mu^{[l]} \rightarrow \mu^{\{1\}[l]}$）。当你训练第二个 **mini-batch**，在这一层和这个 **mini-batch** 中，你就会得到第二个μ（$\mu^{\{2\}[l]}$）值。然后在这一隐藏层的第三个 **mini-batch**，你得到了第三个μ（$\mu^{\{3\}[l]}$）值。正如我们之前用的指数加权平均来计算θ_1，θ_2，θ_3的均值，当时是试着计算当前气温的指数加权平均，你会这样来追踪你看到的这个均值向量的最新平均值，于是这个指数加权平均就成了你对这一隐藏层的z均值的估值。同样的，你可以用指数加权平均来追踪你在这一层的第一个 **mini-batch** 中所见的σ^2的值，以及第二个 **mini-batch** 中所见的σ^2的值等等。因此在用不同的 **mini-batch** 训练神经网络的同时，能够得到你所查看的每一层的μ和σ^2的平均数的实时数值。

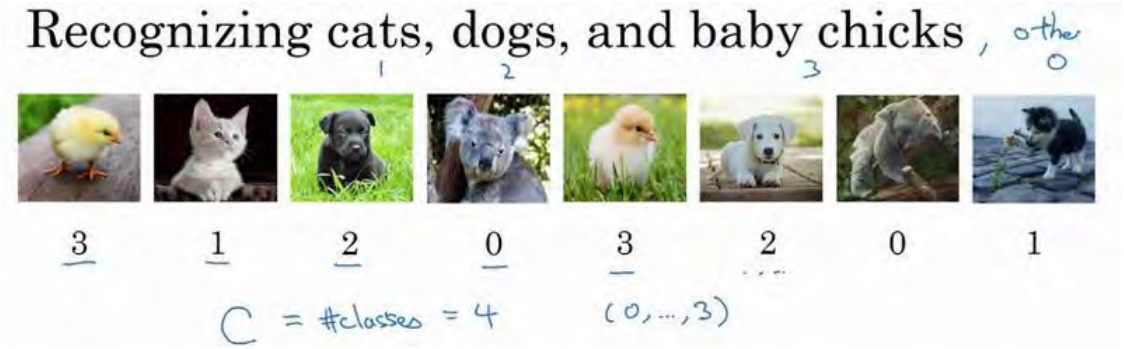
最后在测试时，对应这个等式（$z_{\text{norm}}^{(i)} = \frac{z^{(i)}-\mu}{\sqrt{\sigma^2+\varepsilon}}$），你只需要用你的 z 值来计算$z_{\text{norm}}^{(i)}$，用μ和σ^2的指数加权平均，用你手头的最新数值来做调整，然后你可以用左边我们刚算出来的z_{norm}和你在神经网络训练过程中得到的β和γ参数来计算你那个测试样本的$\tilde{z}$值。

总结一下就是，在训练时，μ和σ^2是在整个 **mini-batch** 上计算出来的包含了像是 64 或 28 或其它一定数量的样本，但在测试时，你可能需要逐一处理样本，方法是根据你的训练集估算μ和σ^2，估算的方式有很多种，理论上你可以在最终的网络中运行整个训练集来得到μ和σ^2，但在实际操作中，我们通常运用指数加权平均来追踪在训练过程中你看到的μ和σ^2的值。还可以用指数加权平均，有时也叫做流动平均来粗略估算μ和σ^2，然后在测试中使用μ和σ^2的值来进行你所需要的隐藏单元z值的调整。在实践中，不管你用什么方式估算μ和σ^2，这套过程都是比较稳健的，因此我不太会担心你具体的操作方式，而且如果你使用的是某种深度学习框架，通常会有默认的估算μ和σ^2的方式，应该一样会起到比较好的效果。但在实践中，任何合理的估算你的隐藏单元z值的均值和方差的方式，在测试中应该都会有效。

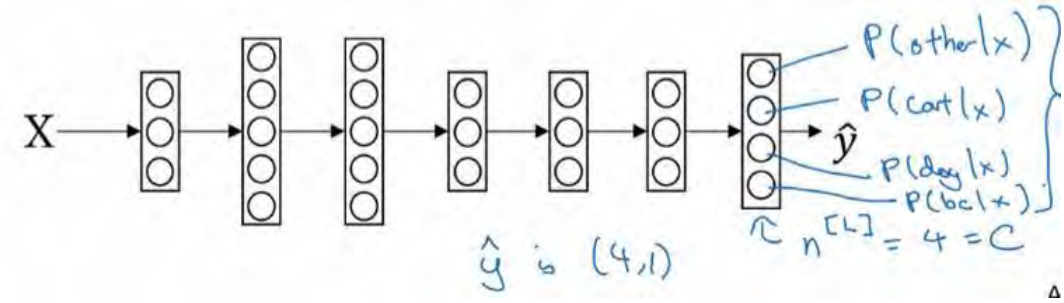
Batch 归一化就讲到这里，使用 **Batch** 归一化，你能够训练更深的网络，让你的学习算法运行速度更快，在结束这周的课程之前，我还想和你们分享一些关于深度学习框架的想法，让我们在下一段视频中一起讨论这个话题。

3.8 Softmax 回归（Softmax regression）

到目前为止，我们讲到过的分类的例子都使用了二分分类，这种分类只有两种可能的标记 0 或 1，这是一只猫或者不是一只猫，如果我们有多种可能的类型的话呢？有一种 **logistic** 回归的一般形式，叫做 **Softmax** 回归，能让你在试图识别某一分类时做出预测，或者说是多种分类中的一个，不只是识别两个分类，我们来一起看一下。



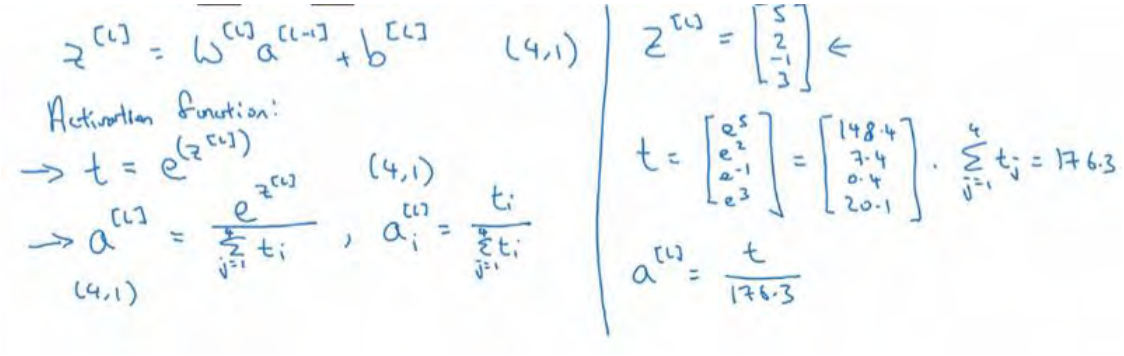
假设你不单需要识别猫，而是想识别猫，狗和小鸡，我把猫加做类 1，狗为类 2，小鸡是类 3，如果不属于以上任何一类，就分到“其它”或者说“以上均不符合”这一类，我把它叫做类 0。这里显示的图片及其对应的分类就是一个例子，这幅图片上是一只小鸡，所以是类 3，猫是类 1，狗是类 2，我猜这是一只考拉，所以以上均不符合，那就是类 0，下一个类 3，以此类推。我们将会用符号表示，我会用大写的C来表示你的输入会被分入的类别总个数，在这个例子中，我们有 4 种可能的类别，包括“其它”或“以上均不符合”这一类。当有 4 个分类时，指示类别的数字，就是从 0 到$C-1$，换句话说就是 0、1、2、3。



在这个例子中，我们建立一个神经网络，其输出层有 4 个，在这个例子中，我们将建立一个神经网络，其输出层有 4 个，或者说C个输出单元，因此n，即输出层也就是L层的单元数量，等于 4，或者一般而言等于C。我们想要输出层单元的数字告诉我们这 4 种类型中每个的概率有多大，所以这里的第一个节点(最后输出的第 1 个方格+圆圈)输出的应该是或者说我们希望它输出“其它”类的概率。在输入X的情况下，这个(最后输出的第 2 个方格+圆圈)

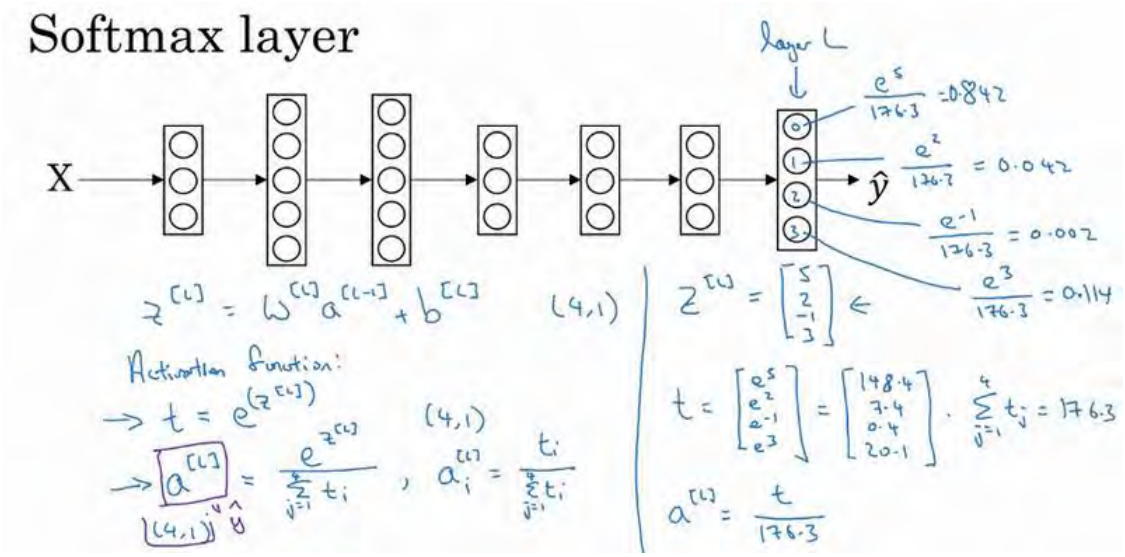
会输出猫的概率。在输入X的情况下，这个会输出狗的概率(最后输出的第 3 个方格+圆圈)。在输入X的情况下，输出小鸡的概率（最后输出的第 3 个方格+圆圈），我把小鸡缩写为 **bc**（**baby chick**）。因此这里的$\hat{y}$将是一个4×1维向量，因为它必须输出四个数字，给你这四种概率，因为它们加起来应该等于 1，输出中的四个数字加起来应该等于 1。

让你的网络做到这一点的标准模型要用到 **Softmax** 层，以及输出层来生成输出，让我把式子写下来，然后回过头来，就会对 **Softmax** 的作用有一点感觉了。

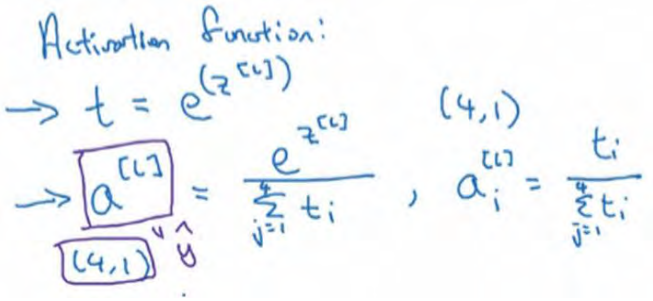


在神经网络的最后一层，你将会像往常一样计算各层的线性部分，$z^{[l]}$这是最后一层的z变量，记住这是大写L层，和往常一样，计算方法是$z^{[l]} = W^{[l]}a^{[L-1]} + b^{[l]}$，算出了$z$之后，你需要应用 **Softmax** 激活函数，这个激活函数对于 **Softmax** 层而言有些不同，它的作用是这样的。首先，我们要计算一个临时变量，我们把它叫做 t，它等于$e^{z^{[l]}}$，这适用于每个元素，而这里的$z^{[l]}$，在我们的例子中，$z^{[l]}$是 4×1 的，四维向量$t = e^{z^{[l]}}$，这是对所有元素求幂，t也是一个 4×1 维向量，然后输出的$a^{[l]}$，基本上就是向量t，但是会归一化，使和为 1。因此$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{j=1}^{4} t_i}$，换句话说，$a^{[l]}$也是一个 4×1 维向量，而这个四维向量的第$i$个元素，我把它写下来，$a_i^{[l]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$，以防这里的计算不够清晰易懂，我们马上会举个例子来详细解释。

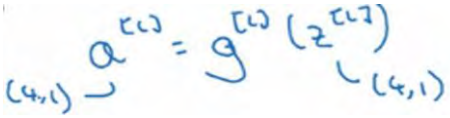
我们来看一个例子，详细解释，假设你算出了$z^{[l]}$，$z^{[l]}$是一个四维向量，假设为$z^{[l]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$，我们要做的就是用这个元素取幂方法来计算$t$，所以$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$，如果你按一下计算器就会得到以下值$t = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$，我们从向量 t 得到向量$a^{[l]}$就只需要将这些项目归一化，使总和为 1。如果你把$t$的元素都加起来，把这四个数字加起来，得到 176.3，最终$a^{[l]} = \frac{t}{176.3}$。



例如这里的第一个节点，它会输出$\frac{e^5}{176.3} = 0.842$，这样说来，对于这张图片，如果这是你得到的$z$值($\begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$)，它是类 0 的概率就是 84.2%。下一个节点输出$\frac{e^2}{176.3} = 0.042$，也就是 4.2%的几率。下一个是$\frac{e^{-1}}{176.3} = 0.002$。最后一个是$\frac{e^3}{176.3} = 0.114$，也就是 11.4%的概率属于类 3，也就是小鸡组，对吧？这就是它属于类 0，类 1，类 2，类 3 的可能性。



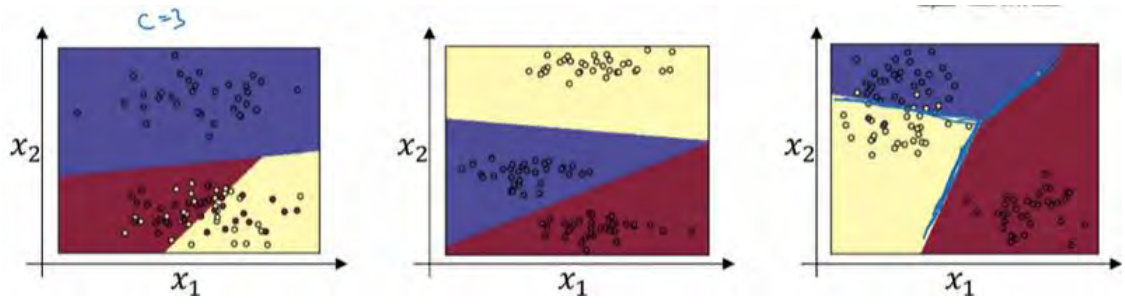
神经网络的输出$a^{[l]}$，也就是$\hat{y}$，是一个 4×1 维向量，这个 4×1 向量的元素就是我们算出来的这四个数字($\begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$)，所以这种算法通过向量$z^{[l]}$计算出总和为 1 的四个概率。



如果我们总结一下从$z^{[l]}$到$a^{[l]}$的计算步骤，整个计算过程，从计算幂到得出临时变量t，再归一化，我们可以将此概括为一个 **Softmax** 激活函数。设$a^{[l]} = g^{[l]}(z^{[l]})$，这一激活函数的与众不同之处在于，这个激活函数$g$ 需要输入一个 4×1 维向量，然后输出一个 4×1 维向量。之前，我们的激活函数都是接受单行数值输入，例如 **Sigmoid** 和 **ReLu** 激活函数，输入

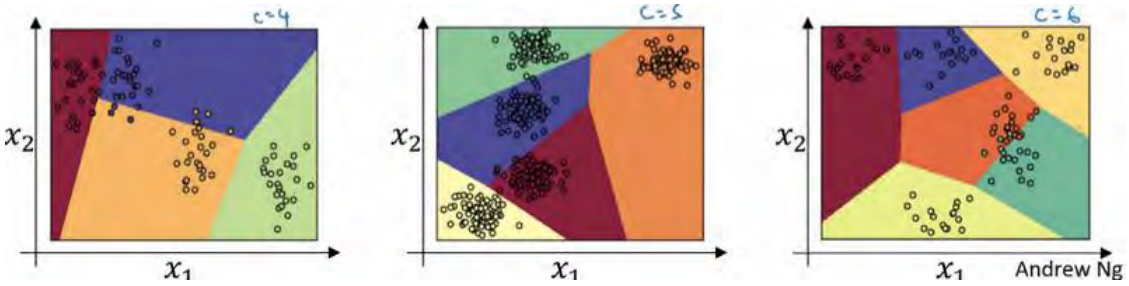
一个实数，输出一个实数。**Softmax** 激活函数的特殊之处在于，因为需要将所有可能的输出归一化，就需要输入一个向量，最后输出一个向量。

那么 **Softmax** 分类器还可以代表其它的什么东西么？我来举几个例子，你有两个输入x_1，x_2，它们直接输入到 **Softmax** 层，它有三四个或者更多的输出节点，输出$\hat{y}$，我将向你展示一个没有隐藏层的神经网络，它所做的就是计算$z^{[1]} = W^{[1]}x + b^{[1]}$，而输出的出$a^{[l]}$，或者说$\hat{y}$，$a^{[l]} = y = g(z^{[1]})$，就是$z^{[1]}$的 **Softmax** 激活函数，这个没有隐藏层的神经网络应该能让你对 **Softmax** 函数能够代表的东西有所了解。



这个例子中（左边图），原始输入只有x_1和x_2，一个$C = 3$个输出分类的 **Softmax** 层能够代表这种类型的决策边界，请注意这是几条线性决策边界，但这使得它能够将数据分到 3 个类别中，在这张图表中，我们所做的是选择这张图中显示的训练集，用数据的 3 种输出标签来训练 **Softmax** 分类器，图中的颜色显示了 **Softmax** 分类器的输出的阈值，输入的着色是基于三种输出中概率最高的那种。因此我们可以看到这是 **logistic** 回归的一般形式，有类似线性的决策边界，但有超过两个分类，分类不只有 0 和 1，而是可以是 0，1 或 2。

这是（中间图）另一个 **Softmax** 分类器可以代表的决策边界的例子，用有三个分类的数据集来训练，这里（右边图）还有一个。对吧，但是直觉告诉我们，任何两个分类之间的决策边界都是线性的，这就是为什么你看到，比如这里黄色和红色分类之间的决策边界是线性边界，紫色和红色之间的也是线性边界，紫色和黄色之间的也是线性决策边界，但它能用这些不同的线性函数来把空间分成三类。



我们来看一下更多分类的例子，这个例子中（左边图）$C = 4$，因此这个绿色分类和 **Softmax** 仍旧可以代表多种分类之间的这些类型的线性决策边界。另一个例子（中间图）是

$C = 5$类，最后一个例子（右边图）是$C = 6$，这显示了 **Softmax** 分类器在没有隐藏层的情况下能够做到的事情，当然更深的神经网络会有x，然后是一些隐藏单元，以及更多隐藏单元等等，你就可以学习更复杂的非线性决策边界，来区分多种不同分类。

我希望你了解了神经网络中的 **Softmax** 层或者 **Softmax** 激活函数有什么作用，下一个视频中，我们来看一下你该怎样训练一个使用 **Softmax** 层的神经网络。

3.9 训练一个 Softmax 分类器（Training a Softmax classifier）

上一个视频中我们学习了 **Softmax** 层和 **Softmax** 激活函数，在这个视频中，你将更深入地了解 **Softmax** 分类，并学习如何训练一个使用了 **Softmax** 层的模型。

(4,1)

$$z^{[L]}=\begin{bmatrix}5\\2\\-1\\3\end{bmatrix}\qquad t=\begin{bmatrix}e^{5}\\e^{2}\\e^{-1}\\e^{3}\end{bmatrix}$$

回忆一下我们之前举的的例子，输出层计算出的$z^{[l]}$如下，$z^{[l]}=\begin{bmatrix}5\\2\\-1\\3\end{bmatrix}$我们有四个分类$C=4$，$z^{[l]}$可以是 4×1 维向量，我们计算了临时变量$t$，$t=\begin{bmatrix}e^{5}\\e^{2}\\e^{-1}\\e^{3}\end{bmatrix}$，对元素进行幂运算，最后，如果你的输出层的激活函数$g^{[L]}()$是 **Softmax** 激活函数，那么输出就会是这样的：

$$a^{[L]}=g^{[L]}(z^{[L]})=\begin{bmatrix}e^{5}/(e^{5}+e^{2}+e^{-1}+e^{3})\\e^{2}/(e^{5}+e^{2}+e^{-1}+e^{3})\\e^{-1}/(e^{5}+e^{2}+e^{-1}+e^{3})\\e^{3}/(e^{5}+e^{2}+e^{-1}+e^{3})\end{bmatrix}=\begin{bmatrix}0.842\\0.042\\0.002\\0.114\end{bmatrix}$$

简单来说就是用临时变量t将它归一化，使总和为 1，于是这就变成了$a^{[L]}$，你注意到向量z中，最大的元素是 5，而最大的概率也就是第一种概率。

Understanding softmax

(4,1)

$$z^{[L]}=\begin{bmatrix}5\\2\\-1\\3\end{bmatrix}\qquad t=\begin{bmatrix}e^{5}\\e^{2}\\e^{-1}\\e^{3}\end{bmatrix}$$

C=4 $g^{[L]}(\cdot)$

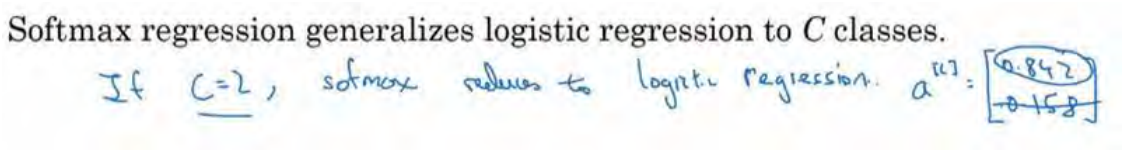
$$a^{[L]}=g^{[L]}(z^{[L]})=\begin{bmatrix}e^{5}/(e^{5}+e^{2}+e^{-1}+e^{3})\\e^{2}/(e^{5}+e^{2}+e^{-1}+e^{3})\\e^{-1}/(e^{5}+e^{2}+e^{-1}+e^{3})\\e^{3}/(e^{5}+e^{2}+e^{-1}+e^{3})\end{bmatrix}=\begin{bmatrix}0.842\\0.042\\0.002\\0.114\end{bmatrix}$$

"hard max"

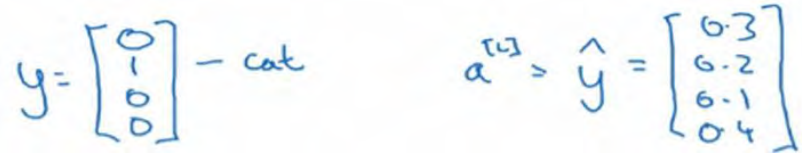
$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$

Softmax 这个名称的来源是与所谓 **hardmax** 对比，**hardmax** 会把向量z变成这个向量$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$，

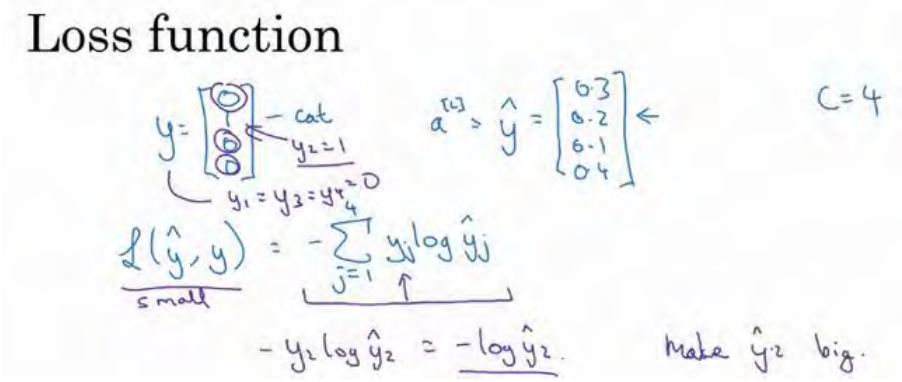
hardmax 函数会观察z的元素，然后在z中最大元素的位置放上 1，其它位置放上 0，所这是一个 **hard max**，也就是最大的元素的输出为 1，其它的输出都为 0。与之相反，**Softmax** 所做的从z到这些概率的映射更为温和，我不知道这是不是一个好名字，但至少这就是 **softmax** 这一名称背后所包含的想法，与 **hardmax** 正好相反。



有一点我没有细讲，但之前已经提到过的，就是 **Softmax** 回归或 **Softmax** 激活函数将 **logistic** 激活函数推广到C类，而不仅仅是两类，结果就是如果$C = 2$，那么$C = 2$的 **Softmax** 实际上变回了 **logistic** 回归，我不会在这个视频中给出证明，但是大致的证明思路是这样的，如果$C = 2$，并且你应用了 **Softmax**，那么输出层$a^{[L]}$将会输出两个数字，如果$C = 2$的话，也许输出 0.842 和 0.158，对吧？这两个数字加起来要等于 1，因为它们的和必须为 1，其实它们是冗余的，也许你不需要计算两个，而只需要计算其中一个，结果就是你最终计算那个数字的方式又回到了 **logistic** 回归计算单个输出的方式。这算不上是一个证明，但我们可以从中得出结论，**Softmax** 回归将 **logistic** 回归推广到了两种分类以上。



接下来我们来看怎样训练带有 **Softmax** 输出层的神经网络，具体而言，我们先定义训练神经网络使会用到的损失函数。举个例子，我们来看看训练集中某个样本的目标输出，真实标签是$\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$，用上一个视频中讲到过的例子，这表示这是一张猫的图片，因为它属于类 1，现在我们假设你的神经网络输出的是$\hat{y}$，$\hat{y}$是一个包括总和为 1 的概率的向量，$y = \begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$，你可以看到总和为 1，这就是$a^{[l]}$，$a^{[l]} = y = \begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$。对于这个样本神经网络的表现不佳，这实际上是一只猫，但却只分配到 20%是猫的概率，所以在本例中表现不佳。



那么你想用什么损失函数来训练这个神经网络？在 **Softmax** 分类中，我们一般用到的损失函数是$L(\hat{y},y)=-\sum_{j=1}^{4}y_j log\,\hat{y}_j$，我们来看上面的单个样本来更好地理解整个过程。注意在这个样本中$y_1=y_3=y_4=0$，因为这些都是 0，只有$y_2=1$，如果你看这个求和，所有含有值为 0 的y_j的项都等于 0，最后只剩下$-y_2 log\,\hat{y}_2$，因为当你按照下标 j 全部加起来，所有的项都为 0，除了$j=2$时，又因为$y_2=1$，所以它就等于$-log\,\hat{y}_2$。 $L(\hat{y},y)=-\sum_{j=1}^{4}y_j\log\hat{y}_j=-y_2\,log\,\hat{y}_2=-log\,\hat{y}_2$

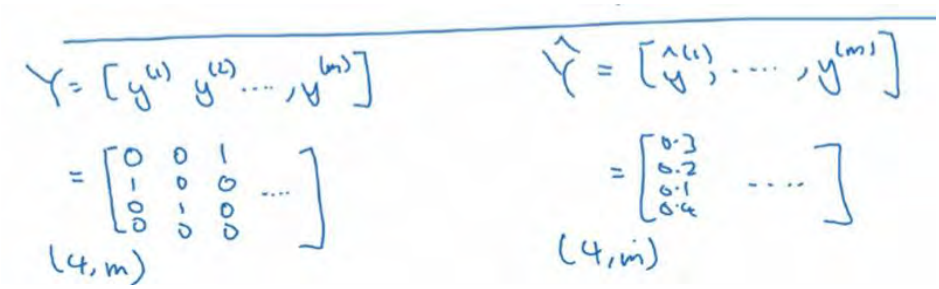
这就意味着，如果你的学习算法试图将它变小，因为梯度下降法是用来减少训练集的损失的，要使它变小的唯一方式就是使$-\log\hat{y}_2$变小，要想做到这一点，就需要使$\hat{y}_2$尽可能大，因为这些是概率，所以不可能比 1 大，但这的确也讲得通，因为在这个例子中x是猫的图片，你就需要这项输出的概率尽可能地大（$y=\begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$中第二个元素）。

概括来讲，损失函数所做的就是它找到你的训练集中的真实类别，然后试图使该类别相应的概率尽可能地高，如果你熟悉统计学中最大似然估计，这其实就是最大似然估计的一种形式。但如果你不知道那是什么意思，也不用担心，用我们刚刚讲过的算法思维也足够了。

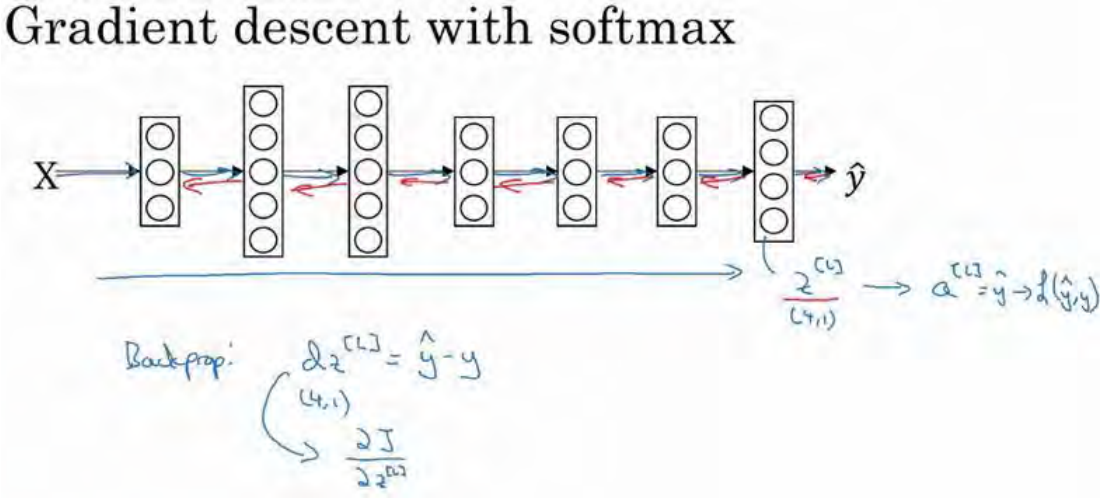
这是单个训练样本的损失，整个训练集的损失J又如何呢？也就是设定参数的代价之类的，还有各种形式的偏差的代价，它的定义你大致也能猜到，就是整个训练集损失的总和，把你的训练算法对所有训练样本的预测都加起来，

$$J(w^{[1]},b^{[1]},\dots\dots)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})$$

因此你要做的就是用梯度下降法，使这里的损失最小化。

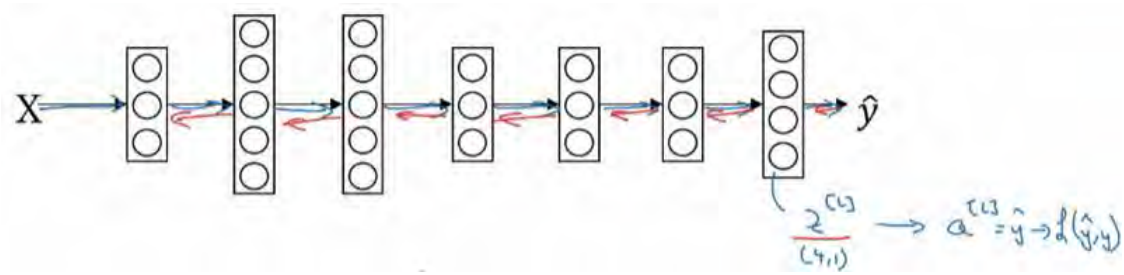


最后还有一个实现细节，注意因为$C=4$，y是一个 4×1 向量，y也是一个 4×1 向量，如果你实现向量化，矩阵大写Y就是$[y^{(1)}y^{(2)}\ldots\ldots y^{(m)}]$，例如如果上面这个样本是你的第一个训练样本，那么矩阵$Y=\begin{bmatrix}0&0&1&\ldots\\1&0&0&\ldots\\0&1&0&\ldots\\0&0&0&\ldots\end{bmatrix}$，那么这个矩阵$Y$最终就是一个$4\times m$维矩阵。类似的，$\hat{Y}=[\hat{y}^{(1)}\,\hat{y}^{(2)}\ldots\ldots\hat{y}^{(m)}]$，这个其实就是$\hat{y}^{(1)}$（$a^{[l](1)}=y^{(1)}=\begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$），或是第一个训练样本的输出，那么$\hat{Y}=\begin{bmatrix}0.3&\ldots\\0.2&\ldots\\0.1&\ldots\\0.4&\ldots\end{bmatrix}$，$\hat{Y}$本身也是一个$4\times m$维矩阵。

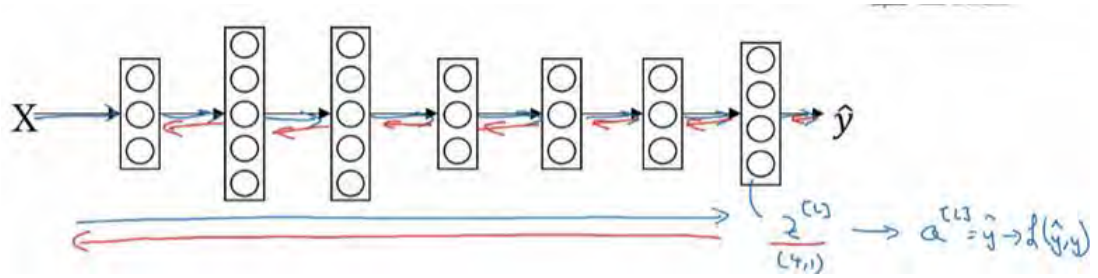


最后我们来看一下，在有 **Softmax** 输出层时如何实现梯度下降法，这个输出层会计算$z^{[l]}$，它是$C\times 1$维的，在这个例子中是 4×1，然后你用 **Softmax** 激活函数来得到$a^{[l]}$或者说y，然后又能由此计算出损失。我们已经讲了如何实现神经网络前向传播的步骤，来得到这些输出，并计算损失，那么反向传播步骤或者梯度下降法又如何呢？其实初始化反向传播所需要的关键步骤或者说关键方程是这个表达式$dz^{[l]}=\hat{y}-y$，你可以用$\hat{y}$这个 4×1 向量减去y这个 4×1 向量，你可以看到这些都会是 4×1 向量，当你有 4 个分类时，在一般情况下就是$C\times 1$，这符合我们对dz的一般定义，这是对$z^{[l]}$损失函数的偏导数（$dz^{[l]}=\frac{\partial J}{\partial z^{[l]}}$），如果你精通微积分就可以自己推导，或者说如果你精通微积分，可以试着自己推导，但如果你需要从零开始

使用这个公式，它也一样有用。



有了这个，你就可以计算$dz^{[l]}$，然后开始反向传播的过程，计算整个神经网络中所需要的所有导数。



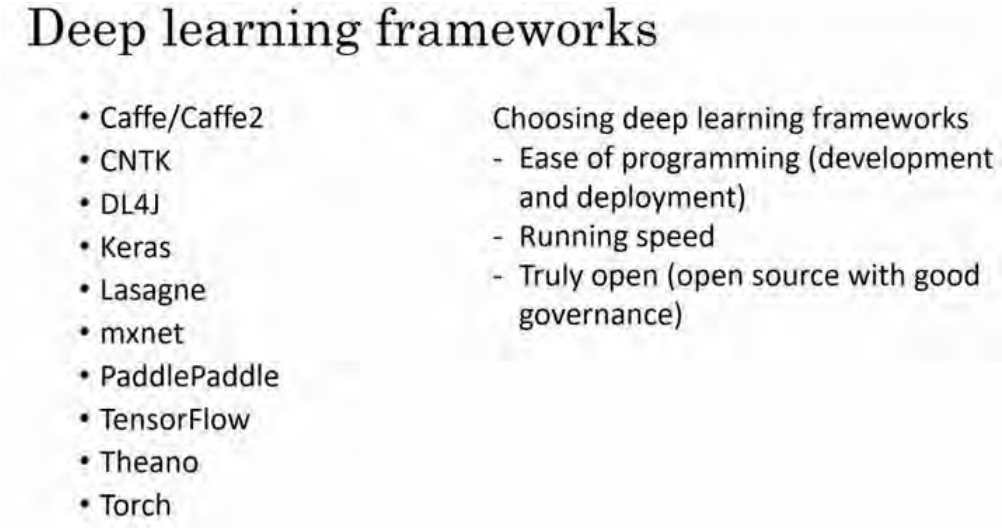
但在这周的初级练习中，我们将开始使用一种深度学习编程框架，对于这些编程框架，通常你只需要专注于把前向传播做对，只要你将它指明为编程框架，前向传播，它自己会弄明白怎样反向传播，会帮你实现反向传播，所以这个表达式值得牢记（$dz^{[l]}=\hat{y}-y$），如果你需要从头开始，实现 **Softmax** 回归或者 **Softmax** 分类，但其实在这周的初级练习中你不会用到它，因为编程框架会帮你搞定导数计算。

Softmax 分类就讲到这里，有了它，你就可以运用学习算法将输入分成不止两类，而是C个不同类别。接下来我想向你展示一些深度学习编程框架，可以让你在实现深度学习算法时更加高效，让我们在下一个视频中一起讨论。

3.10 深度学习框架（Deep Learning frameworks）

你已经差不多从零开始学习了使用 **Python** 和 **NumPy** 实现深度学习算法，很高兴你这样做了，因为我希望你理解这些深度学习算法实际上在做什么。但你会发现，除非应用更复杂的模型，例如卷积神经网络，或者循环神经网络，或者当你开始应用很大的模型，否则它就越来越不实用了，至少对大多数人而言，从零开始全部靠自己实现并不现实。

幸运的是，现在有很多好的深度学习软件框架，可以帮助你实现这些模型。类比一下，我猜你知道如何做矩阵乘法，你还应该知道如何编程实现两个矩阵相乘，但是当你在建很大的应用时，你很可能不想用自己的矩阵乘法函数，而是想要访问一个数值线性代数库，它会更高效，但如果你明白两个矩阵相乘是怎么回事还是挺有用的。我认为现在深度学习已经很成熟了，利用一些深度学习框架会更加实用，会使你的工作更加有效，那就让我们来看下有哪些框架。



现在有许多深度学习框架，能让实现神经网络变得更简单，我们来讲主要的几个。每个框架都针对某一用户或开发群体的，我觉得这里的每一个框架都是某类应用的可靠选择，有很多人写文章比较这些深度学习框架，以及这些深度学习框架发展得有多好，而且因为这些框架往往不断进化，每个月都在进步，如果你想看看关于这些框架的优劣之处的讨论，我留给你自己去网上搜索，但我认为很多框架都在很快进步，越来越好，因此我就不做强烈推荐了，而是与你分享推荐一下选择框架的标准。

一个重要的标准就是便于编程，这既包括神经网络的开发和迭代，还包括为产品进行配置，为了成千上百万，甚至上亿用户的实际使用，取决于你想要做什么。

第二个重要的标准是运行速度，特别是训练大数据集时，一些框架能让你更高效地运行

和训练神经网络。

还有一个标准人们不常提到，但我觉得很重要，那就是这个框架是否真的开放，要是一个框架真的开放，它不仅需要开源，而且需要良好的管理。不幸的是，在软件行业中，一些公司有开源软件的历史，但是公司保持着对软件的全权控制，当几年时间过去，人们开始使用他们的软件时，一些公司开始逐渐关闭曾经开放的资源，或将功能转移到他们专营的云服务中。因此我会注意的一件事就是你能否相信这个框架能长时间保持开源，而不是在一家公司的控制之下，它未来有可能出于某种原因选择停止开源，即便现在这个软件是以开源的形式发布的。但至少在短期内，取决于你对语言的偏好，看你更喜欢 **Python**，**Java** 还是 **C++**或者其它什么，也取决于你在开发的应用，是计算机视觉，还是自然语言处理或者线上广告，等等，我认为这里的多个框架都是很好的选择。

程序框架就讲到这里，通过提供比数值线性代数库更高程度的抽象化，这里的每一个程序框架都能让你在开发深度机器学习应用时更加高效。

3.11 TensorFlow

欢迎来到这周的最后一个视频，有很多很棒的深度学习编程框架，其中一个是 **TensorFlow**，我很期待帮助你开始学习使用 **TensorFlow**，我想在这个视频中向你展示 **TensorFlow** 程序的基本结构，然后让你自己练习，学习更多细节，并运用到本周的编程练习中，这周的编程练习需要花些时间来做，所以请务必留出一些空余时间。

先提一个启发性的问题，假设你有一个损失函数J需要最小化，在本例中，我将使用这个高度简化的损失函数，$Jw = w^2 - 10w + 25$，这就是损失函数，也许你已经注意到该函数其实就是$(w-5)^2$，如果你把这个二次方式子展开就得到了上面的表达式，所以使它最小的w值是 5，但假设我们不知道这点，你只有这个函数，我们来看一下怎样用 **TensorFlow** 将其最小化，因为一个非常类似的程序结构可以用来训练神经网络。其中可以有一些复杂的损失函数$J(w,b)$取决于你的神经网络的所有参数，然后类似的，你就能用 **TensorFlow** 自动找到使损失函数最小的w和b的值。但让我们先从左边这个更简单的例子入手。

```
In [1]: import numpy as np
        import tensorflow as tf

In [2]: w=tf.Variable(0,dtype=tf.float32)
        cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
        train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)
        init=tf.global_variables_initializer()
        session=tf.Session()
        session.run(init)
        print(session.run(w))

        0.0
```

我在我的 **Jupyter notebook** 中运行 **Python**，

```
import numpy as np
import tensorflow as tf
#导入 TensorFlow
w = tf.Variable(0,dtype = tf.float32)
#接下来，让我们定义参数 w，在 TensorFlow 中，你要用 tf.Variable()来定义参数
#然后我们定义损失函数：
cost = tf.add(tf.add(w**2,tf.multiply(- 10.,w)),25)
#然后我们定义损失函数 J
```

#然后我们再写：

```
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
```

#(让我们用 0.01 的学习率，目标是最小化损失)。

#最后下面的几行是惯用表达式：

```
init = tf.global_variables_initializer()
session = tf.Sessions()#这样就开启了一个 TensorFlow session。
session.run(init)#来初始化全局变量。
```

#然后让 TensorFlow 评估一个变量，我们要用到：

```
session.run(w)
```

#上面的这一行将 w 初始化为 0，并定义损失函数，我们定义 train 为学习算法，它用梯度下降法优化器使损失函数最小化，但实际上我们还没有运行学习算法，所以#上面的这一行将 w 初始化为 0，并定义损失函数，我们定义 train 为学习算法，它用梯度下降法优化器使损失函数最小化，但实际上我们还没有运行学习算法，所以session.run(w)评估了 w，让我：：

```
print(session.run(w))
```

#所以如果我们运行这个，它评估w等于 0，因为我们什么都还没运行。

#现在让我们输入：

```
session.run(train)，它所做的就是运行一步梯度下降法。
```

#接下来在运行了一步梯度下降法后，让我们评估一下 w 的值，再 print：

```
print(session.run(w))
```

#在一步梯度下降法之后，w 现在是 0.1。

```
In [3]: session.run(train)
        print(session.run(w))

        0.1
```

现在我们运行梯度下降 1000 次迭代：

```
In [4]: for i in range(1000):
            session.run(train)
        print(session.run(w))

        4.99999
```

这是运行了梯度下降的 1000 次迭代，最后w变成了 4.99999，记不记得我们说$(w-5)^2$最小化，因此w的最优值是 5，这个结果已经很接近了。

希望这个让你对 **TensorFlow** 程序的大致结构有了了解，当你做编程练习，使用更多 **TensorFlow** 代码时，我这里用到的一些函数你会熟悉起来，这里有个地方要注意，w是我们想要优化的参数，因此将它称为变量，注意我们需要做的就是定义一个损失函数，使用这些 `add` 和 `multiply` 之类的函数。**TensorFlow** 知道如何对 `add` 和 `mutiply`，还有其它函数求导，这就是为什么你只需基本实现前向传播，它能弄明白如何做反向传播和梯度计算，因为它已经内置在 `add`，`multiply` 和平方函数中。

对了，要是觉得这种写法不好看的话，**TensorFlow** 其实还重载了一般的加减运算等等，因此你也可以把$cost$写成更好看的形式，把之前的 `cost` 标成注释，重新运行，得到了同样的结果。

```
In [5]: w=tf.Variable(0,dtype=tf.float32)
        #cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
        cost=w**2-10*w+25
        train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)

        init=tf.global_variables_initializer()
        session=tf.Session()
        session.run(init)
        print(session.run(w))

        0.0

In [6]: session.run(train)
        print(session.run(w))

        0.1

In [7]: for i in range(1000):
            session.run(train)
        print(session.run(w))

        4.99999
```

一旦w被称为 **TensorFlow** 变量，平方，乘法和加减运算都重载了，因此你不必使用上面这种不好看的句法。

TensorFlow 还有一个特点，我想告诉你，那就是这个例子将w的一个固定函数最小化了。如果你想要最小化的函数是训练集函数又如何呢？不管你有什么训练数据x，当你训练神经网络时，训练数据x会改变，那么如何把训练数据加入 **TensorFlow** 程序呢？

我会定义x，把它想做扮演训练数据的角色，事实上训练数据有x和y，但这个例子中只

有x，把x定义为：

`x = tf.placeholder(tf.float32,[3,1])`，让它成为[3,1]数组，我要做的就是，因为$cost$这个二次方程的三项前有固定的系数，它是$w^2+10w+25$，我们可以把这些数字 1，-10 和 25 变成数据，我要做的就是把$cost$替换成：

`cost = x[0][0]*w**2 +x[1][0]*w + x[2][0]`，现在x变成了控制这个二次函数系数的数据，这个 **placeholder** 函数告诉 **TensorFlow**，你稍后会为x提供数值。

让我们再定义一个数组，`coefficient = np.array([[1.],[-10.],[25.]])`，这就是我们要接入x的数据。最后我们需要用某种方式把这个系数数组接入变量x，做到这一点的句法是，在训练这一步中，要提供给x的数值，我在这里设置：

`feed_dict = {x:coefficients}`

好了，希望没有语法错误，我们重新运行它，希望得到和之前一样的结果。

```
In [11]: coefficients=np.array([[1.],[-10.],[25.]])

         w=tf.Variable(0,dtype=tf.float32)
         x=tf.placeholder(tf.float32,[3,1])
         #cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
         #cost=w**2-10*w+25
         cost=x[0][0]*w**2+x[1][0]*w+x[2][0]
         train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)

         init=tf.global_variables_initializer()
         session=tf.Session()
         session.run(init)
         print(session.run(w))

         0.0

In [12]: session.run(train,feed_dict={x:coefficients})
         print(session.run(w))

         0.1

In [13]: for i in range(1000):
             session.run(train,feed_dict={x:coefficients})
         print(session.run(w))

         4.99999
```

现在如果你想改变这个二次函数的系数，假设你把：

`coefficient = np.array([[1.],[-10.],[25.]])`

改为：

`coefficient = np.array([[1.],[-20.],[100.]])`

现在这个函数就变成了$(w-10)^2$，如果我重新运行，希望我得到的使$(w-10)^2$最小化的w值为 10，让我们看一下，很好，在梯度下降 1000 次迭代之后，我们得到接近 10 的w。

```
In [14]: coefficients=np.array([[1.],[-20.],[100.]])

         w=tf.Variable(0,dtype=tf.float32)
         x=tf.placeholder(tf.float32,[3,1])
         #cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
         #cost=w**2-10*w+25
         cost=x[0][0]*w**2+x[1][0]*w+x[2][0]
         train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)

         init=tf.global_variables_initializer()
         session=tf.Session()
         session.run(init)
         print(session.run(w))

         0.0

In [15]: session.run(train,feed_dict={x:coefficients})
         print(session.run(w))

         0.2

In [16]: for i in range(1000):
             session.run(train,feed_dict={x:coefficients})
         print(session.run(w))

         9.99998
```

在你做编程练习时，见到更多的是，**TensorFlow** 中的 **placeholder** 是一个你之后会赋值的变量，这种方式便于把训练数据加入损失方程，把数据加入损失方程用的是这个句法，当你运行训练迭代，用 `feed_dict` 来让 `x=coefficients`。如果你在做 **mini-batch** 梯度下降，在每次迭代时，你需要插入不同的 **mini-batch**，那么每次迭代，你就用 `feed_dict` 来喂入训练集的不同子集，把不同的 **mini-batch** 喂入损失函数需要数据的地方。

希望这让你了解了 **TensorFlow** 能做什么，让它如此强大的是，你只需说明如何计算损失函数，它就能求导，而且用一两行代码就能运用梯度优化器，**Adam** 优化器或者其他优化器。

Code example

```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0],dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]     # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

for i in range(1000):
    session.run(train, feed_dict={x:coefficients})
print(session.run(w))
```

```
with tf.Session() as session:
    session.run(init)
    print(session.run(w))
```

这还是刚才的代码，我稍微整理了一下，尽管这些函数或变量看上去有点神秘，但你在做编程练习时多练习几次就会熟悉起来了。

```
session = tf.Session()
session.run(init)
print(session.run(w))
```

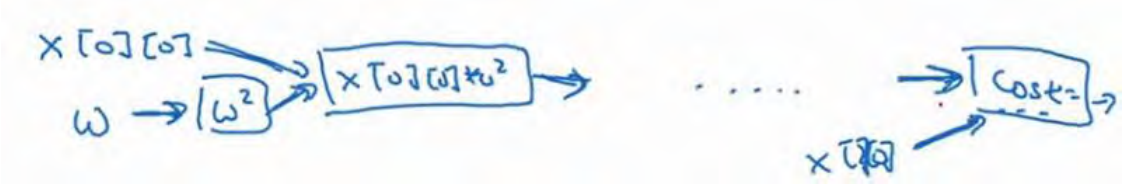
```
with tf.Session() as session:
    session.run(init)
    print(session.run(w))
```

还有最后一点我想提一下，这三行（蓝色大括号部分）在 **TensorFlow** 里是符合表达习惯的，有些程序员会用这种形式来替代，作用基本上是一样的。

但这个 **with** 结构也会在很多 **TensorFlow** 程序中用到，它的意思基本上和左边的相同，但是 **Python** 中的 **with** 命令更方便清理，以防在执行这个内循环时出现错误或例外。所以你也会在编程练习中看到这种写法。那么这个代码到底做了什么呢？让我们看这个等式：

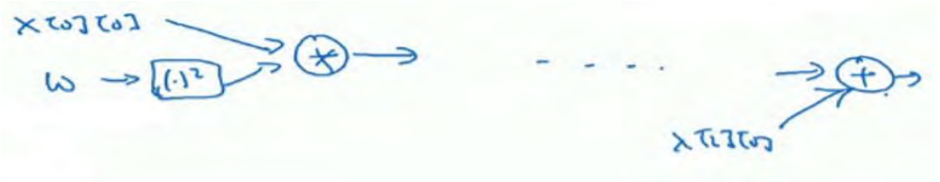
```
cost = x[0][0]*w**2 +x[1][0]*w + x[2][0]#(w-5)**2
```

TensorFlow 程序的核心是计算损失函数，然后 **TensorFlow** 自动计算出导数，以及如何最小化损失，因此这个等式或者这行代码所做的就是让 **TensorFlow** 建立计算图，计算图所做的就是取$x[0][0]$，取w，然后将它平方，然后$x[0][0]$和w^2相乘，你就得到了$x[0][0]*w^2$，以此类推，最终整个建立起来计算$cost = [0][0]*w**2 + x[1][0]*w + x[2][0]$，最后你得到了损失函数。



TensorFlow 的优点在于，通过用这个计算损失，计算图基本实现前向传播，**TensorFlow** 已经内置了所有必要的反向函数，回忆一下训练深度神经网络时的一组前向函数和一组反向函数，而像 **TensorFlow** 之类的编程框架已经内置了必要的反向函数，这也是为什么通过内置函数来计算前向函数，它也能自动用反向函数来实现反向传播，即便函数非常复杂，再帮你计算导数，这就是为什么你不需要明确实现反向传播，这是编程框架能帮你变得高效的原因之一。

如果你看 **TensorFlow** 的使用说明，我只是指出 **TensorFlow** 的说明用了一套和我不太一样的符号来画计算图，它用了$x[0][0]$，w，然后它不是写出值，想这里的w^2，**TensorFlow** 使用说明倾向于只写运算符，所以这里就是平方运算，而这两者一起指向乘法运算，以此类推，然后在最后的节点，我猜应该是一个将$x[2][0]$加上去得到最终值的加法运算。



为本课程起见，我认为计算图用第一种方式会更容易理解，但是如果你去看 **TensorFlow** 的使用说明，如果你看到说明里的计算图，你会看到另一种表示方式，节点都用运算来标记而不是值，但这两种呈现方式表达的是同样的计算图。

在编程框架中你可以用一行代码做很多事情，例如，你不想用梯度下降法，而是想用 **Adam** 优化器，你只要改变这行代码，就能很快换掉它，换成更好的优化算法。所有现代深度学习编程框架都支持这样的功能，让你很容易就能编写复杂的神经网络。

我希望我帮助你了解了 **TensorFlow** 程序典型的结构，概括一下这周的内容，你学习了如何系统化地组织超参数搜索过程，我们还讲了 **Batch** 归一化，以及如何用它来加速神经网络的训练，最后我们讲了深度学习的编程框架，有很多很棒的编程框架，这最后一个视频我

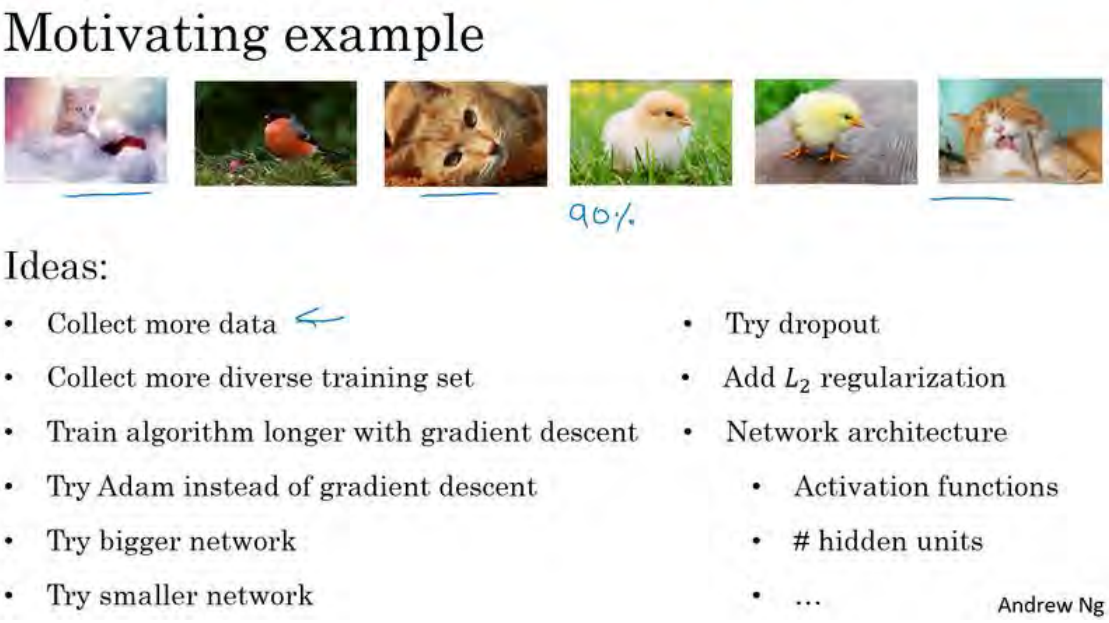
们重点讲了 **TensorFlow**。有了它，我希望你享受这周的编程练习，帮助你更熟悉这些概念。

第三门课 结构化机器学习项目（Structuring Machine Learning Projects）

第一周 机器学习（ML）策略（1）（ML strategy（1））

1.1 为什么是 ML 策略？（Why ML Strategy?）

大家好，欢迎收听本课，如何构建你的机器学习项目也就是说机器学习的策略。我希望通过这门课程你们能够学到如何更快速高效地优化你的机器学习系统。那么，什么是机器学习策略呢？



我们从一个启发性的例子开始讲，假设你正在调试你的猫分类器，经过一段时间的调整，你的系统达到了 90%准确率，但对你的应用程序来说还不够好。

你可能有很多想法去改善你的系统，比如，你可能想我们去收集更多的训练数据吧。或者你会说，可能你的训练集的多样性还不够，你应该收集更多不同姿势的猫咪图片，或者更多样化的反例集。或者你想再用梯度下降训练算法，训练久一点。或者你想尝试用一个完全不同的优化算法，比如 **Adam** 优化算法。或者尝试使用规模更大或者更小的神经网络。或者你想试试 **dropout** 或者$L2$正则化。或者你想修改网络的架构，比如修改激活函数，改变隐藏单元的数目之类的方法。

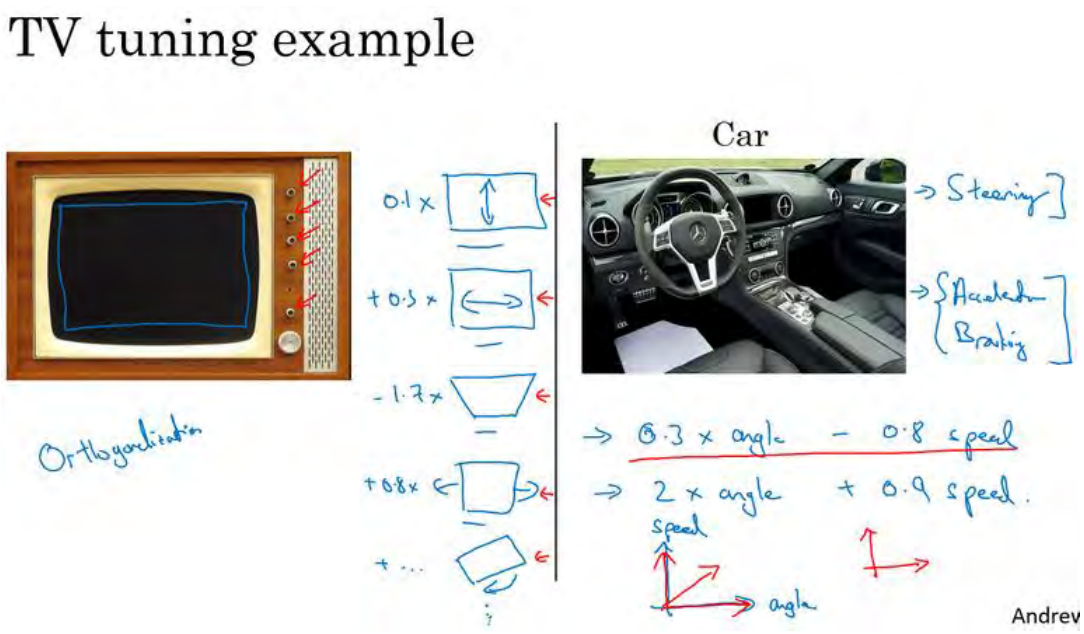
当你尝试优化一个深度学习系统时，你通常可以有很多想法可以去试，问题在于，如果

你做出了错误的选择，你完全有可能白费 6 个月的时间，往错误的方向前进，在 6 个月之后才意识到这方法根本不管用。比如，我见过一些团队花了 6 个月时间收集更多数据，却在 6 个月之后发现，这些数据几乎没有改善他们系统的性能。所以，假设你的项目没有 6 个月的时间可以浪费，如果有快速有效的方法能够判断哪些想法是靠谱的，或者甚至提出新的想法，判断哪些是值得一试的想法，哪些是可以放心舍弃的。

我希望在这门课程中，可以教给你们一些策略，一些分析机器学习问题的方法，可以指引你们朝着最有希望的方向前进。这门课中，我会和你们分享我在搭建和部署大量深度学习产品时学到的经验和教训，我想这些内容是这门课程独有的。比如说，很多大学深度学习课程很少提到这些策略。事实上，机器学习策略在深度学习的时代也在变化，因为现在对于深度学习算法来说能够做到的事情，比上一代机器学习算法大不一样。我希望这些策略能帮助你们提高效率，让你们的深度学习系统更快投入实用。

1.2 正交化（Orthogonalization）

搭建建立机器学习系统的挑战之一是，你可以尝试和改变的东西太多太多了。包括，比如说，有那么多的超参数可以调。我留意到，那些效率很高的机器学习专家有个特点，他们思维清晰，对于要调整什么来达到某个效果，非常清楚，这个步骤我们称之为正交化，让我告诉你是什么意思吧。



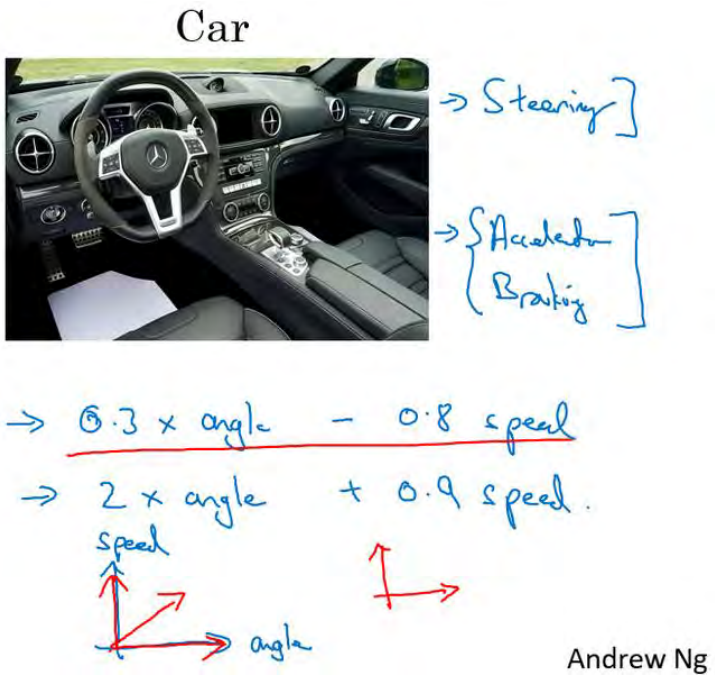
这是一张老式电视图片，有很多旋钮可以用来调整图像的各种性质，所以对于这些旧式电视，可能有一个旋钮用来调图像垂直方向的高度，另外有一个旋钮用来调图像宽度，也许还有一个旋钮用来调梯形角度，还有一个旋钮用来调整图像左右偏移，还有一个旋钮用来调图像旋转角度之类的。电视设计师花了大量时间设计电路，那时通常都是模拟电路来确保每个旋钮都有相对明确的功能。如一个旋钮来调整这个（高度），一个旋钮调整这个（宽度），一个旋钮调整这个（梯形角度），以此类推。

相比之下，想象一下，如果你有一个旋钮调的是$0.1x$表示图像高度，$+0.3x$表示图像宽度，$-1.7x$表示梯形角度，$+0.8x$表示图像在水平轴上的坐标之类的。如果你调整这个（其中一个）旋钮，那么图像的高度、宽度、梯形角度、平移位置全部都会同时改变，如果你有这样的旋钮，那几乎不可能把电视调好，让图像显示在区域正中。

所以在这种情况下，正交化指的是电视设计师设计这样的旋钮，使得每个旋钮都只调整一个性质，这样调整电视图像就容易得多，就可以把图像调到正中。

接下来是另一个正交化例子，你想想学车的时候，一辆车有三个主要控制，第一是方向

盘，方向盘决定你往左右偏多少，还有油门和刹车。就是这三个控制，其中一个控制方向，另外两个控制你的速度，这样就比较容易解读。知道不同控制的不同动作会对车子运动有什么影响。



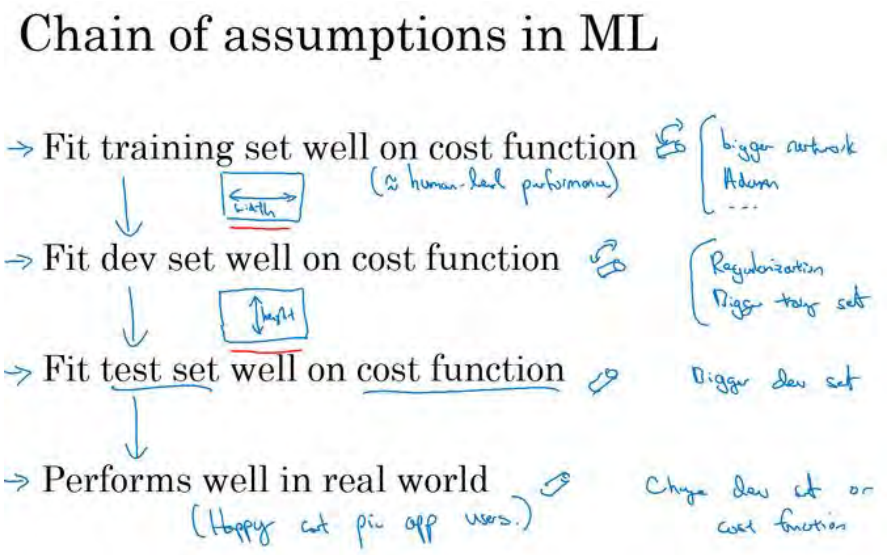
想象一下，如果有人这么造车，造了个游戏手柄，手柄的一个轴控制的是0.3 ×转向角-速度，然后还有一个轴控制的是2 ×转向角+0.9 ×车速，理论上来说，通过调整这两个旋钮你是可以将车子调整到你希望得到的角度和速度，但这样比单独控制转向角度，分开独立的速度控制要难得多。

所以正交化的概念是指，你可以想出一个维度，这个维度你想做的是控制转向角，还有另一个维度来控制你的速度，那么你就需要一个旋钮尽量只控制转向角，另一个旋钮，在这个开车的例子里其实是油门和刹车控制了你的速度。但如果你有一个控制旋钮将两者混在一起，比如说这样一个控制装置同时影响你的转向角和速度，同时改变了两个性质，那么就很难令你的车子以想要的速度和角度前进。然而正交化之后，正交意味着互成 90 度。设计出正交化的控制装置，最理想的情况是和你实际想控制的性质一致，这样你调整参数时就容易得多。可以单独调整转向角，还有你的油门和刹车，令车子以你想要的方式运动。

那么这与机器学习有什么关系呢？要弄好一个监督学习系统，你通常需要调你的系统的旋钮。

确保四件事情，首先，你通常必须确保至少系统在训练集上得到的结果不错，所以训练集上的表现必须通过某种评估，达到能接受的程度，对于某些应用，这可能意味着达到人类

水平的表现，但这取决于你的应用，我们将在下周更多地谈谈如何与人类水平的表现进行比较。但是，在训练集上表现不错之后，你就希望系统也能在开发集上有好的表现，然后你希望系统在测试集上也有好的表现。在最后，你希望系统在测试集上系统的成本函数在实际使用中表现令人满意，比如说，你希望这些猫图片应用的用户满意。



我们回到电视调节的例子，如果你的电视图像太宽或太窄，你想要一个旋钮去调整，你可不想要仔细调节五个不同的旋钮，它们也会影响别的图像性质，你只需要一个旋钮去改变电视图像的宽度。

所以类似地，如果你的算法在成本函数上不能很好地拟合训练集，你想要一个旋钮，是的我画这东西表示旋钮，或者一组特定的旋钮，这样你可以用来确保你的可以调整你的算法，让它很好地拟合训练集，所以你用来调试的旋钮是你可能可以训练更大的网络，或者可以切换到更好的优化算法，比如 **Adam** 优化算法，等等。我们将在本周和下周讨论一些其他选项。

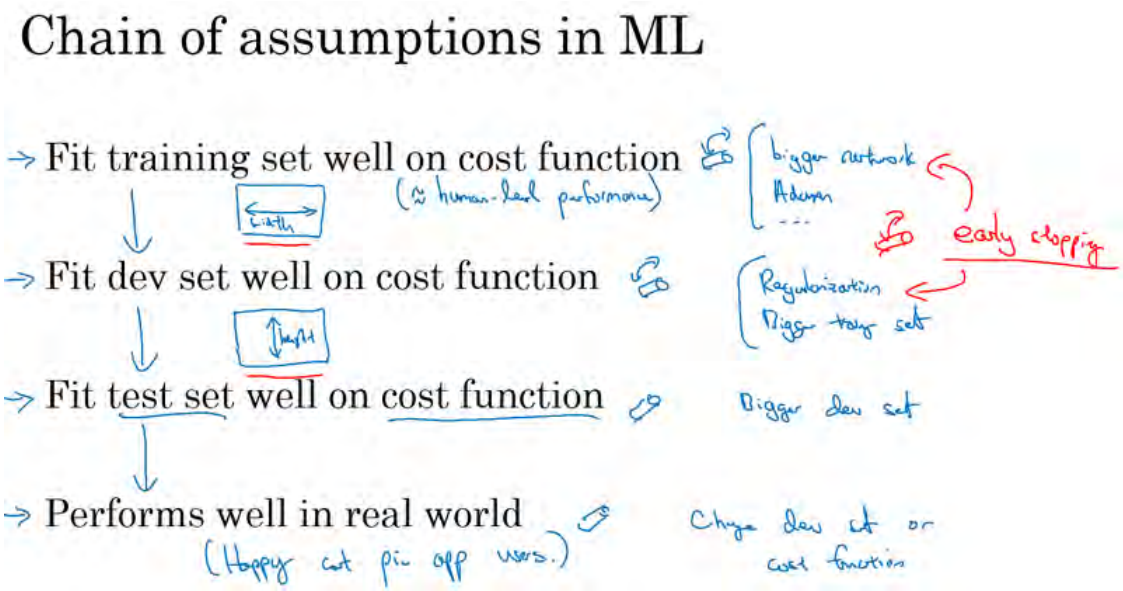
相比之下，如果发现算法对开发集的拟合很差，那么应该有独立的一组旋钮，是的，这就是我画得毛毛躁躁的另一个旋钮，你希望有一组独立的旋钮去调试。比如说，你的算法在开发集上做的不好，它在训练集上做得很好，但开发集不行，然后你有一组正则化的旋钮可以调节，尝试让系统满足第二个条件。类比到电视，就是现在你调好了电视的宽度，如果图像的高度不太对，你就需要一个不同的旋钮来调节电视图像的高度，然后你希望这个旋钮尽量不会影响到电视的宽度。增大训练集可以是另一个可用的旋钮，它可以帮助你的学习算法更好地归纳开发集的规律，现在调好了电视图像的高度和宽度。

如果它不符合第三个标准呢？如果系统在开发集上做的很好，但测试集上做得不好呢？如果是这样，那么你需要调的旋钮，可能是更大的开发集。因为如果它在开发集上做的不错，

但测试集不行这可能意味着你对开发集过拟合了，你需要往回退一步，使用更大的开发集。

最后，如果它在测试集上做得很好，但无法给你的猫图片应用用户提供良好的体验，这意味着你需要回去，改变开发集或成本函数。因为如果根据某个成本函数，系统在测试集上做的很好，但它无法反映你的算法在现实世界中的表现，这意味着要么你的开发集分布设置不正确，要么你的成本函数测量的指标不对。

我们很快会逐一讲到这些例子，我们以后会详细介绍这些特定的旋钮，在本周和下周晚些时候会介绍的。所以如果现在你无法理解全部细节，别担心，但我希望你们对这种正交化过程有个概念。你要非常清楚，到底是四个问题中的哪一个，知道你可以调节哪些不同的东西尝试解决那个问题。



当我训练神经网络时，我一般不用早期停止，这个技巧也还不错，很多人都这么干。但个人而言，我觉得早期停止有点难以分析，因为这个旋钮会同时影响你对训练集的拟合，因为如果你早期停止，那么对训练集的拟合就不太好，但它同时也用来改善开发集的表现，所以这个旋钮没那么正交化。因为它同时影响两件事情，就像一个旋钮同时影响电视图像的宽度和高度。不是说这样就不要用，如果你想用也是可以的。但如果你有更多的正交化控制，比如我这里写出的其他手段，用这些手段调网络会简单不少。

所以我希望你们对正交化的意义有点概念，就像你看电视图像一样。如果你说，我的电视图像太宽，所以我要调整这个旋钮（宽度旋钮）。或者它太高了，所以我要调整那个旋钮（高度旋钮）。或者它太梯形了，所以我要调整这个旋钮（梯形角度旋钮），这就很好。

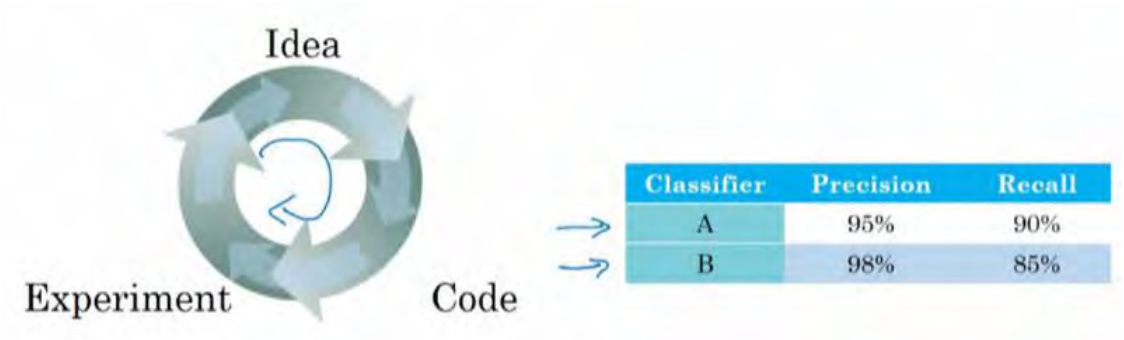
在机器学习中，如果你可以观察你的系统，然后说这一部分是错的，它在训练集上做的不好、在开发集上做的不好、它在测试集上做的不好，或者它在测试集上做的不错，但在现

实世界中不好，这就很好。必须弄清楚到底是什么地方出问题了，然后我们刚好有对应的旋钮，或者一组对应的旋钮，刚好可以解决那个问题，那个限制了机器学习系统性能的问题。

这就是我们这周和下周要讲到的，如何诊断出系统性能瓶颈到底在哪。还有找到你可以用的一组特定的旋钮来调整你的系统，来改善它特定方面的性能，我们开始详细讲讲这个过程吧。

1.3 单一数字评估指标（Single number evaluation metric）

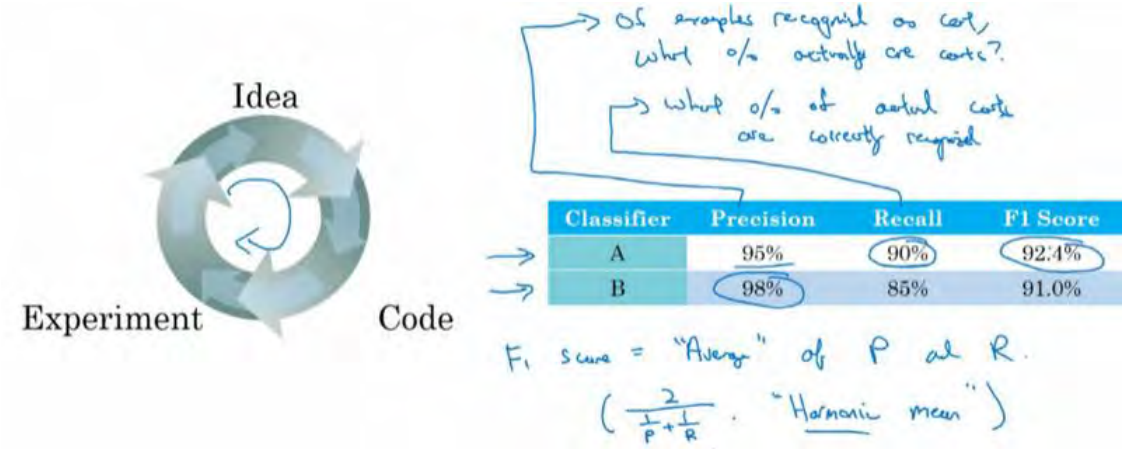
无论你是调整超参数，或者是尝试不同的学习算法，或者在搭建机器学习系统时尝试不同手段，你会发现，如果你有一个单实数评估指标，你的进展会快得多，它可以快速告诉你，新尝试的手段比之前的手段好还是差。所以当团队开始进行机器学习项目时，我经常推荐他们为问题设置一个单实数评估指标。



Classifier	Precision	Recall
A	95%	90%
B	98%	85%

我们来看一个例子，你之前听过我说过，应用机器学习是一个非常经验性的过程，我们通常有一个想法，编程序，跑实验，看看效果如何，然后使用这些实验结果来改善你的想法，然后继续走这个循环，不断改进你的算法。

比如说对于你的猫分类器，之前你搭建了某个分类器A，通过改变超参数，还有改变训练集等手段，你现在训练出来了一个新的分类器 B，所以评估你的分类器的一个合理方式是观察它的查准率（**precision**）和查全率（**recall**）。



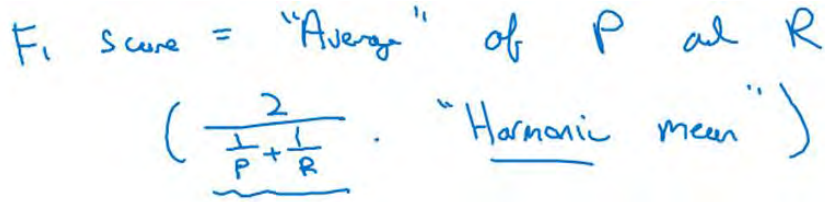
查准率和查全率的确切细节对于这个例子来说不太重要。但简而言之，查准率的定义是在你的分类器标记为猫的例子中，有多少真的是猫。所以如果分类器A有 95%的查准率，这意味着你的分类器说这图有猫的时候，有 95%的机会真的是猫。

查全率就是，对于所有真猫的图片，你的分类器正确识别出了多少百分比。实际为猫的

图片中，有多少被系统识别出来？如果分类器A查全率是 90%，这意味着对于所有的图像，比如说你的开发集都是真的猫图，分类器A准确地分辨出了其中的 90%。

所以关于查准率和查全率的定义，不用想太多。事实证明，查准率和查全率之间往往需要折衷，两个指标都要顾及到。你希望得到的效果是，当你的分类器说某个东西是猫的时候，有很大的机会它真的是一只猫，但对于所有是猫的图片，你也希望系统能够将大部分分类为猫，所以用查准率和查全率来评估分类器是比较合理的。

但使用查准率和查全率作为评估指标的时候，有个问题，如果分类器A在查全率上表现更好，分类器B在查准率上表现更好，你就无法判断哪个分类器更好。如果你尝试了很多不同想法，很多不同的超参数，你希望能够快速试验不仅仅是两个分类器，也许是十几个分类器，快速选出“最好的”那个，这样你可以从那里出发再迭代。如果有两个评估指标，就很难去快速地二中选一或者十中选一，所以我并不推荐使用两个评估指标，查准率和查全率来选择一个分类器。你只需要找到一个新的评估指标，能够结合查准率和查全率。

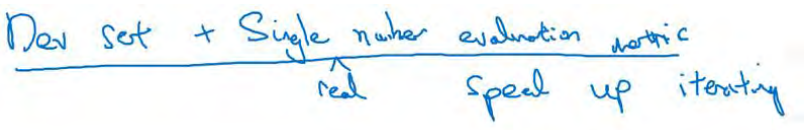


在机器学习文献中，结合查准率和查全率的标准方法是所谓的F_1分数，F_1分数的细节并不重要。但非正式的，你可以认为这是查准率P和查全率R的平均值。正式来看，F_1分数的定义是这个公式：$\frac{2}{\frac{1}{P}+\frac{1}{R}}$

在数学中，这个函数叫做查准率P和查全率R的调和平均数。但非正式来说，你可以将它看成是某种查准率和查全率的平均值，只不过你算的不是直接的算术平均，而是用这个公式定义的调和平均。这个指标在权衡查准率和查全率时有一些优势。

Classifier	Precision	Recall	F1 Score
A	95%	90%	92.4%
B	98%	85%	91.0%

但在这个例子中，你可以马上看出，分类器A的F_1分数更高。假设F_1分数是结合查准率和查全率的合理方式，你可以快速选出分类器A，淘汰分类器B。



我发现很多机器学习团队就是这样，有一个定义明确的开发集用来测量查准率和查全率，再加上这样一个单一数值评估指标，有时我叫单实数评估指标，能让你快速判断分类器A或者分类器B更好。所以有这样一个开发集，加上单实数评估指标，你的迭代速度肯定会很快，它可以加速改进您的机器学习算法的迭代过程。

Algorithm	US	China	India	Other
A	3%	7%	5%	9%
B	5%	6%	5%	10%
C	2%	3%	4%	5%
D	5%	8%	7%	2%
E	4%	5%	2%	4%
F	7%	11%	8%	12%

我们来看另一个例子，假设你在开发一个猫应用来服务四个地理大区的爱猫人士，美国、中国、印度还有世界其他地区。我们假设你的两个分类器在来自四个地理大区的数据中得到了不同的错误率，比如算法A在美国用户上传的图片中达到了 3%错误率，等等。

所以跟踪一下，你的分类器在不同市场和地理大区中的表现应该是有用的，但是通过跟踪四个数字，很难扫一眼这些数值就快速判断算法A或算法B哪个更好。如果你测试很多不同的分类器，那么看着那么多数字，然后快速选一个最优是很难的。所以在这个例子中，我建议，除了跟踪分类器在四个不同的地理大区的表现，也要算算平均值。假设平均表现是一个合理的单实数评估指标，通过计算平均值，你就可以快速判断。

Algorithm	US	China	India	Other	Average
A	3%	7%	5%	9%	6%
B	5%	6%	5%	10%	6.5%
C	2%	3%	4%	5%	3.5%
D	5%	8%	7%	2%	5.25%
E	4%	5%	2%	4%	3.75%
F	7%	11%	8%	12%	9.5%

看起来算法C的平均错误率最低，然后你可以继续用那个算法。你必须选择一个算法，然后不断迭代，所以你的机器学习的工作流程往往是你有一个想法，你尝试实现它，看看这个想法好不好。

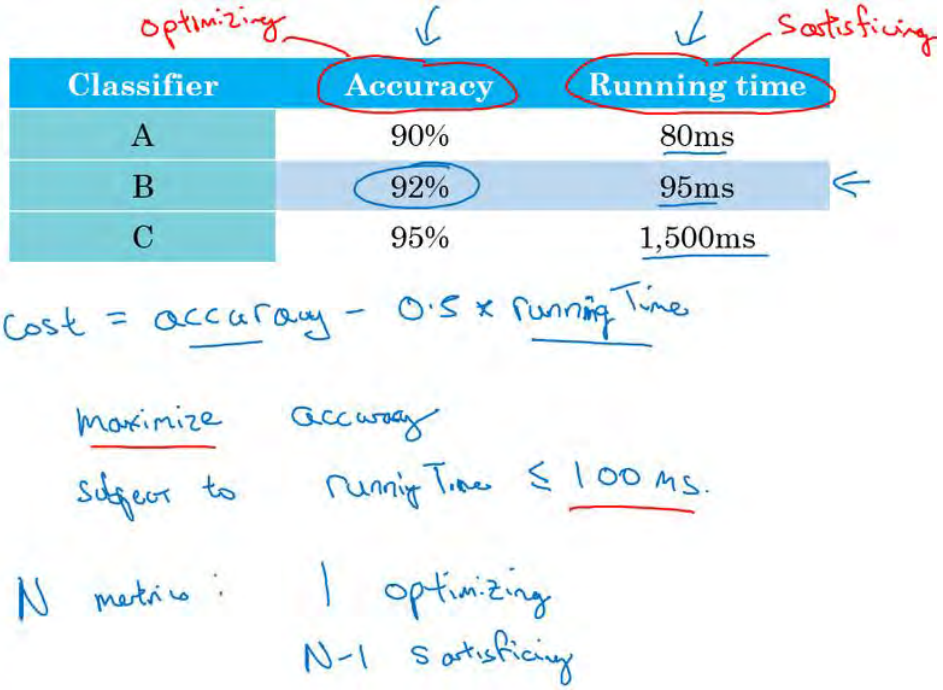
所以本视频介绍的是，有一个单实数评估指标真的可以提高你的效率，或者提高你的团队做出这些决策的效率。现在我们还没有完整讨论如何有效地建立评估指标。在下一个视频中，我会教你们如何设置优化以及满足指标，我们来看下一段视频。

1.4 满足和优化指标（Satisficing and optimizing metrics）

要把你顾及到的所有事情组合成单实数评估指标有时并不容易，在那些情况里，我发现有时候设立满足和优化指标是很重要的，让我告诉你是什么意思吧。

Classifier	Accuracy	Running time
A	90%	80ms
B	92%	95ms
C	95%	1,500ms

假设你已经决定你很看重猫分类器的分类准确度，这可以是F_1分数或者用其他衡量准确度的指标。但除了准确度之外，我们还需要考虑运行时间，就是需要多长时间来分类一张图。分类器A需要 80 毫秒，B需要 95 毫秒，C 需要 1500 毫秒，就是说需要 1.5 秒来分类图像。



你可以这么做，将准确度和运行时间组合成一个整体评估指标。所以成本，比如说，总体成本是$cost = accuracy - 0.5 \times \text{runningTime}$，这种组合方式可能太刻意，只用这样的公式来组合准确度和运行时间，两个数值的线性加权求和。

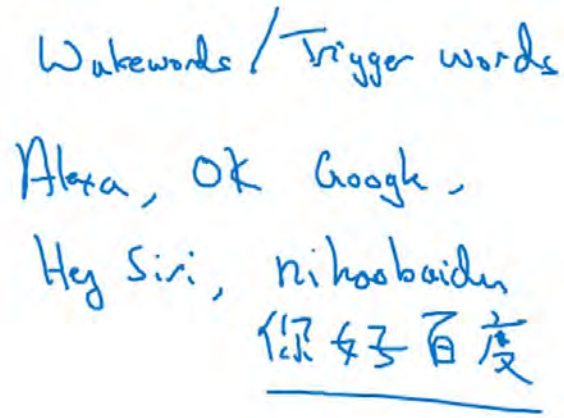
你还可以做其他事情，就是你可能选择一个分类器，能够最大限度提高准确度，但必须满足运行时间要求，就是对图像进行分类所需的时间必须小于等于 100 毫秒。所以在这种情况下，我们就说准确度是一个优化指标，因为你想要准确度最大化，你想做的尽可能准确，但是运行时间就是我们所说的满足指标，意思是它必须足够好，它只需要小于 100 毫秒，达

到之后，你不在乎这指标有多好，或者至少你不会那么在乎。所以这是一个相当合理的权衡方式，或者说将准确度和运行时间结合起来的方式。实际情况可能是，只要运行时间少于 100 毫秒，你的用户就不会在乎运行时间是 100 毫秒还是 50 毫秒，甚至更快。

Classifier	Accuracy	Running time
A	90%	80ms
B	92%	95ms
C	95%	1,500ms

通过定义优化和满足指标，就可以给你提供一个明确的方式，去选择“最好的”分类器。在这种情况下分类器 B 最好，因为在所有的运行时间都小于 100 毫秒的分类器中，它的准确度最好。

所以更一般地说，如果你要考虑N个指标，有时候选择其中一个指标做为优化指标是合理的。所以你想尽量优化那个指标，然后剩下$N-1$个指标都是满足指标，意味着只要它们达到一定阈值，例如运行时间快于 100 毫秒，但只要达到一定的阈值，你不在乎它超过那个门槛之后的表现，但它们必须达到这个门槛。

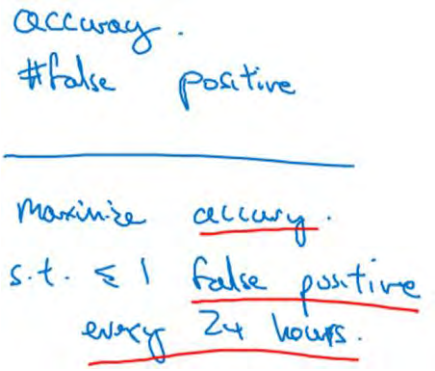


这里是另一个例子，假设你正在构建一个系统来检测唤醒语，也叫触发词，这指的是语音控制设备。比如亚马逊 **Echo**，你会说“**Alexa**”，或者用“**Okay Google**”来唤醒谷歌设备，或者对于苹果设备，你会说“**Hey Siri**”，或者对于某些百度设备，我们用“你好百度”唤醒。

对的，这些就是唤醒词，可以唤醒这些语音控制设备，然后监听你想说的话。所以你可

能会在乎触发字检测系统的准确性，所以当有人说出其中一个触发词时，有多大概率可以唤醒你的设备。

你可能也需要顾及假阳性（**false positive**）的数量，就是没有人在说这个触发词时，它被随机唤醒的概率有多大？所以这种情况下，组合这两种评估指标的合理方式可能是最大化精确度。所以当某人说出唤醒词时，你的设备被唤醒的概率最大化，然后必须满足 24 小时内最多只能有 1 次假阳性，对吧？所以你的设备平均每天只会没有人真的在说话时随机唤醒一次。所以在这种情况下，准确度是优化指标，然后每 24 小时发生一次假阳性是满足指标，你只要每 24 小时最多有一次假阳性就满足了。



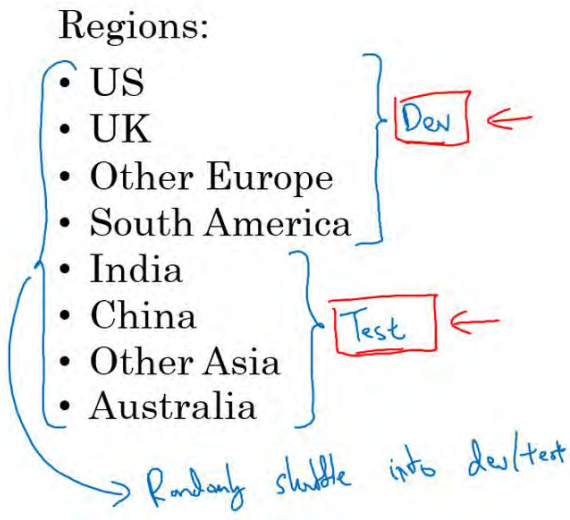
总结一下，如果你需要顾及多个指标，比如说，有一个优化指标，你想尽可能优化的，然后还有一个或多个满足指标，需要满足的，需要达到一定的门槛。现在你就有一个全自动的方法，在观察多个成本大小时，选出"最好的"那个。现在这些评估指标必须是在训练集或开发集或测试集上计算或求出来的。所以你还需要做一件事，就是设立训练集、开发集，还有测试集。在下一个视频里，我想和大家分享一些如何设置训练、开发和测试集的指导方针，我们下一个视频继续。

1.5 训练/开发/测试集划分（Train/dev/test distributions）

设立训练集，开发集和测试集的方式大大影响了你或者你的团队在建立机器学习应用方面取得进展的速度。同样的团队，即使是大公司里的团队，在设立这些数据集的方式，真的会让团队的进展变慢而不是加快，我们看看应该如何设立这些数据集，让你的团队效率最大化。

在这个视频中，我想集中讨论如何设立开发集和测试集，开发（**dev**）集也叫做开发集（**development set**），有时称为保留交叉验证集（**hold out cross validation set**）。然后，机器学习中的工作流程是，你尝试很多思路，用训练集训练不同的模型，然后使用开发集来评估不同的思路，然后选择一个，然后不断迭代去改善开发集的性能，直到最后你可以得到一个令你满意的成本，然后你再用测试集去评估。

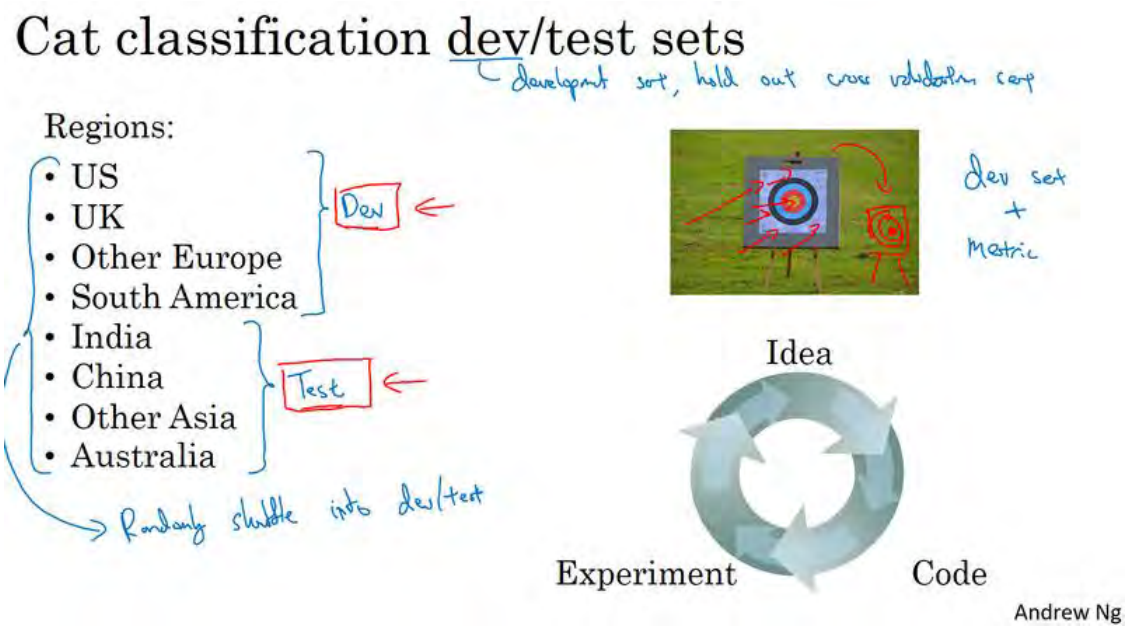
现在，举个例子，你要开发一个猫分类器，然后你在这些区域里运营，美国、英国、其他欧洲国家，南美洲、印度、中国，其他亚洲国家和澳大利亚，那么你应该如何设立开发集和测试集呢？



其中一种做法是，你可以选择其中 4 个区域，我打算使用这四个（前四个），但也可以是随机选的区域，然后说，来自这四个区域的数据构成开发集。然后其他四个区域，我打算用这四个（后四个），也可以随机选择 4 个，这些数据构成测试集。

事实证明，这个想法非常糟糕，因为这个例子中，你的开发集和测试集来自不同的分布。

我建议你们不要这样，而是让你的开发集和测试集来自同一分布。我的意思是这样，你们要记住，我想就是设立你的开发集加上一个单实数评估指标，这就是像是定下目标，然后告诉你的团队，那就是你要瞄准的靶心，因为你一旦建立了这样的开发集和指标，团队就可以快速迭代，尝试不同的想法，跑实验，可以很快地使用开发集和指标去评估不同分类器，然后尝试选出最好的那个。所以，机器学习团队一般都很擅长使用不同方法去逼近目标，然后不断迭代，不断逼近靶心。所以，针对开发集上的指标优化。

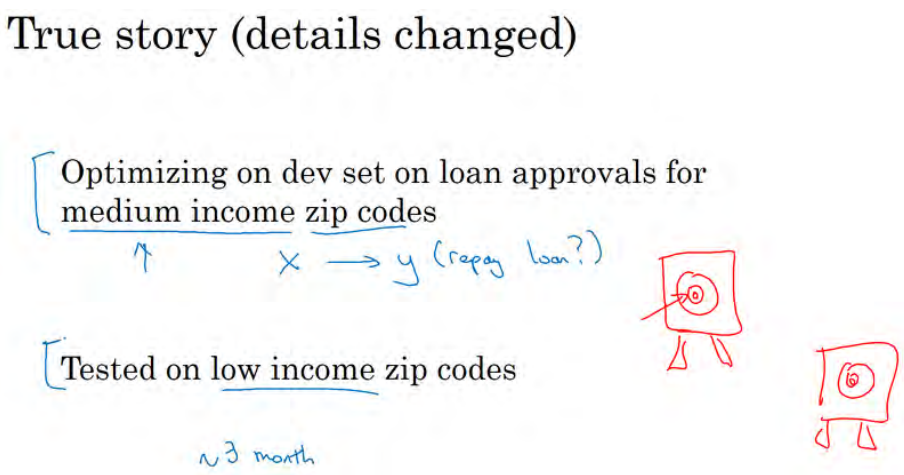


然后在左边的例子中，设立开发集和测试集时存在一个问题，你的团队可能会花上几个月时间在开发集上迭代优化，结果发现，当你们最终在测试集上测试系统时，来自这四个国家或者说下面这四个地区的数据（即测试集数据）和开发集里的数据可能差异很大，所以你可能会收获"意外惊喜"，并发现，花了那么多个月的时间去针对开发集优化，在测试集上的表现却不佳。所以，如果你的开发集和测试集来自不同的分布，就像你设了一个目标，让你的团队花几个月尝试逼近靶心，结果在几个月工作之后发现，你说“等等”，测试的时候，"我要把目标移到这里"，然后团队可能会说"好吧，为什么你让我们花那么多个月的时间去逼近那个靶心，然后突然间你可以把靶心移到不同的位置？"。

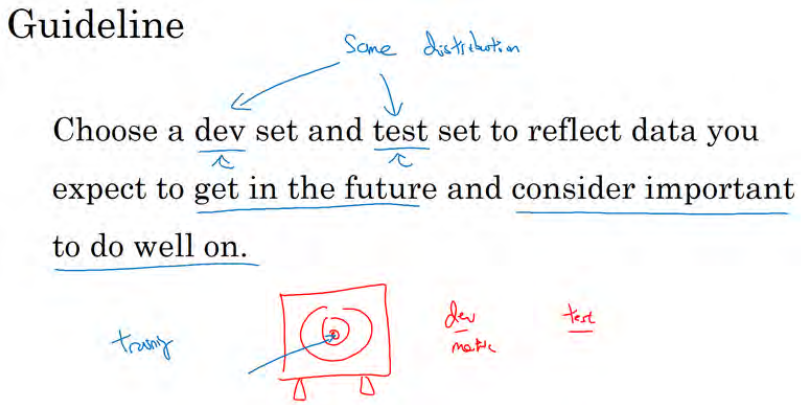
所以，为了避免这种情况，我建议的是你将所有数据随机洗牌，放入开发集和测试集，所以开发集和测试集都有来自八个地区的数据，并且开发集和测试集都来自同一分布，这分布就是你的所有数据混在一起。

这里有另一个例子，这是个真实的故事，但有一些细节变了。所以我知道有一个机器学习团队，花了好几个月在开发集上优化，开发集里面有中等收入邮政编码的贷款审批数据。

那么具体的机器学习问题是，输入x为贷款申请，你是否可以预测输出y，y是他们有没有还贷能力？所以这系统能帮助银行判断是否批准贷款。所以开发集来自贷款申请，这些贷款申请来自中等收入邮政编码，**zip code** 就是美国的邮政编码。但是在这上面训练了几个月之后，团队突然决定要在，低收入邮政编码数据上测试一下。当然了，这个分布数据里面中等收入和低收入邮政编码数据是很不一样的，而且他们花了大量时间针对前面那组数据优化分类器，导致系统在后面那组数据中效果很差。所以这个特定团队实际上浪费了 3 个月的时间，不得不退回去重新做很多工作。



这里实际发生的事情是，这个团队花了三个月瞄准一个目标，三个月之后经理突然问"你们试试瞄准那个目标如何？"，这新目标位置完全不同，所以这件事对于这个团队来说非常崩溃。



所以我建议你们在设立开发集和测试集时，要选择这样的开发集和测试集，能够反映你未来会得到的数据，认为很重要的数据，必须得到好结果的数据，特别是，这里的开发集和测试集可能来自同一个分布。所以不管你未来会得到什么样的数据，一旦你的算法效果不错，要尝试收集类似的数据，而且，不管那些数据是什么，都要随机分配到开发集和测试集上。因为这样，你才能将瞄准想要的目标，让你的团队高效迭代来逼近同一个目标，希望最好是

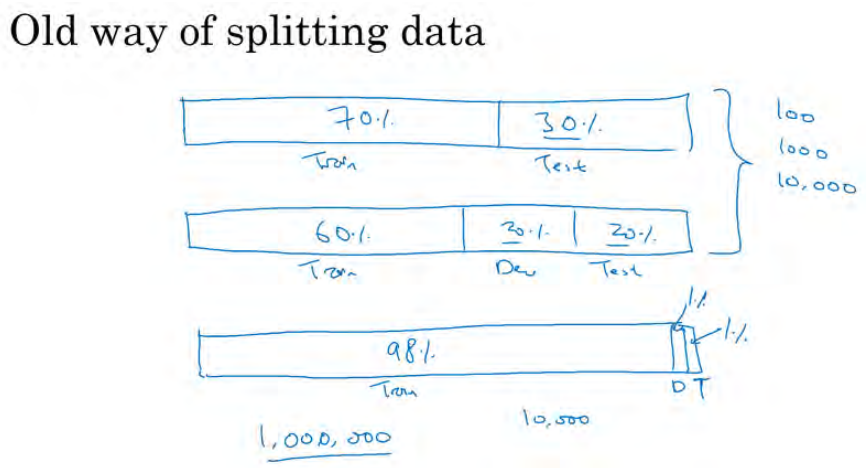
同一个目标。

我们还没提到如何设立训练集，我们会在之后的视频里谈谈如何设立训练集，但这个视频的重点在于，设立开发集以及评估指标，真的就定义了你要瞄准的目标。我们希望通过在同一分布中设立开发集和测试集，你就可以瞄准你所希望的机器学习团队瞄准的目标。而设立训练集的方式则会影响你逼近那个目标有多快，但我们可以在另一个讲座里提到。我知道有一些机器学习团队，他们如果能遵循这个方针，就可以省下几个月的工作，所以我希望这些方针也能帮到你们。

接下来，实际上你的开发集和测试集的规模，如何选择它们的大小，在深度学习时代也在变化，我们会在下一个视频里提到这些内容。

1.6 开发集和测试集的大小（Size of dev and test sets）

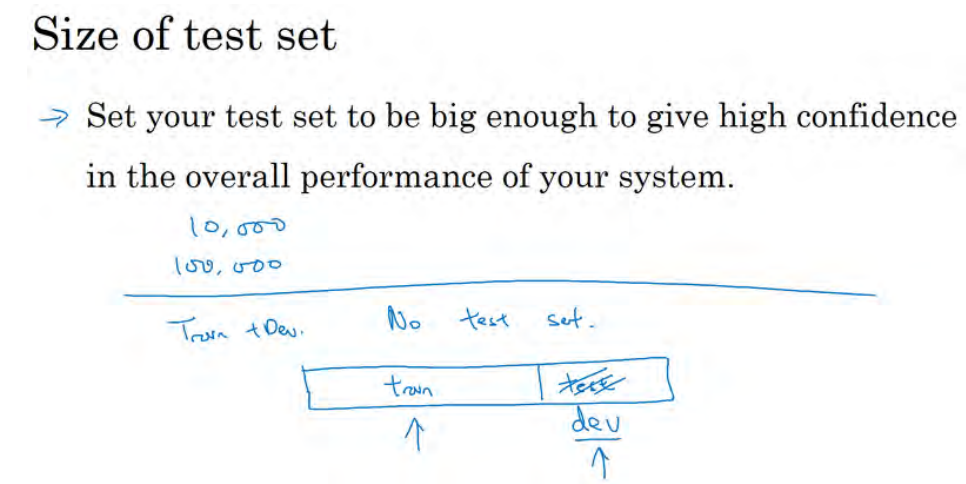
在上一个视频中你们知道了你的开发集和测试集为什么必须来自同一分布，但它们规模应该多大？在深度学习时代，设立开发集和测试集的方针也在变化，我们来看看一些最佳做法。



你可能听说过一条经验法则，在机器学习中，把你取得的全部数据用 70/30 比例分成训练集和测试集。或者如果你必须设立训练集、开发集和测试集，你会这么分 60%训练集，20%开发集，20%测试集。在机器学习的早期，这样分是相当合理的，特别是以前的数据集大小要小得多。所以如果你总共有 100 个样本，这样 70/30 或者 60/20/20 分的经验法则是相当合理的。如果你有几千个样本或者有一万个样本，这些做法也还是合理的。

但在现代机器学习中，我们更习惯操作规模大得多的数据集，比如说你有 1 百万个训练样本，这样分可能更合理，98%作为训练集，1%开发集，1%测试集，我们用D和T缩写来表示开发集和测试集。因为如果你有 1 百万个样本，那么 1%就是 10,000 个样本，这对于开发集和测试集来说可能已经够了。所以在现代深度学习时代，有时我们拥有大得多的数据集，所以使用小于 20%的比例或者小于 30%比例的数据作为开发集和测试集也是合理的。而且因为深度学习算法对数据的胃口很大，我们可以看到那些有海量数据集的问题，有更高比例的数据划分到训练集里，那么测试集呢？

要记住，测试集的目的是完成系统开发之后，测试集可以帮你评估投产系统的性能。方针就是，令你的测试集足够大，能够以高置信度评估系统整体性能。所以除非你需要对最终投产系统有一个很精确的指标，一般来说测试集不需要上百万个例子。对于你的应用程序，也许你想，有 10,000 个例子就能给你足够的置信度来给出性能指标了，也许 100,000 个之类的可能就够了，这数目可能远远小于比如说整体数据集的 30%，取决于你有多少数据。



对于某些应用，你也许不需要对系统性能有置信度很高的评估，也许你只需要训练集和开发集。我认为，不单独分出一个测试集也是可以的。事实上，有时在实践中有些人会只分成训练集和测试集，他们实际上在测试集上迭代，所以这里没有测试集，他们有的是训练集和开发集，但没有测试集。如果你真的在调试这个集，这个开发集或这个测试集，这最好称为开发集。

不过在机器学习的历史里，不是每个人都把术语定义分得很清的，有时人们说的开发集，其实应该看作测试集。但如果你只要有数据去训练，有数据去调试就够了。你打算不管测试集，直接部署最终系统，所以不用太担心它的实际表现，我觉得这也是很好的，就将它们称为训练集、开发集就好。然后说清楚你没有测试集，这是不是有点不正常？我绝对不建议在搭建系统时省略测试集，因为有个单独的测试集比较令我安心。因为你可以使用这组不带偏差的数据来测量系统的性能。但如果你的开发集非常大，这样你就不会对开发集过拟合得太厉害，这种情况，只有训练集和测试集也不是完全不合理的。不过我一般不建议这么做。

总结一下，在大数据时代旧的经验规则，这个 70/30 不再适用了。现在流行的是把大量数据分到训练集，然后少量数据分到开发集和测试集，特别是当你有一个非常大的数据集时。以前的经验法则其实是为了确保开发集足够大，能够达到它的目的，就是帮你评估不同的想法，然后选出A还是B更好。测试集的目的是评估你最终的成本偏差，你只需要设立足够大的测试集，可以用来这么评估就行了，可能只需要远远小于总体数据量的 30%。

所以我希望本视频能给你们一点指导和建议，让你们知道如何在深度学习时代设立开发和测试集。接下来，有时候在研究机器学习的问题途中，你可能需要更改评估指标，或者改动你的开发集和测试集，我们会讲什么时候需要这样做。

1.7 什么时候该改变开发/测试集和指标？（When to change dev/test sets and metrics）

你已经学过如何设置开发集和评估指标，就像是把目标定在某个位置，让你的团队瞄准。但有时候在项目进行途中，你可能意识到，目标的位置放错了。这种情况下，你应该移动你的目标。

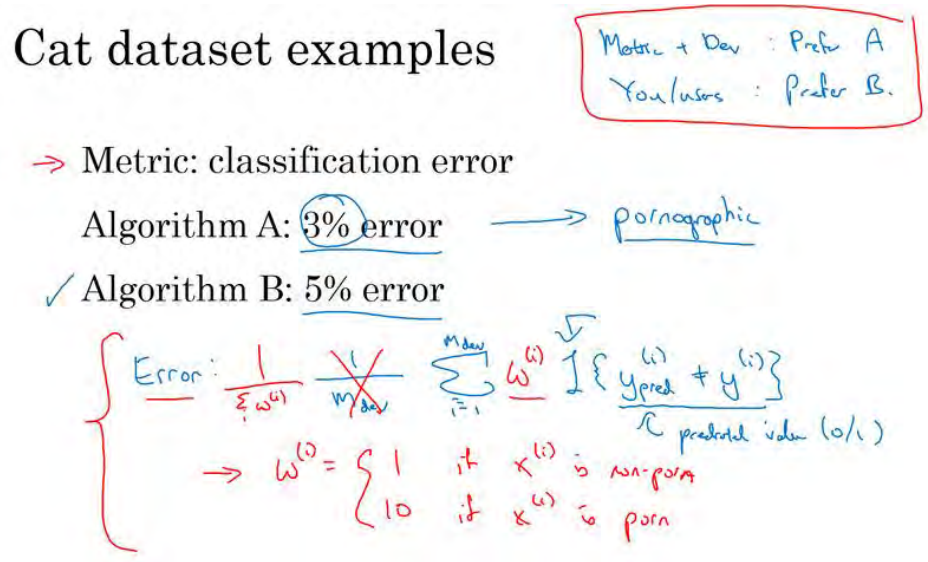
Metric: classification error

Algorithm A: 3% error

Algorithm B: 5% error

我们来看一个例子，假设你在构建一个猫分类器，试图找到很多猫的照片，向你的爱猫人士用户展示，你决定使用的指标是分类错误率。所以算法A和B分别有 3%错误率和 5%错误率，所以算法A似乎做得更好。

但我们实际试一下这些算法，你观察一下这些算法，算法A由于某些原因，把很多色情图像分类成猫了。如果你部署算法A，那么用户就会看到更多猫图，因为它识别猫的错误率只有 3%，但它同时也会给用户推送一些色情图像，这是你的公司完全不能接受的，你的用户也完全不能接受。相比之下，算法B有 5%的错误率，这样分类器就得到较少的图像，但它不会推送色情图像。所以从你们公司的角度来看，以及从用户接受的角度来看，算法B实际上是一个更好的算法，因为它不让任何色情图像通过。

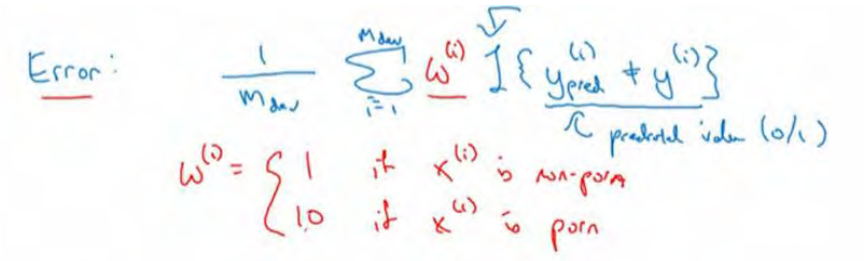


那么在这个例子中，发生的事情就是，算法 A 在评估指标上做得更好，它的错误率达到

3%，但实际上是个更糟糕的算法。在这种情况下，评估指标加上开发集它们都倾向于选择算法A，因为它们会说，看算法 A 的错误率较低，这是你们自己定下来的指标评估出来的。但你和你的用户更倾向于使用算法B，因为它不会将色情图像分类为猫。所以当这种情况发生时，当你的评估指标无法正确衡量算法之间的优劣排序时，在这种情况下，原来的指标错误地预测算法 A 是更好的算法这就发出了信号，你应该改变评估指标了，或者要改变开发集或测试集。在这种情况下，你用的分类错误率指标可以写成这样：

$$Error = \frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} I\{y_{pred}^{(i)} \neq y^{(i)}\}$$

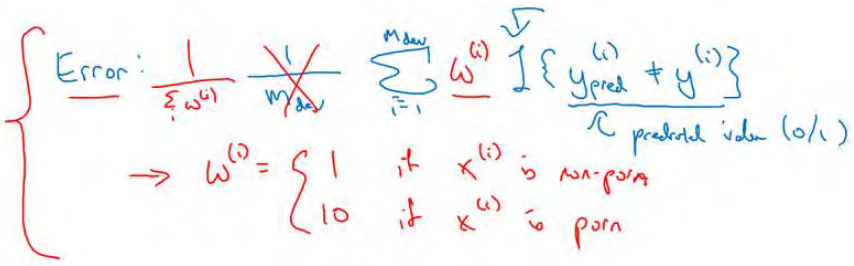
m_{dev}是你的开发集例子数，用$y_{pred}^{(i)}$表示预测值，其值为 0 或 1，I这符号表示一个函数，统计出里面这个表达式为真的样本数，所以这个公式就统计了分类错误的样本。这个评估指标的问题在于，它对色情图片和非色情图片一视同仁，但你其实真的希望你的分类器不会错误标记色情图像。比如说把一张色情图片分类为猫，然后推送给不知情的用户，他们看到色情图片会非常不满。



其中一个修改评估指标的方法是，这里（$\frac{1}{m_{dev}}$与$\sum_{i=1}^{m_{dev}} I\{y_{pred}^{(i)} \neq y^{(i)}\}$之间）加个权重项，即：

$$Error = \frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} w^{(i)} I\{y_{pred}^{(i)} \neq y^{(i)}\}$$

我们将这个称为$w^{(i)}$，其中如果图片$x^{(i)}$不是色情图片，则$w^{(i)} = 1$。如果$x^{(i)}$是色情图片，$w^{(i)}$可能就是 10 甚至 100，这样你赋予了色情图片更大的权重，让算法将色情图分类为猫图时，错误率这个项快速变大。这个例子里，你把色情图片分类成猫这一错误的惩罚权重加大 10 倍。



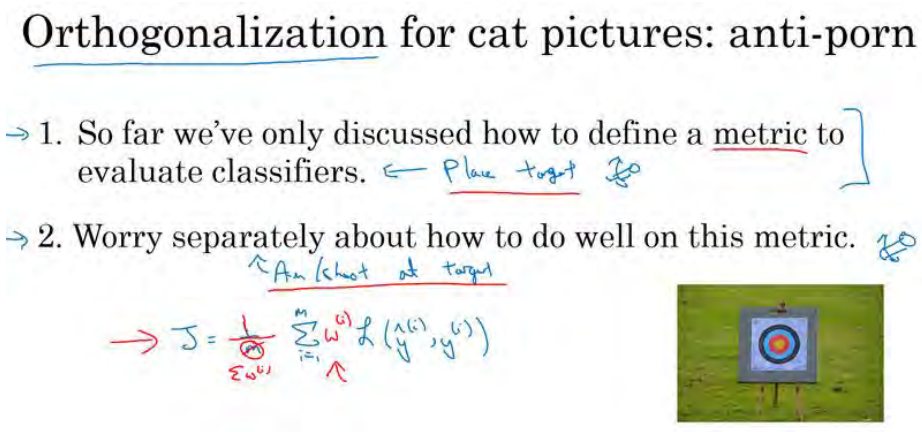
如果你希望得到归一化常数，在技术上，就是$w^{(i)}$对所有i求和，这样错误率仍然在 0 和 1 之间，即：

$$Error = \frac{1}{\sum w^{(i)}} \sum_{i=1}^{m_{dev}} w^{(i)} I\{y_{pred}^{(i)} \neq y^{(i)}\}$$

加权的细节并不重要，实际上要使用这种加权，你必须自己过一遍开发集和测试集，在开发集和测试集里，自己把色情图片标记出来，这样你才能使用这个加权函数。

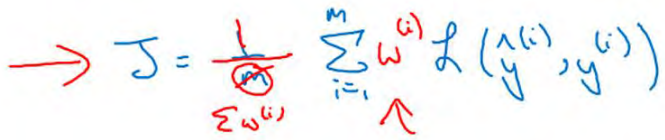
但粗略的结论是，如果你的评估指标无法正确评估好算法的排名，那么就需要花时间定义一个新的评估指标。这是定义评估指标的其中一种可能方式（上述加权法）。评估指标的意义在于，准确告诉你已知两个分类器，哪一个更适合你的应用。就这个视频的内容而言，我们不需要太注重新错误率指标是怎么定义的，关键在于，如果你对旧的错误率指标不满意，那就不要一直沿用你不满意的错误率指标，而应该尝试定义一个新的指标，能够更加符合你的偏好，定义出实际更适合的算法。

你可能注意到了，到目前为止我们只讨论了如何定义一个指标去评估分类器，也就是说，我们定义了一个评估指标帮助我们更好的把分类器排序，能够区分出它们在识别色情图片的不同水平，这实际上是一个正交化的例子。



我想你处理机器学习问题时，应该把它切分成独立的步骤。一步是弄清楚如何定义一个指标来衡量你想做的事情的表现，然后我们可以分开考虑如何改善系统在这个指标上的表现。你们要把机器学习任务看成两个独立的步骤，用目标这个比喻，第一步就是设定目标。

所以要定义你要瞄准的目标，这是完全独立的一步，这是你可以调节的一个旋钮。如何设立目标是一个完全独立的问题，把它看成是一个单独的旋钮，可以调试算法表现的旋钮，如何精确瞄准，如何命中目标，定义指标是第一步。

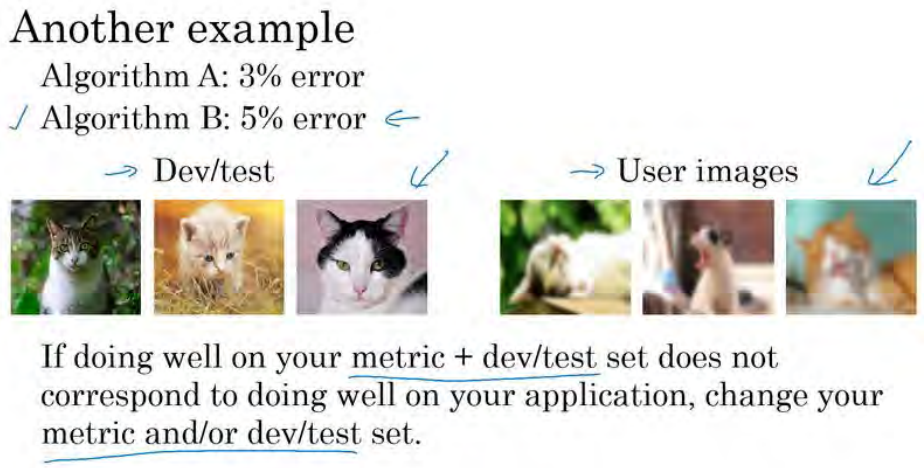


然后第二步要做别的事情，在逼近目标的时候，也许你的学习算法针对某个长这样的成本函数优化，$J=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})$，你要最小化训练集上的损失。你可以做的其中一件事是，修改这个，为了引入这些权重，也许最后需要修改这个归一化常数，即：

$$J=\frac{1}{\sum w^{(i)}}\sum_{i=1}^{m}w^{(i)}L(\hat{y}^{(i)},y^{(i)})$$

再次，如何定义J并不重要，关键在于正交化的思路，把设立目标定为第一步，然后瞄准和射击目标是独立的第二步。换种说法，我鼓励你们将定义指标看成一步，然后在定义了指标之后，你才能想如何优化系统来提高这个指标评分。比如改变你神经网络要优化的成本函数J。

在继续之前，我们再讲一个例子。假设你的两个猫分类器A和B，分别有用开发集评估得到 3%的错误率和 5%的错误率。或者甚至用在网上下载的图片构成的测试集上，这些是高质量，取景框很专业的图像。但也许你在部署算法产品时，你发现算法B看起来表现更好，即使它在开发集上表现不错，你发现你一直在用从网上下载的高质量图片训练，但当你部署到手机应用时，算法作用到用户上传的图片时，那些图片取景不专业，没有把猫完整拍下来，或者猫的表情很古怪，也许图像很模糊，当你实际测试算法时，你发现算法B表现其实更好。



这是另一个指标和开发集测试集出问题的例子，问题在于，你做评估用的是很漂亮的高

分辨率的开发集和测试集，图片取景很专业。但你的用户真正关心的是，他们上传的图片能不能被正确识别。那些图片可能是没那么专业的照片，有点模糊，取景很业余。

所以方针是，如果你在指标上表现很好，在当前开发集或者开发集和测试集分布中表现很好，但你的实际应用程序，你真正关注的地方表现不好，那么就需要修改指标或者你的开发测试集。换句话说，如果你发现你的开发测试集都是这些高质量图像，但在开发测试集上做的评估无法预测你的应用实际的表现。因为你的应用处理的是低质量图像，那么就应该改变你的开发测试集，让你的数据更能反映你实际需要处理好的数据。

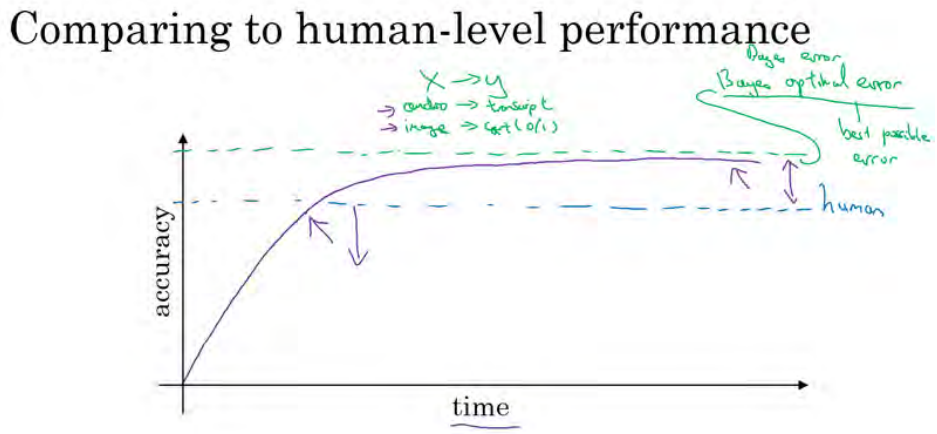
但总体方针就是，如果你当前的指标和当前用来评估的数据和你真正关心必须做好的事情关系不大，那就应该更改你的指标或者你的开发测试集，让它们能更够好地反映你的算法需要处理好的数据。

有一个评估指标和开发集让你可以更快做出决策，判断算法A还是算法B更优，这真的可以加速你和你的团队迭代的速度。所以我的建议是，即使你无法定义出一个很完美的评估指标和开发集，你直接快速设立出来，然后使用它们来驱动你们团队的迭代速度。如果在这之后，你发现选的不好，你有更好的想法，那么完全可以马上改。对于大多数团队，我建议最好不要在没有评估指标和开发集时跑太久，因为那样可能会减慢你的团队迭代和改善算法的速度。本视频讲的是什么时候需要改变你的评估指标和开发测试集，我希望这些方针能让你的整个团队设立一个明确的目标，一个你们可以高效迭代，改善性能的目标。

1.8 为什么是人的表现？（Why human-level performance?）

在过去的几年里，更多的机器学习团队一直在讨论如何比较机器学习系统和人类的表现，为什么呢？

我认为有两个主要原因，首先是因为深度学习系统的进步，机器学习算法突然变得更好了。在许多机器学习的应用领域已经开始见到算法已经可以威胁到人类的表现了。其次，事实证明，当你试图让机器做人类能做的事情时，可以精心设计机器学习系统的工作流程，让工作流程效率更高，所以在这些场合，比较人类和机器是很自然的，或者你要让机器模仿人类的行为。

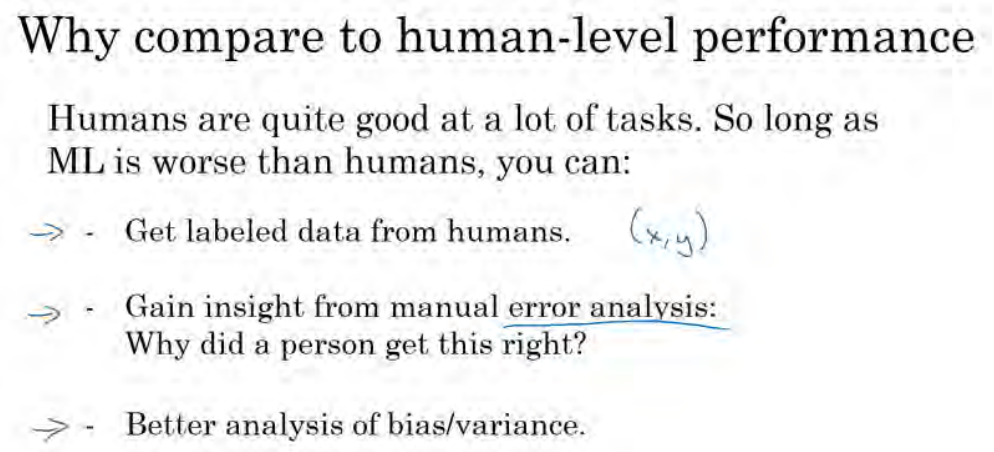


我们来看几个这样的例子，我看到很多机器学习任务中，当你在一个问题上付出了很多时间之后，所以x轴是时间，这可能是很多个月甚至是很多年。在这些时间里，一些团队或一些研究小组正在研究一个问题，当你开始往人类水平努力时，进展是很快的。但是过了一段时间，当这个算法表现比人类更好时，那么进展和精确度的提升就变得更慢了。也许它还会越来越好，但是在超越人类水平之后，它还可以变得更好，但性能增速，准确度上升的速度这个斜率，会变得越来越平缓，我们都希望能达到理论最佳性能水平。随着时间的推移，当您继续训练算法时，可能模型越来越大，数据越来越多，但是性能无法超过某个理论上限，这就是所谓的贝叶斯最优错误率（**Bayes optimal error**）。所以贝叶斯最优错误率一般认为是理论上可能达到的最优错误率，就是说没有任何办法设计出一个x到y的函数，让它能够超过一定的准确度。

例如，对于语音识别来说，如果x是音频片段，有些音频就是这么嘈杂，基本不可能知道说的是什么，所以完美的准确率可能不是 100%。或者对于猫图识别来说，也许一些图像非常模糊，不管是人类还是机器，都无法判断该图片中是否有猫。所以，完美的准确度可能

不是 100%。

而贝叶斯最优错误率有时写作 **Bayesian**，即省略 **optimal**，就是从x到y映射的理论最优函数，永远不会被超越。所以你们应该不会感到意外，这紫色线，无论你在一个问题上工作多少年，你永远不会超越贝叶斯错误率，贝叶斯最佳错误率。



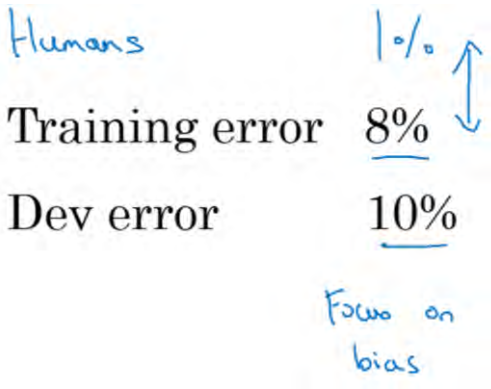
事实证明，机器学习的进展往往相当快，直到你超越人类的表现之前一直很快，当你超越人类的表现时，有时进展会变慢。我认为有两个原因，为什么当你超越人类的表现时，进展会慢下来。一个原因是人类水平在很多任务中离贝叶斯最优错误率已经不远了，人们非常擅长看图像，分辨里面有没有猫或者听写音频。所以，当你超越人类的表现之后也许没有太多的空间继续改善了。但第二个原因是，只要你的表现比人类的表现更差，那么实际上可以使用某些工具来提高性能。一旦你超越了人类的表现，这些工具就没那么好用了。

我的意思是这样，对于人类相当擅长的任务，包括看图识别事物，听写音频，或阅读语言，人类一般很擅长处理这些自然数据。对于人类擅长的任务，只要你的机器学习算法比人类差，你就可以从让人帮你标记数据，你可以让人帮忙或者花钱请人帮你标记例子，这样你就有更多的数据可以喂给学习算法。下周我们会讨论，人工错误率分析，但只要人类的表现比任何其他算法都要好，你就可以让人类看看你算法处理的例子，知道错误出在哪里，并尝试了解为什么人能做对，算法做错。下周我们会看到，这样做有助于提高算法的性能。你也可以更好地分析偏差和方差，我们稍后会谈一谈。但是只要你的算法仍然比人类糟糕，你就有这些重要策略可以改善算法。而一旦你的算法做得比人类好，这三种策略就很难利用了。所以这可能是另一个和人类表现比较的好处，特别是在人类做得很好的任务上。

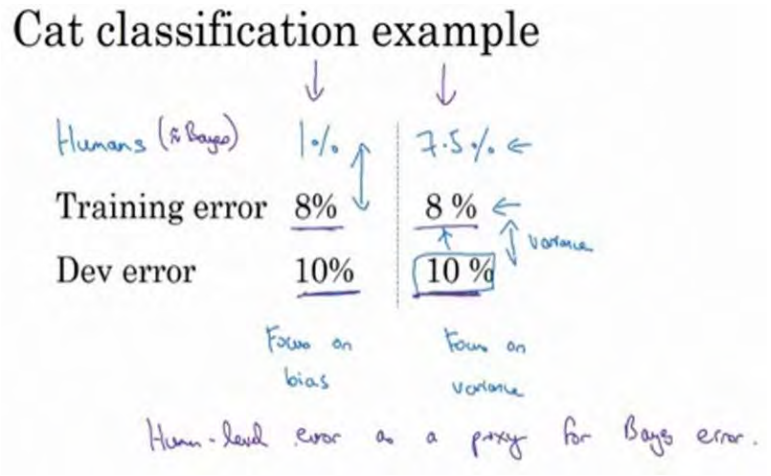
为什么机器学习算法往往很擅长模仿人类能做的事情，然后赶上甚至超越人类的表现。特别是，即使你知道偏差是多少，方差是多少。知道人类在特定任务上能做多好可以帮助你更好地了解你应该重点尝试减少偏差，还是减少方差，我想在下一个视频中给你一个例子。

1.9 可避免偏差（Avoidable bias）

我们讨论过，你希望你的学习算法能在训练集上表现良好，但有时你实际上并不想做得太好。你得知道人类水平的表现是怎样的，可以确切告诉你算法在训练集上的表现到底应该有多好，或者有多不好，让我告诉你是什么意思吧。



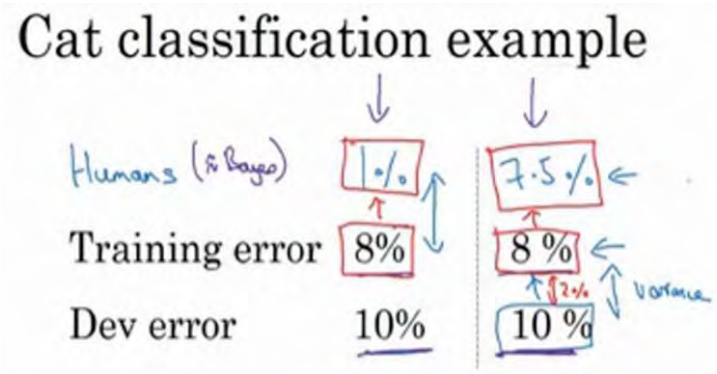
我们经常使用猫分类器来做例子，比如人类具有近乎完美的准确度，所以人类水平的错误是 1%。在这种情况下，如果您的学习算法达到 8%的训练错误率和 10%的开发错误率，那么你也许想在训练集上得到更好的结果。所以事实上，你的算法在训练集上的表现和人类水平的表现有很大差距的话，说明你的算法对训练集的拟合并不好。所以从减少偏差和方差的工具这个角度看，在这种情况下，我会把重点放在减少偏差上。你需要做的是，比如说训练更大的神经网络，或者跑久一点梯度下降，就试试能不能在训练集上做得更好。



但现在我们看看同样的训练错误率和开发错误率，假设人类的表现不是 1%，我们就把它抄写过来。但你知道，在不同的应用或者说用在不同的数据集上，假设人类水平错误实际上是 7.5%，也许你的数据集中的图像非常模糊，即使人类都无法判断这张照片中有没有猫。这个例子可能稍微更复杂一些，因为人类其实很擅长看照片，分辨出照片里有没有猫。但就

为了举这个例子，比如说你的数据集中的图像非常模糊，分辨率很低，即使人类错误率也达到 7.5%。在这种情况下，即使你的训练错误率和开发错误率和其他例子里一样，你就知道，也许你的系统在训练集上的表现还好，它只是比人类的表现差一点点。在第二个例子中，你可能希望专注减少这个分量，减少学习算法的方差，也许你可以试试正则化，让你的开发错误率更接近你的训练错误率。

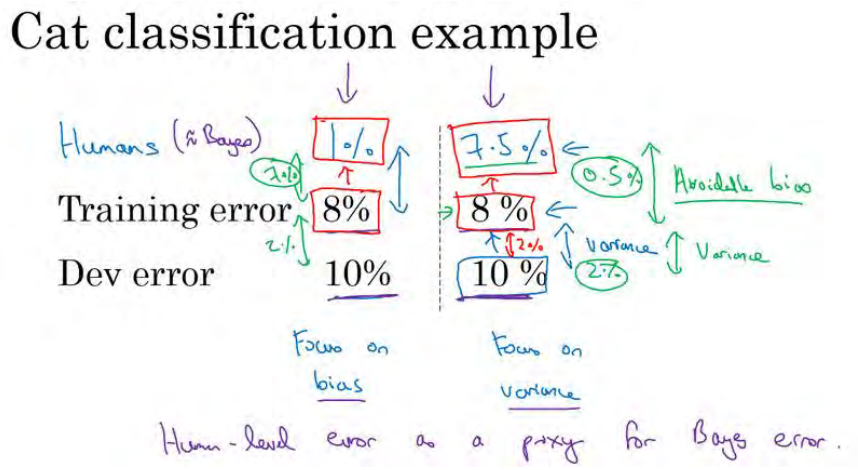
所以在之前的课程关于偏差和方差的讨论中，我们主要假设有一些任务的贝叶斯错误率几乎为 0。所以要解释这里发生的事情，看看这个猫分类器，用人类水平的错误率估计或代替贝叶斯错误率或贝叶斯最优错误率，对于计算机视觉任务而言，这样替代相当合理，因为人类实际上是非常擅长计算机视觉任务的，所以人类能做到的水平和贝叶斯错误率相差不远。根据定义，人类水平错误率比贝叶斯错误率高一点，因为贝叶斯错误率是理论上限，但人类水平错误率离贝叶斯错误率不会太远。所以这里比较意外的是取决于人类水平错误率有多少，或者这真的就很接近贝叶斯错误率，所以我们假设它就是，但取决于我们认为什么样的水平是可以实现的。



在这两种情况下，具有同样的训练错误率和开发错误率，我们决定专注于减少偏差的策略或者减少方差的策略。那么左边的例子发生了什么？ 8%的训练错误率真的很高，你认为你可以把它降到 1%，那么减少偏差的手段可能有效。而在右边的例子中，如果你认为贝叶斯错误率是 7.5%，这里我们使用人类水平错误率来替代贝叶斯错误率，但是你认为贝叶斯错误率接近 7.5%，你就知道没有太多改善的空间了，不能继续减少你的训练错误率了，你也不会希望它比 7.5%好得多，因为这种目标只能通过可能需要提供更进一步的训练。而这边，就还（训练误差和开发误差之间）有更多的改进空间，可以将这个 2%的差距缩小一点，使用减少方差的手段应该可行，比如正则化，或者收集更多的训练数据。

所以要给这些概念命名一下，这不是广泛使用的术语，但我觉得这么说思考起来比较流畅。就是把这个差值，贝叶斯错误率或者对贝叶斯错误率的估计和训练错误率之间的差值称为可避免偏差，你可能希望一直提高训练集表现，直到你接近贝叶斯错误率，但实际上你也

不希望做到比贝叶斯错误率更好，这理论上是不可能超过贝叶斯错误率的，除非过拟合。而这个训练错误率和开发错误率之前的差值，就大概说明你的算法在方差问题上还有多少改善空间。

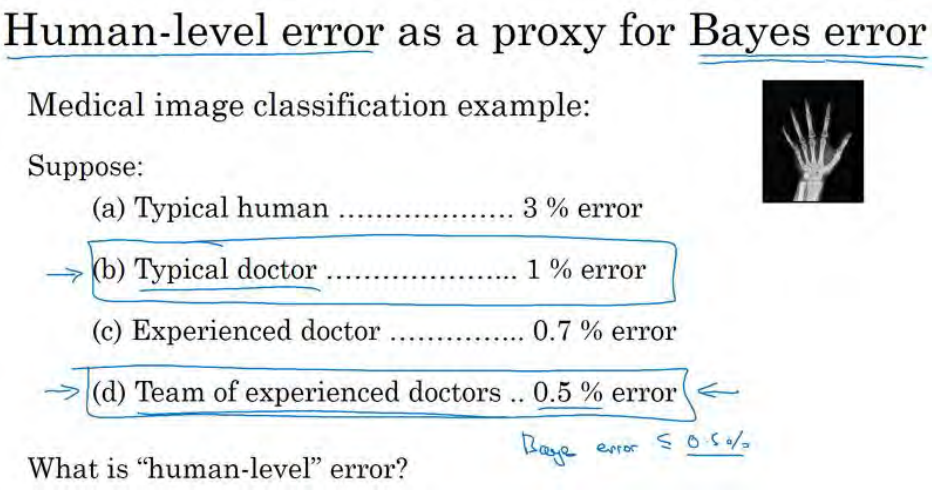


可避免偏差这个词说明了有一些别的偏差，或者错误率有个无法超越的最低水平，那就是说如果贝叶斯错误率是 7.5%。你实际上并不想得到低于该级别的错误率，所以你不会说你的训练错误率是 8%，然后 8%就衡量了例子中的偏差大小。你应该说，可避免偏差可能在 0.5%左右，或者 0.5%是可避免偏差的指标。而这个 2%是方差的指标，所以要减少这个 2%比减少这个 0.5%空间要大得多。而在左边的例子中，这 7%衡量了可避免偏差大小，而 2%衡量了方差大小。所以在左边这个例子里，专注减少可避免偏差可能潜力更大。

所以在这个例子中，当你理解人类水平错误率，理解你对贝叶斯错误率的估计，你就可以在不同的场景中专注于不同的策略，使用避免偏差策略还是避免方差策略。在训练时如何考虑人类水平表现来决定工作着力点，具体怎么做还有更多微妙的细节，所以在下一个视频中，我们会深入了解人类水平表现的真正意义。

1.10 理解人的表现（Understanding human-level performance）

人类水平表现这个词在论文里经常随意使用，但我现在告诉你这个词更准确的定义，特别是使用人类水平表现这个词的定义，可以帮助你们推动机器学习项目的进展。还记得上个视频中，我们用过这个词“人类水平错误率”用来估计贝叶斯误差，那就是理论最低的错误率，任何函数不管是现在还是将来，能够到达的最低值。我们先记住这点，然后看看医学图像分类例子。



假设你要观察这样的放射科图像，然后作出分类诊断，假设一个普通的人类，未经训练的人类，在此任务上达到 3%的错误率。普通的医生，也许是普通的放射科医生，能达到 1%的错误率。经验丰富的医生做得更好，错误率为 0.7%。还有一队经验丰富的医生，就是说如果你有一个经验丰富的医生团队，让他们都看看这个图像，然后讨论并辩论，他们达成共识的意见达到 0.5%的错误率。所以我想问你的问题是，你应该如何界定人类水平错误率？人类水平错误率 3%,1%, 0.7%还是 0.5%？

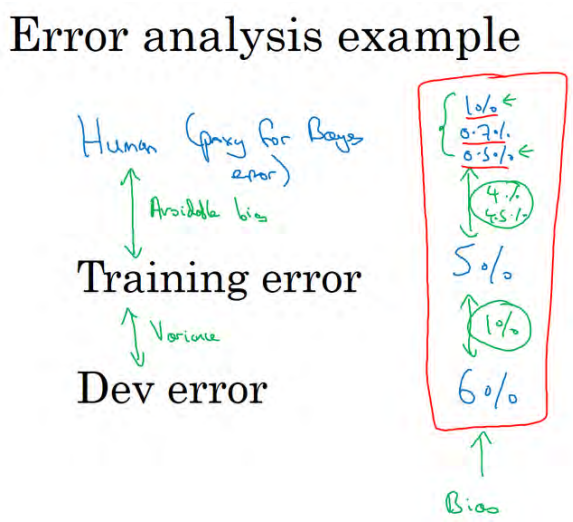
你也可以暂停视频思考一下，要回答这个问题，我想请你记住，思考人类水平错误率最有用的方式之一是，把它作为贝叶斯错误率的替代或估计。如果你愿意，也可以暂停视频，思考一下这个问题。

但这里我就直接给出人类水平错误率的定义，就是如果你想要替代或估计贝叶斯错误率，那么一队经验丰富的医生讨论和辩论之后，可以达到 0.5%的错误率。我们知道贝叶斯错误率小于等于 0.5%，因为有些系统，这些医生团队可以达到 0.5%的错误率。所以根据定义，最优错误率必须在 0.5%以下。我们不知道多少更好，也许有一个更大的团队，更有经验的医生能做得更好，所以也许比 0.5%好一点。但是我们知道最优错误率不能高于 0.5%，那么在这个背景下，我就可以用 0.5%估计贝叶斯错误率。所以我将人类水平定义为 0.5%，至少

如果你希望使用人类水平错误来分析偏差和方差的时候，就像上个视频那样。

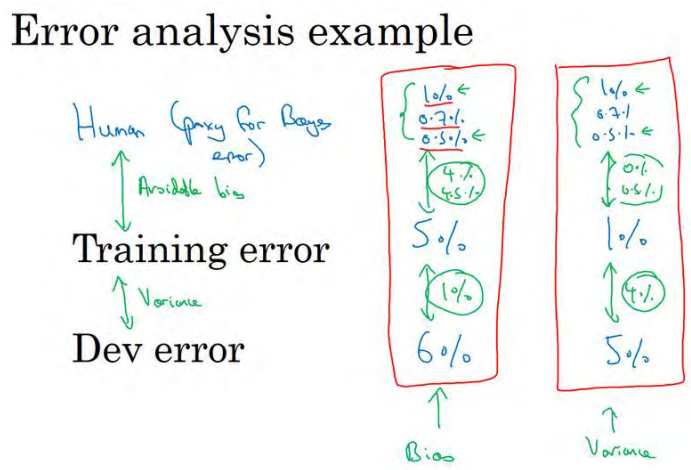
现在，为了发表研究论文或者部署系统，也许人类水平错误率的定义可以不一样，你可以使用 1%，只要你超越了一个普通医生的表现，如果能达到这种水平，那系统已经达到实用了。也许超过一名放射科医生，一名医生的表现，意味着系统在一些情况下可以有部署价值了。

本视频的要点是，在定义人类水平错误率时，要弄清楚你的目标所在，如果要表明你可以超越单个人类，那么就有理由在某些场合部署你的系统，也许这个定义是合适的。但是如果您的目标是替代贝叶斯错误率，那么这个定义（经验丰富的医生团队——0.5%）才合适。



要了解为什么这个很重要，我们来看一个错误率分析的例子。比方说，在医学图像诊断例子中，你的训练错误率是 5%，你的开发错误率是 6%。而在上一张幻灯片的例子中，我们的人类水平表现，我将它看成是贝叶斯错误率的替代品，取决于你是否将它定义成普通单个医生的表现，还是有经验的医生或医生团队的表现，你可能会用 1%或 0.7%或 0.5%。同时也回想一下，前面视频中的定义，贝叶斯错误率或者说贝叶斯错误率的估计和训练错误率直接的差值就衡量了所谓的可避免偏差，这（训练误差与开发误差之间的差值）可以衡量或者估计你的学习算法的方差问题有多严重。

所以在这个第一个例子中，无论你做出哪些选择，可避免偏差大概是 4%，这个值我想介于……，如果你取 1%就是 4%，如果你取 0.5%就是 4.5%，而这个差距（训练误差与开发误差之间的差值）是 1%。所以在这个例子中，我得说，不管你怎么定义人类水平错误率，使用单个普通医生的错误率定义，还是单个经验丰富医生的错误率定义或经验丰富的医生团队的错误率定义，这是 4%还是 4.5%，这明显比都比方差问题更大。所以在这种情况下，你应该专注于减少偏差的技术，例如培训更大的网络。

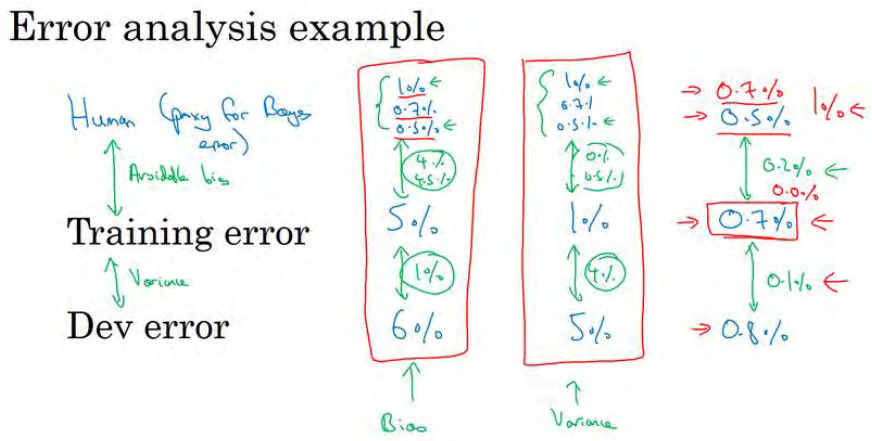


现在来看看第二个例子，比如说你的训练错误率是 1%，开发错误率是 5%，这其实也不怎么重要，这种问题更像学术界讨论的，人类水平表现是 1%或 0.7%还是 0.5%。因为不管你使用哪一个定义，你测量可避免偏差的方法是，如果用那个值，就是 0%到 0.5%之前，对吧？那就是人类水平和训练错误率之前的差距，而这个差距是 4%，所以这个 4%差距比任何一种定义的可避免偏差都大。所以他们就建议，你应该主要使用减少方差的工具，比如正则化或者去获取更大的训练集。

什么时候真正有效呢?

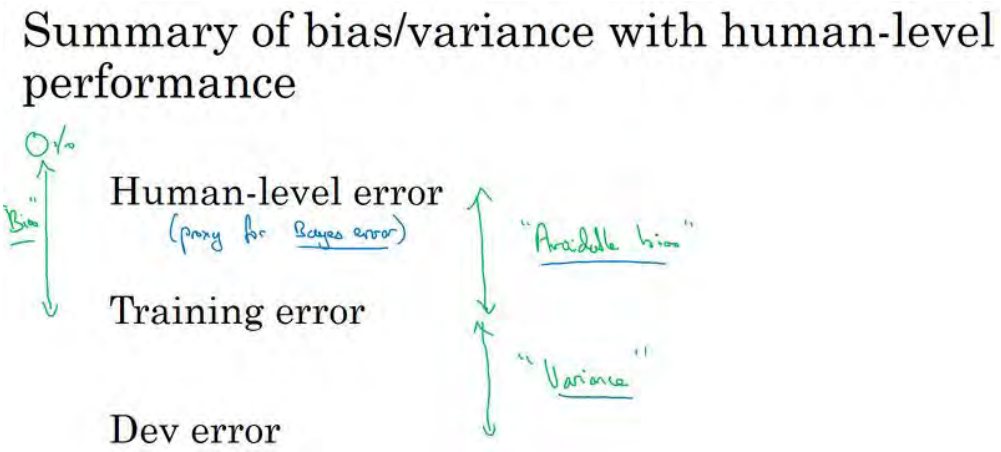
就是比如你的训练错误率是 0.7%，所以你现在已经做得很好了，你的开发错误率是 0.8%。在这种情况下，你用 0.5%来估计贝叶斯错误率关系就很大。因为在这种情况下，你测量到的可避免偏差是 0.2%，这是你测量到的方差问题 0.1%的两倍，这表明也许偏差和方差都存在问题。但是，可避免偏差问题更严重。在这个例子中，我们在上一张幻灯片中讨论的是 0.5%，就是对贝叶斯错误率的最佳估计，因为一群人类医生可以实现这一目标。如果你用 0.7 代替贝叶斯错误率，你测得的可避免偏差基本上是 0%，那你就可能忽略可避免偏差了。实际上你应该试试能不能在训练集上做得更好。

我希望讲这个能让你们有点概念，知道为什么机器学习问题上取得进展会越来越难，当你接近人类水平时进展会越来越难。



在这个例子中，一旦你接近 0.7%错误率，除非你非常小心估计贝叶斯错误率，你可能无法知道离贝叶斯错误率有多远，所以你应该尽量减少可避免偏差。事实上，如果你只知道单个普通医生能达到 1%错误率，这可能很难知道是不是应该继续去拟合训练集，这种问题只会出现在你的算法已经做得很好的时候，只有你已经做到 0.7%, 0.8%, 接近人类水平时会出现。

而在左边的两个例子中，当你远离人类水平时，将优化目标放在偏差或方差上可能更容易一点。这就说明了，为什么当你们接近人类水平时，更难分辨出问题是偏差还是方差。所以机器学习项目的进展在你已经做得很好的时候，很难更进一步。



总结一下我们讲到的，如果你想理解偏差和方差，那么在人类可以做得很好的任务中，你可以估计人类水平的错误率，你可以使用人类水平错误率来估计贝叶斯错误率。所以你到贝叶斯错误率估计值的差距，告诉你可避免偏差问题有多大，可避免偏差问题有多严重，而训练错误率和开发错误率之间的差值告诉你方差上的问题有多大，你的算法是否能够从训练集泛化推广到开发集。

今天讲的和之前课程中见到的重大区别是，以前你们比较的是训练错误率和 0%，直接用这个值估计偏差。相比之下，在这个视频中，我们有一个更微妙的分析，其中并没有假设

你应该得到 0%错误率，因为有时贝叶斯错误率是非零的，有时基本不可能做到比某个错误率阈值更低。所以在之前的课程中，我们测量的是训练错误率，然后观察的是训练错误率比0%高多少，就用这个差值来估计偏差有多大。而事实证明，对于贝叶斯错误率几乎是 0%的问题这样就行了，例如识别猫，人类表现接近完美，所以贝叶斯错误率也接近完美。所以当贝叶斯错误率几乎为零时，可以那么做。但数据噪点很多时，比如背景声音很嘈杂的语言识别，有时几乎不可能听清楚说的是什么，并正确记录下来。对于这样的问题，更好的估计贝叶斯错误率很有必要，可以帮助你更好地估计可避免偏差和方差，这样你就能更好的做出决策，选择减少偏差的策略，还是减少方差的策略。

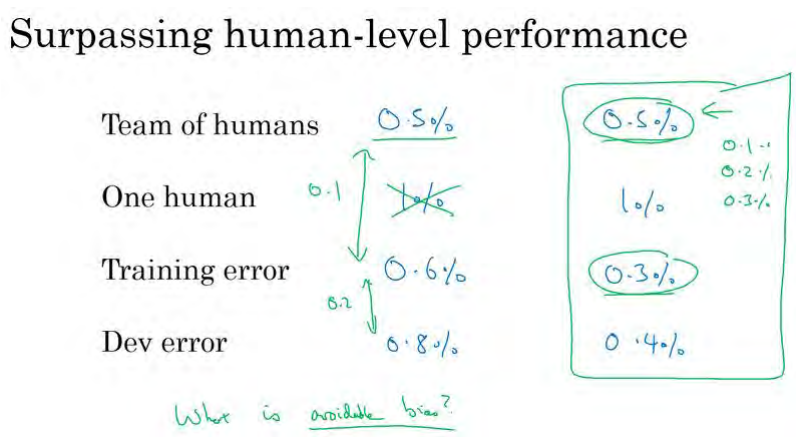
回顾一下，对人类水平有大概的估计可以让你做出对贝叶斯错误率的估计，这样可以让你更快地作出决定是否应该专注于减少算法的偏差，或者减少算法的方差。这个决策技巧通常很有效，直到你的系统性能开始超越人类，那么你对贝叶斯错误率的估计就不再准确了，但这些技巧还是可以帮你做出明确的决定。

现在，深度学习的令人兴奋的发展之一就是对于越来越多的任务，我们的系统实际上可以超越人类了。在下一个视频中，让我们继续谈谈超越人类水平的过程。

1.11 超过人的表现（Surpassing human- level performance）

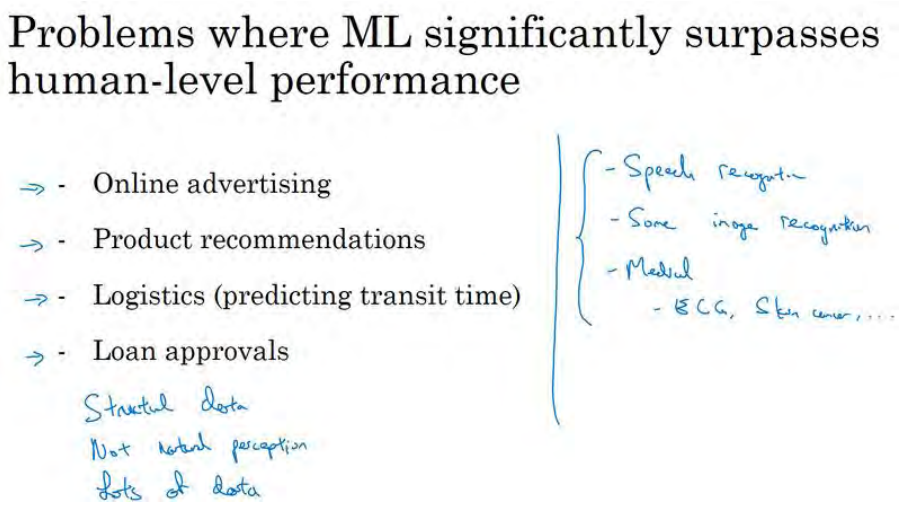
很多团队会因为机器在特定的识别分类任务中超越了人类水平而激动不已，我们谈谈这些情况，看看你们自己能不能达到。

我们讨论过机器学习进展，会在接近或者超越人类水平的时候变得越来越慢。我们举例谈谈为什么会这样。



假设你有一个问题，一组人类专家充分讨论辩论之后，达到 0.5%的错误率，单个人类专家错误率是 1%，然后你训练出来的算法有 0.6%的训练错误率，0.8%的开发错误率。所以在这种情况下，可避免偏差是多少？这个比较容易回答，0.5%是你对贝叶斯错误率的估计，所以可避免偏差就是 0.1%。你不会用这个 1%的数字作为参考，你用的是这个差值，所以也许你对可避免偏差的估计是至少 0.1%，然后方差是 0.2%。和减少可避免偏差比较起来，减少方差可能空间更大。

但现在我们来看一个比较难的例子，一个人类专家团和单个人类专家的表现和以前一样，但你的算法可以得到 0.3%训练错误率，还有 0.4%开发错误率。现在，可避免偏差是什么呢？现在其实很难回答，事实上你的训练错误率是 0.3%，这是否意味着你过拟合了 0.2%，或者说贝叶斯错误率其实是 0.1%呢？或者也许贝叶斯错误率是 0.2%？或者贝叶斯错误率是 0.3%呢？你真的不知道。但是基于本例中给出的信息，你实际上没有足够的信息来判断优化你的算法时应该专注减少偏差还是减少方差，这样你取得进展的效率就会降低。还有比如说，如果你的错误率已经比一群充分讨论辩论后的人类专家更低，那么依靠人类直觉去判断你的算法还能往什么方向优化就很难了。所以在这个例子中，一旦你超过这个 0.5%的门槛，要进一步优化你的机器学习问题就没有明确的选项和前进的方向了。这并不意味着你不能取得进展，你仍然可以取得重大进展。但现有的一些工具帮助你指明方向的工具就没那么好用了。



现在，机器学习有很多问题已经可以大大超越人类水平了。例如，我想网络广告，估计某个用户点击广告的可能性，可能学习算法做到的水平已经超越任何人类了。还有提出产品建议，向你推荐电影或书籍之类的任务。我想今天的网站做到的水平已经超越你最亲近的朋友了。还有物流预测，从A到B开车需要多久，或者预测快递车从A开到B需要多少时间。或者预测某人会不会偿还贷款，这样你就能判断是否批准这人的贷款。我想这些问题都是今天的机器学习远远超过了单个人类的表现。

请注意这四个例子，所有这四个例子都是从结构化数据中学习得来的，这里你可能有个数据库记录用户点击的历史，你的购物历史数据库，或者从A开到B需要多长时间的数据库，以前的贷款申请及结果的数据库，这些并不是自然感知问题，这些不是计算机视觉问题，或语音识别，或自然语言处理任务。人类在自然感知任务中往往表现非常好，所以有可能对计算机来说在自然感知任务的表现要超越人类要更难一些。

最后，这些问题中，机器学习团队都可以访问大量数据，所以比如说，那四个应用中，最好的系统看到的数据量可能比任何人类能看到的都多，所以这样就相对容易得到超越人类水平的系统。现在计算机可以检索那么多数据，它可以比人类更敏锐地识别出数据中的统计规律。

除了这些问题，今天已经有语音识别系统超越人类水平了，还有一些计算机视觉任务，一些图像识别任务，计算机已经超越了人类水平。但是由于人类对这种自然感知任务非常擅长，我想计算机达到那种水平要难得多。还有一些医疗方面的任务，比如阅读 **ECG** 或诊断皮肤癌，或者某些特定领域的放射科读图任务，这些任务计算机做得非常好了，也许超越了单个人类的水平。

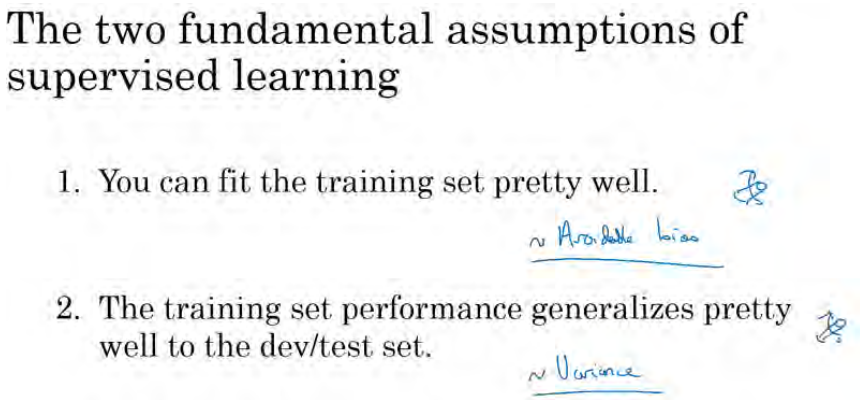
在深度学习的最新进展中，其中一个振奋人心的方面是，即使在自然感知任务中，在某

些情况下，计算机已经可以超越人类的水平了。不过现在肯定更加困难，因为人类一般很擅长这种自然感知任务。

所以要达到超越人类的表现往往不容易，但如果有足够多的数据，已经有很多深度学习系统，在单一监督学习问题上已经超越了人类的水平，所以这对你在开发的应用是有意义的。我希望有一天你也能够搭建出超越人类水平的深度学习系统。

1.12 改善你的模型的表现（Improving your model performance）

你们学过正交化，如何设立开发集和测试集，用人类水平错误率来估计贝叶斯错误率以及如何估计可避免偏差和方差。我们现在把它们全部组合起来写成一套指导方针，如何提高学习算法性能的指导方针。



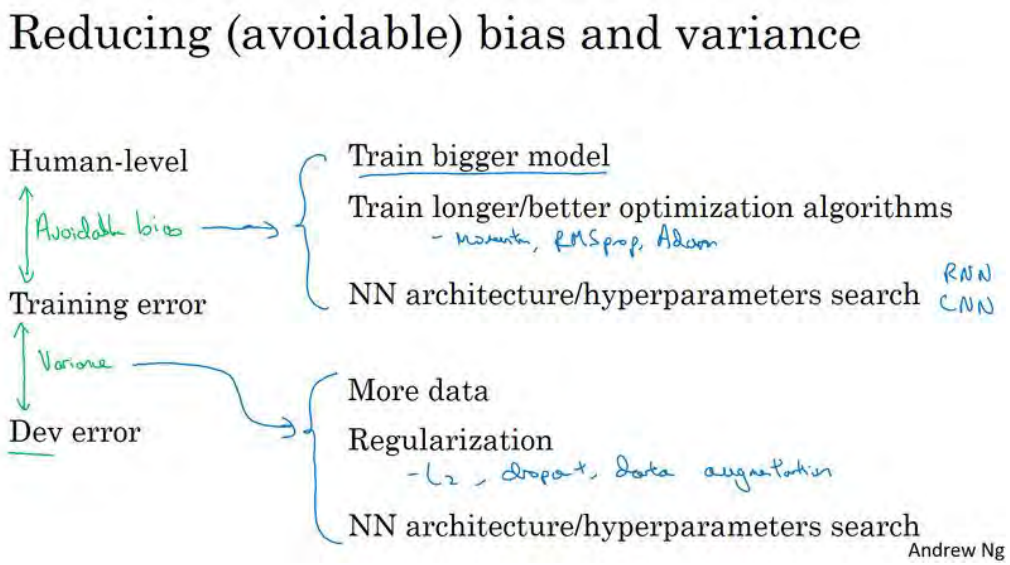
所以我想要让一个监督学习算法达到实用，基本上希望或者假设你可以完成两件事情。首先，你的算法对训练集的拟合很好，这可以看成是你能做到可避免偏差很低。还有第二件事你可以做好的是，在训练集中做得很好，然后推广到开发集和测试集也很好，这就是说方差不是太大。

在正交化的精神下，你可以看到这里有第二组旋钮，可以修正可避免偏差问题，比如训练更大的网络或者训练更久。还有一套独立的技巧可以用来处理方差问题，比如正则化或者收集更多训练数据。

总结一下前几段视频我们见到的步骤，如果你想提升机器学习系统的性能，我建议你们看看训练错误率和贝叶斯错误率估计值之间的距离，让你知道可避免偏差有多大。换句话说，就是你觉得还能做多好，你对训练集的优化还有多少空间。然后看看你的开发错误率和训练错误率之间的距离，就知道你的方差问题有多大。换句话说，你应该做多少努力让你的算法表现能够从训练集推广到开发集，算法是没有在开发集上训练的。

如果你想用尽一切办法减少可避免偏差，我建议试试这样的策略：比如使用规模更大的模型，这样算法在训练集上的表现会更好，或者训练更久。使用更好的优化算法，比如说加入 **momentum** 或者 **RMSprop**，或者使用更好的算法，比如 **Adam**。你还可以试试寻找更好的新神经网络架构，或者说更好的超参数。这些手段包罗万有，你可以改变激活函数，改变层数或者隐藏单位数，虽然你这么做可能会让模型规模变大。或者试用其他模型，其他架构，

如循环神经网络和卷积神经网络。在之后的课程里我们会详细介绍的，新的神经网络架构能否更好地拟合你的训练集，有时也很难预先判断，但有时换架构可能会得到好得多的结果。



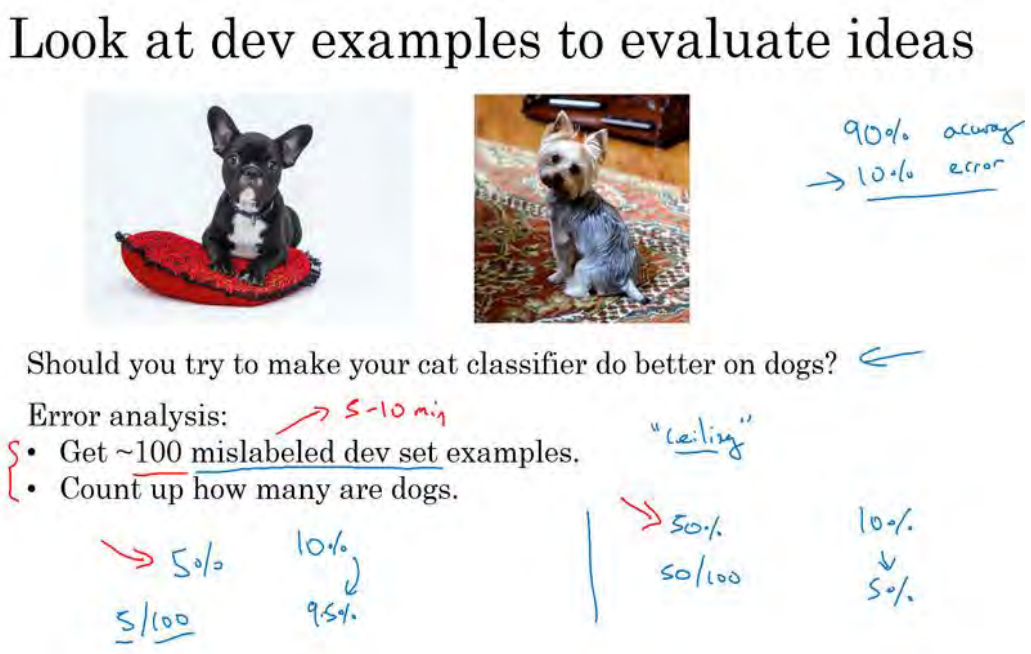
另外当你发现方差是个问题时，你可以试用很多技巧，包括以下这些：你可以收集更多数据，因为收集更多数据去训练可以帮你更好地推广到系统看不到的开发集数据。你可以尝试正则化，包括$L2$正则化，**dropout** 正则化或者我们在之前课程中提到的数据增强。同时你也可以试用不同的神经网络架构，超参数搜索，看看能不能帮助你，找到一个更适合你的问题的神经网络架构。

我想这些偏差、可避免偏差和方差的概念是容易上手，难以精通的。如果你能系统全面地应用本周课程里的概念，你实际上会比很多现有的机器学习团队更有效率、更系统、更有策略地系统提高机器学习系统的性能。

第二周：机器学习策略（2）(ML Strategy (2))

2.1 进行误差分析（Carrying out error analysis）

你好，欢迎回来，如果你希望让学习算法能够胜任人类能做的任务，但你的学习算法还没有达到人类的表现，那么人工检查一下你的算法犯的错误也许可以让你了解接下来应该做什么。这个过程称为错误分析，我们从一个例子开始讲吧。



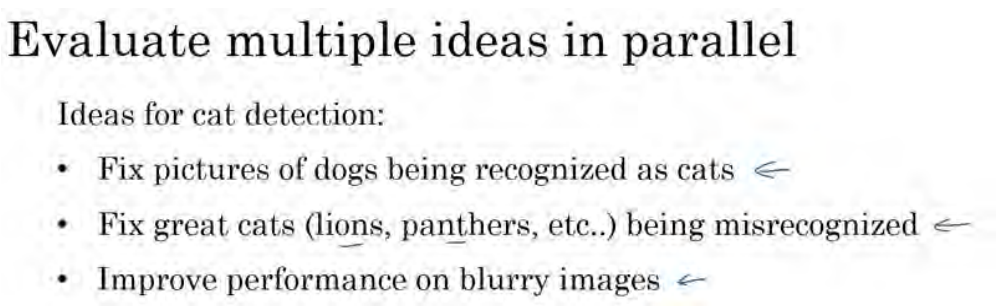
假设你正在调试猫分类器，然后你取得了 90%准确率，相当于 10%错误，，在你的开发集上做到这样，这离你希望的目标还有很远。也许你的队员看了一下算法分类出错的例子，注意到算法将一些狗分类为猫，你看看这两只狗，它们看起来是有点像猫，至少乍一看是。所以也许你的队友给你一个建议，如何针对狗的图片优化算法。试想一下，你可以针对狗，收集更多的狗图，或者设计一些只处理狗的算法功能之类的，为了让你的猫分类器在狗图上做的更好，让算法不再将狗分类成猫。所以问题在于，你是不是应该去开始做一个项目专门处理狗？这项目可能需要花费几个月的时间才能让算法在狗图片上犯更少的错误，这样做值得吗？或者与其花几个月做这个项目，有可能最后发现这样一点用都没有。这里有个错误分析流程，可以让你很快知道这个方向是否值得努力。

这是我建议你做的，首先，收集一下，比如说 100 个错误标记的开发集样本，然后手动检查，一次只看一个，看看你的开发集里有多少错误标记的样本是狗。现在，假设事实上，你的 100 个错误标记样本中只有 5%是狗，就是说在 100 个错误标记的开发集样本中，有 5

个是狗。这意味着 100 个样本，在典型的 100 个出错样本中，即使你完全解决了狗的问题，你也只能修正这 100 个错误中的 5 个。或者换句话说，如果只有 5%的错误是狗图片，那么如果你在狗的问题上花了很多时间，那么你最多只能希望你的错误率从 10%下降到 9.5%，对吧？错误率相对下降了 5%（总体下降了 0.5%，100 的错误样本，错误率为 10%，则样本为 1000），那就是 10%下降到 9.5%。你就可以确定这样花时间不好，或者也许应该花时间，但至少这个分析给出了一个上限。如果你继续处理狗的问题，能够改善算法性能的上限，对吧？在机器学习中，有时我们称之为性能上限，就意味着，最好能到哪里，完全解决狗的问题可以对你有多少帮助。

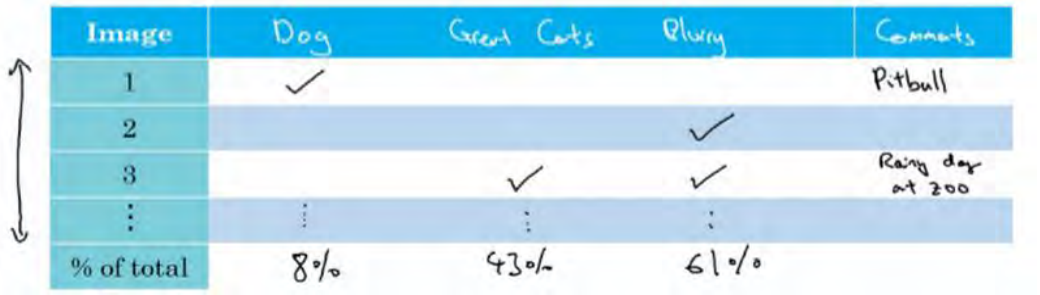
但现在，假设发生了另一件事，假设我们观察一下这 100 个错误标记的开发集样本，你发现实际有 50 张图都是狗，所以有 50%都是狗的照片，现在花时间去解决狗的问题可能效果就很好。这种情况下，如果你真的解决了狗的问题，那么你的错误率可能就从 10%下降到 5%了。然后你可能觉得让错误率减半的方向值得一试，可以集中精力减少错误标记的狗图的问题。

我知道在机器学习中，有时候我们很鄙视手工操作，或者使用了太多人为数值。但如果你要搭建应用系统，那这个简单的人工统计步骤，错误分析，可以节省大量时间，可以迅速决定什么是最重要的，或者最有希望的方向。实际上，如果你观察 100 个错误标记的开发集样本，也许只需要 5 到 10 分钟的时间，亲自看看这 100 个样本，并亲自统计一下有多少是狗。根据结果，看看有没有占到 5%、50%或者其他东西。这个在 5 到 10 分钟之内就能给你估计这个方向有多少价值，并且可以帮助你做出更好的决定，是不是把未来几个月的时间投入到解决错误标记的狗图这个问题。

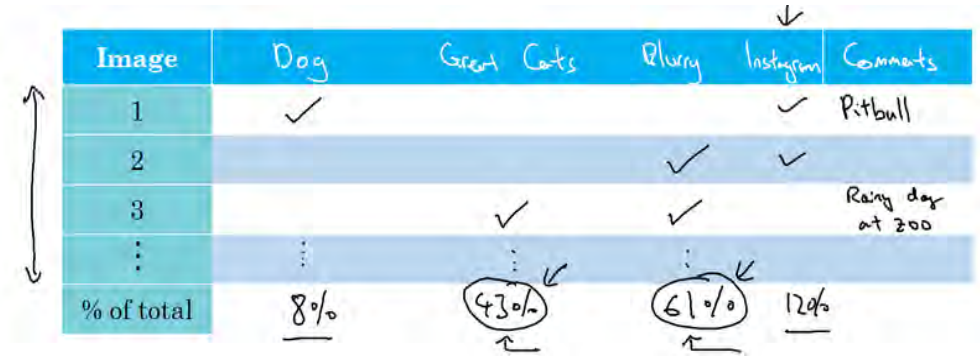


在本幻灯片中，我们要描述一下如何使用错误分析来评估某个想法，这个样本里狗的问题是否值得解决。有时你在做错误分析时，也可以同时并行评估几个想法，比如，你有几个改善猫检测器的想法，也许你可以改善针对狗图的性能，或者有时候要注意，那些猫科动物，如狮子，豹，猎豹等等，它们经常被分类成小猫或者家猫，所以你也许可以想办法解决这个

错误。或者也许你发现有些图像是模糊的，如果你能设计出一些系统，能够更好地处理模糊图像。也许你有些想法，知道大概怎么处理这些问题，要进行错误分析来评估这三个想法。



我会做的是建立这样一个表格，我通常用电子表格来做，但普通文本文件也可以。在最左边，人工过一遍你想分析的图像集，所以图像可能是从 1 到 100，如果你观察 100 张图的话。电子表格的一列就对应你要评估的想法，所以狗的问题，猫科动物的问题，模糊图像的问题，我通常也在电子表格中留下空位来写评论。所以记住，在错误分析过程中，你就看看算法识别错误的开发集样本，如果你发现第一张识别错误的图片是狗图，那么我就在那里打个勾，为了帮我自己记住这些图片，有时我会在评论里注释，也许这是一张比特犬的图。如果第二张照片很模糊，也记一下。如果第三张是在下雨天动物园里的狮子，被识别成猫了，这是大型猫科动物，还有图片模糊，在评论部分写动物园下雨天，是雨天让图像模糊的之类的。最后，这组图像过了一遍之后，我可以统计这些算法(错误)的百分比，或者这里每个错误类型的百分比，有多少是狗，大猫或模糊这些错误类型。所以也许你检查的图像中 8%是狗，可能 43%属于大猫，61%属于模糊。这意味着扫过每一列，并统计那一列有多少百分比图像打了勾。



在这个步骤做到一半时，有时你可能会发现其他错误类型，比如说你可能发现有 **Instagram** 滤镜，那些花哨的图像滤镜，干扰了你的分类器。在这种情况下，实际上可以在错误分析途中，增加这样一列，比如多色滤镜 **Instagram** 滤镜和 **Snapchat** 滤镜，然后再过一遍，也统计一下那些问题，并确定这个新的错误类型占了多少百分比，这个分析步骤的结

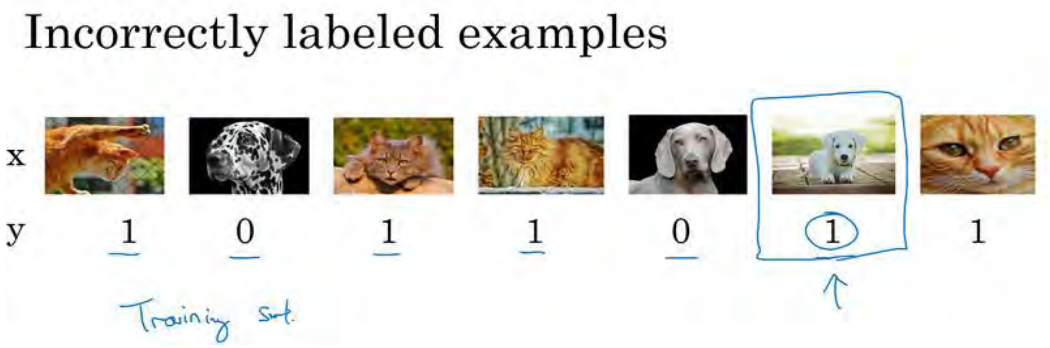
果可以给出一个估计，是否值得去处理每个不同的错误类型。

例如，在这个样本中，有很多错误来自模糊图片，也有很多错误类型是大猫图片。所以这个分析的结果不是说你一定要处理模糊图片，这个分析没有给你一个严格的数学公式，告诉你应该做什么，但它能让你对应该选择那些手段有个概念。它也告诉你，比如说不管你对狗图片或者 **Instagram** 图片处理得有多好，在这些例子中，你最多只能取得 8%或者 12%的性能提升。而在大猫图片这一类型，你可以做得更好。或者模糊图像，这些类型有改进的潜力。这些类型里，性能提高的上限空间要大得多。所以取决于你有多少改善性能的想法，比如改善大猫图片或者模糊图片的表现。也许你可以选择其中两个，或者你的团队成员足够多，也许你把团队可以分成两个团队，其中一个想办法改善大猫的识别，另一个团队想办法改善模糊图片的识别。但这个快速统计的步骤，你可以经常做，最多需要几小时，就可以真正帮你选出高优先级任务，并了解每种手段对性能有多大提升空间。

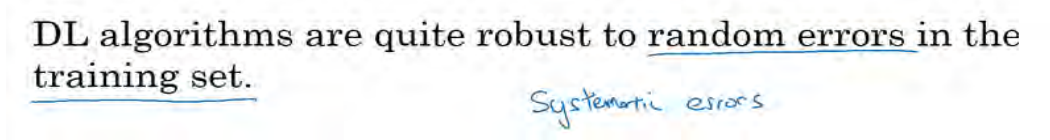
所以总结一下，进行错误分析，你应该找一组错误样本，可能在你的开发集里或者测试集里，观察错误标记的样本，看看假阳性（**false positives**）和假阴性（**false negatives**），统计属于不同错误类型的错误数量。在这个过程中，你可能会得到启发，归纳出新的错误类型，就像我们看到的那样。如果你过了一遍错误样本，然后说，天，有这么多 **Instagram** 滤镜或 **Snapchat** 滤镜，这些滤镜干扰了我的分类器，你就可以在途中新建一个错误类型。总之，通过统计不同错误标记类型占总数的百分比，可以帮你发现哪些问题需要优先解决，或者给你构思新优化方向的灵感。在做错误分析的时候，有时你会注意到开发集里有些样本被错误标记了，这时应该怎么做呢？我们下一个视频来讨论。

2.2 清楚标注错误的数据（Cleaning up Incorrectly labeled data）

你的监督学习问题的数据由输入x和输出标签 y 构成，如果你观察一下你的数据，并发现有些输出标签 y 是错的，这些输出标签 y 是错的，你的数据有些标签是错的，是否值得花时间去修正这些标签呢？



我们看看在猫分类问题中，图片是猫，$y=1$；不是猫，$y=0$。所以假设你看了一些数据样本，发现这（倒数第二张图片）其实不是猫，所以这是标记错误的样本。我用了这个词，“标记错误的样本”来表示你的学习算法输出了错误的 y 值。但我要说的是，对于标记错误的样本，参考你的数据集，在训练集或者测试集 y 的标签，人类给这部分数据加的标签，实际上是错的，这实际上是一只狗，所以 y 其实应该是 0，也许做标记的那人疏忽了。如果你发现你的数据有一些标记错误的样本，你该怎么办？

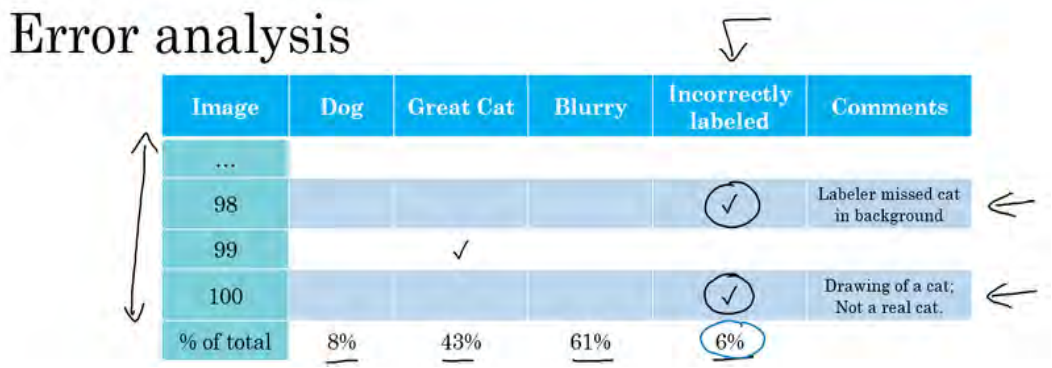


首先，我们来考虑训练集，事实证明，深度学习算法对于训练集中的随机错误是相当健壮的（**robust**）。只要你的标记出错的样本，只要这些错误样本离随机错误不太远，有时可能做标记的人没有注意或者不小心，按错键了，如果错误足够随机，那么放着这些错误不管可能也没问题，而不要花太多时间修复它们。

当然你浏览一下训练集，检查一下这些标签，并修正它们也没什么害处。有时候修正这些错误是有价值的，有时候放着不管也可以，只要总数据集总足够大，实际错误率可能不会太高。我见过一大批机器学习算法训练的时候，明知训练集里有x个错误标签，但最后训练出来也没问题。

我这里先警告一下，深度学习算法对随机误差很健壮，但对系统性的错误就没那么健壮

了。所以比如说，如果做标记的人一直把白色的狗标记成猫，那就成问题了。因为你的分类器学习之后，会把所有白色的狗都分类为猫。但随机错误或近似随机错误，对于大多数深度学习算法来说不成问题。

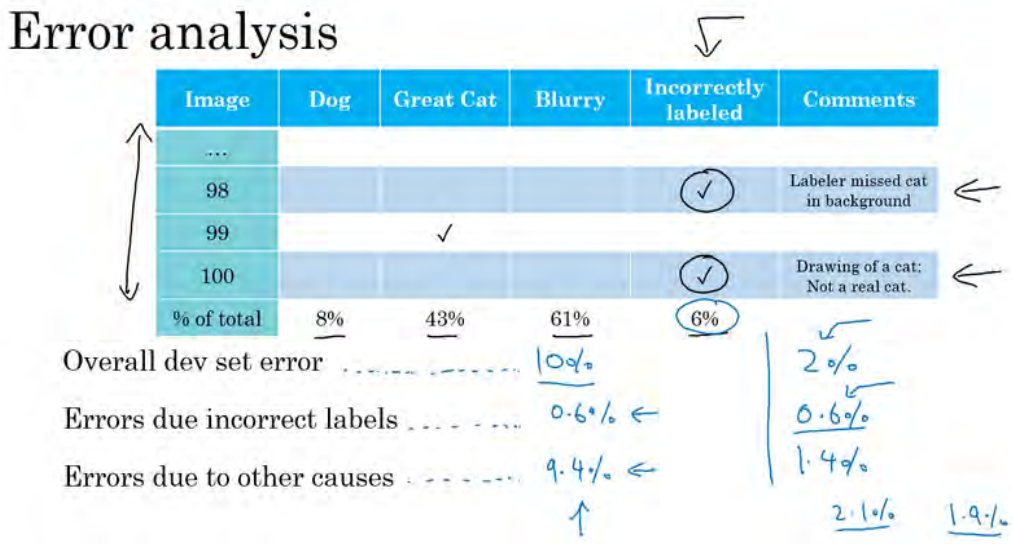


现在，之前的讨论集中在训练集中的标记出错的样本，那么如果是开发集和测试集中有这些标记出错的样本呢？如果你担心开发集或测试集上标记出错的样本带来的影响，他们一般建议你在错误分析时，添加一个额外的列，这样你也可以统计标签 $y = 1$错误的样本数。所以比如说，也许你统计一下对 100 个标记出错的样本的影响，所以你会找到 100 个样本，其中你的分类器的输出和开发集的标签不一致，有时对于其中的少数样本，你的分类器输出和标签不同，是因为标签错了，而不是你的分类器出错。所以也许在这个样本中，你发现标记的人漏了背景里的一只猫，所以那里打个勾，来表示样本 98 标签出错了。也许这张图实际上是猫的画，而不是一只真正的猫，也许你希望标记数据的人将它标记为$y = 0$，而不是$y = 1$，然后再在那里打个勾。当你统计出其他错误类型的百分比后，就像我们在之前的视频中看到的那样，你还可以统计因为标签错误所占的百分比，你的开发集里的 y 值是错的，这就解释了为什么你的学习算法做出和数据集里的标记不一样的预测 1。

所以现在问题是，是否值得修正这 6%标记出错的样本，我的建议是，如果这些标记错误严重影响了你在开发集上评估算法的能力，那么就应该去花时间修正错误的标签。但是，如果它们没有严重影响到你用开发集评估成本偏差的能力，那么可能就不应该花宝贵的时间去处理。

我给你看一个样本，解释清楚我的意思。所以我建议你看 3 个数字来确定是否值得去人工修正标记出错的数据，我建议你看看整体的开发集错误率，在我们以前的视频中的样本，我们说也许我们的系统达到了 90%整体准确度，所以有 10%错误率，那么你应该看看错误标记引起的错误的数量或者百分比。所以在这种情况下，6%的错误来自标记出错，所以 10%的 6%就是 0.6%。也许你应该看看其他原因导致的错误，如果你的开发集上有 10%错误，其中 0.6%是因为标记出错，剩下的占 9.4%，是其他原因导致的，比如把狗误认为猫，大猫图

片。所以在这种情况下，我说有 9.4%错误率需要集中精力修正，而标记出错导致的错误是总体错误的一小部分而已，所以如果你一定要这么做，你也可以手工修正各种错误标签，但也许这不是当下最重要的任务。



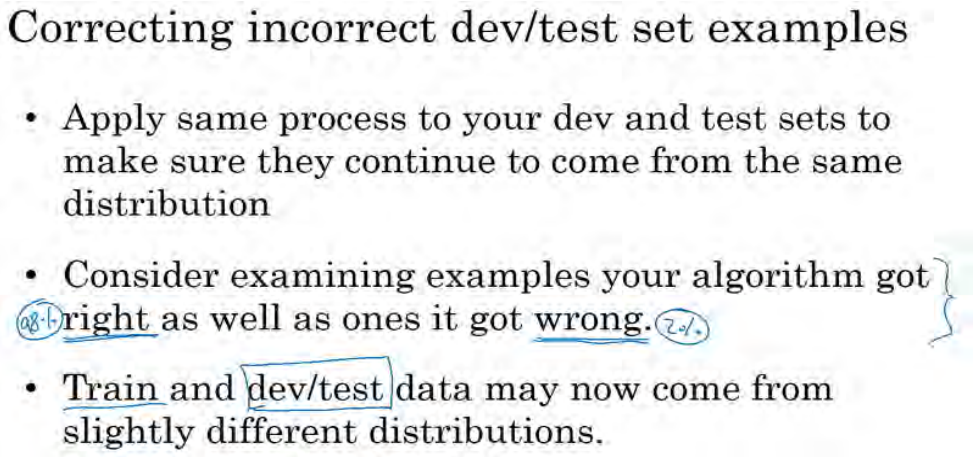
我们再看另一个样本，假设你在学习问题上取得了很大进展，所以现在错误率不再是 10%了，假设你把错误率降到了 2%，但总体错误中的 0.6%还是标记出错导致的。所以现在，如果你想检查一组标记出错的开发集图片，开发集数据有 2%标记错误了，那么其中很大一部分，0.6%除以 2%，实际上变成 30%标签而不是 6%标签了。有那么多错误样本其实是因为标记出错导致的，所以现在其他原因导致的错误是 1.4%。当测得的那么大一部分的错误都是开发集标记出错导致的，那似乎修正开发集里的错误标签似乎更有价值。

Goal of dev set is to help you select between two classifiers A & B.

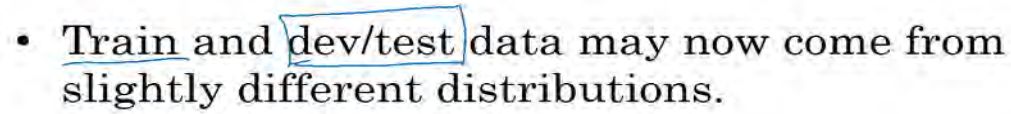
如果你还记得设立开发集的目标的话，开发集的主要目的是，你希望用它来从两个分类器A和B中选择一个。所以当你测试两个分类器A和B时，在开发集上一个有 2.1%错误率，另一个有 1.9%错误率，但是你不能再信任开发集了，因为它无法告诉你这个分类器是否比这个好，因为 0.6%的错误率是标记出错导致的。那么现在你就有很好的理由去修正开发集里的错误标签，因为在右边这个样本中，标记出错对算法错误的整体评估标准有严重的影响。而左边的样本中，标记出错对你算法影响的百分比还是相对较小的。

现在如果你决定要去修正开发集数据，手动重新检查标签，并尝试修正一些标签，这里还有一些额外的方针和原则需要考虑。首先，我鼓励你不管用什么修正手段，都要同时作用到开发集和测试集上，我们之前讨论过为什么，开发和测试集必须来自相同的分布。开发集确定了你的目标，当你击中目标后，你希望算法能够推广到测试集上，这样你的团队能够更

高效的在来自同一分布的开发集和测试集上迭代。如果你打算修正开发集上的部分数据，那么最好也对测试集做同样的修正以确保它们继续来自相同的分布。所以我们雇佣了一个人来仔细检查这些标签，但必须同时检查开发集和测试集。



其次，我强烈建议你要考虑同时检验算法判断正确和判断错误的样本，要检查算法出错的样本很容易，只需要看看那些样本是否需要修正，但还有可能有些样本算法判断正确，那些也需要修正。如果你只修正算法出错的样本，你对算法的偏差估计可能会变大，这会让你的算法有一点不公平的优势，我们就需要再次检查出错的样本，但也需要再次检查做对的样本，因为算法有可能因为运气好把某个东西判断对了。在那个特例里，修正那些标签可能会让算法从判断对变成判断错。这第二点不是很容易做，所以通常不会这么做。通常不会这么做的原因是，如果你的分类器很准确，那么判断错的次数比判断正确的次数要少得多。那么就有 2%出错，98%都是对的，所以更容易检查 2%数据上的标签，然而检查 98%数据上的标签要花的时间长得多，所以通常不这么做，但也是要考虑到的。



最后，如果你进入到一个开发集和测试集去修正这里的部分标签，你可能会，也可能不会去对训练集做同样的事情，还记得我们在其他视频里讲过，修正训练集中的标签其实相对没那么重要，你可能决定只修正开发集和测试集中的标签，因为它们通常比训练集小得多，你可能不想把所有额外的精力投入到修正大得多的训练集中的标签，所以这样其实是可以的。我们将在本周晚些时候讨论一些步骤，用于处理你的训练数据分布和开发与测试数据不同的情况，对于这种情况学习算法其实相当健壮，你的开发集和测试集来自同一分布非常重要。但如果你的训练集来自稍微不同的分布，通常这是一件很合理的事情，我会在本周晚些时候谈谈如何处理这个问题。

最后我讲几个建议：

首先，深度学习研究人员有时会喜欢这样说：“我只是把数据提供给算法，我训练过了，效果拔群”。这话说出了很多深度学习错误的真相，更多时候，我们把数据喂给算法，然后训练它，并减少人工干预，减少使用人类的见解。但我认为，在构造实际系统时，通常需要更多的人工错误分析，更多的人类见解来架构这些系统，尽管深度学习的研究人员不愿意承认这点。

其次，不知道为什么，我看一些工程师和研究人员不愿意亲自去看这些样本，也许做这些事情很无聊，坐下来看 100 或几百个样本来统计错误数量，但我经常亲自这么做。当我带领一个机器学习团队时，我想知道它所犯的错误，我会亲自去看看这些数据，尝试和一部分错误作斗争。我想就因为花了这几分钟，或者几个小时去亲自统计数据，真的可以帮你找到需要优先处理的任务，我发现花时间亲自检查数据非常值得，所以我强烈建议你们这样做，如果你在搭建你的机器学习系统的话，然后你想确定应该优先尝试哪些想法，或者哪些方向。

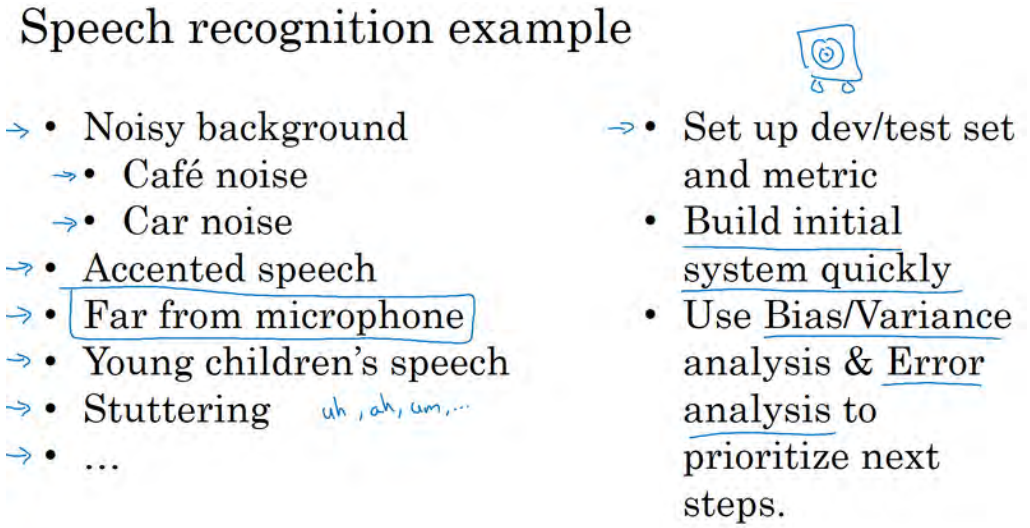
这就是错误分析过程，在下一个视频中，我想分享一下错误分析是如何在启动新的机器学习项目中发挥作用的。

2.3 快速搭建你的第一个系统，并进行迭代（Build your first system quickly, then iterate）

如果你正在开发全新的机器学习应用，我通常会给你这样的建议，你应该尽快建立你的第一个系统原型，然后快速迭代。

让我告诉你我的意思，我在语音识别领域研究了很多年，如果你正在考虑建立一个新的语音识别系统，其实你可以走很多方向，可以优先考虑很多事情。

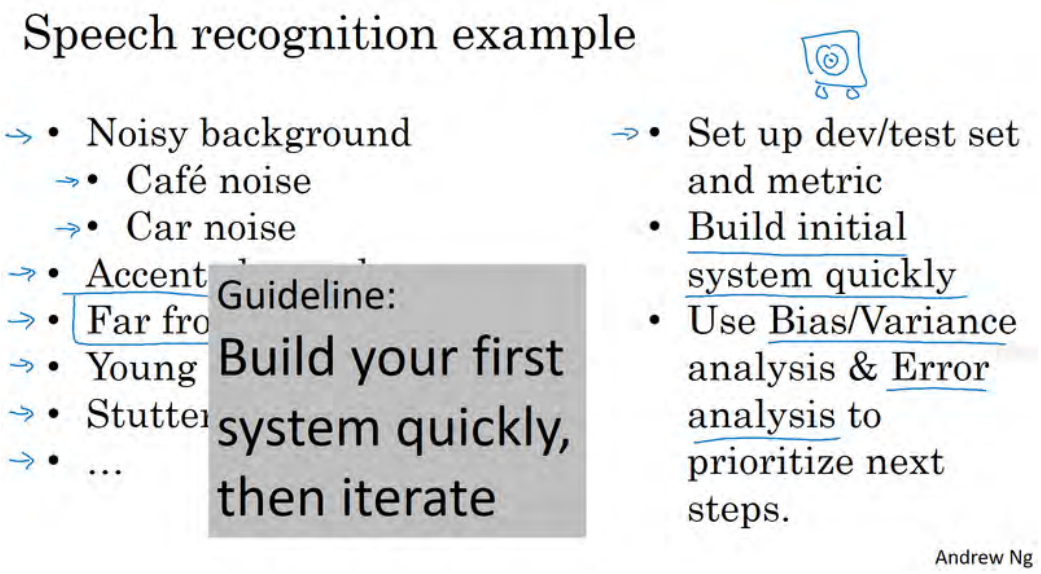
比如，有一些特定的技术，可以让语音识别系统对嘈杂的背景更加健壮，嘈杂的背景可能是说咖啡店的噪音，背景里有很多人在聊天，或者车辆的噪音，高速上汽车的噪音或者其他类型的噪音。有一些方法可以让语音识别系统在处理带口音时更健壮，还有特定的问题和麦克风与说话人距离很远有关，就是所谓的远场语音识别。儿童的语音识别带来特殊的挑战，挑战来自单词发音方面，还有他们选择的词汇，他们倾向于使用的词汇。还有比如说，说话人口吃，或者说了很多无意义的短语，比如“哦”，“啊”之类的。你可以选择很多不同的技术，让你听写下来的文本可读性更强，所以你可以做很多事情来改进语音识别系统。



一般来说，对于几乎所有的机器学习程序可能会有 50 个不同的方向可以前进，并且每个方向都是相对合理的可以改善你的系统。但挑战在于，你如何选择一个方向集中精力处理。即使我已经在语音识别领域工作多年了，如果我要为一个新应用程序域构建新系统，我还是觉得很难不花时间去思考这个问题就直接选择方向。所以我建议你们，如果你想搭建全新的机器学习程序，就是快速搭好你的第一个系统，然后开始迭代。我的意思是我建议你快速设立开发集和测试集还有指标，这样就决定了你的目标所在，如果你的目标定错了，之后改也是可以的。但一定要设立某个目标，然后我建议你马上搭好一个机器学习系统原型，然后找

到训练集，训练一下，看看效果，开始理解你的算法表现如何，在开发集测试集，你的评估指标上表现如何。当你建立第一个系统后，你就可以马上用到之前说的偏差方差分析，还有之前最后几个视频讨论的错误分析，来确定下一步优先做什么。特别是如果错误分析让你了解到大部分的错误的来源是说话人远离麦克风，这对语音识别构成特殊挑战，那么你就有很好的理由去集中精力研究这些技术，所谓远场语音识别的技术，这基本上就是处理说话人离麦克风很远的情况。

建立这个初始系统的所有意义在于，它可以是一个快速和粗糙的实现（**quick and dirty implementation**），你知道的，别想太多。初始系统的全部意义在于，有一个学习过的系统，有一个训练过的系统，让你确定偏差方差的范围，就可以知道下一步应该优先做什么，让你能够进行错误分析，可以观察一些错误，然后想出所有能走的方向，哪些是实际上最有希望的方向。



所以回顾一下，我建议你们快速建立你的第一个系统，然后迭代。不过如果你在这个应用程序领域有很多经验，这个建议适用程度要低一些。还有一种情况适应程度更低，当这个领域有很多可以借鉴的学术文献，处理的问题和你要解决的几乎完全相同，所以，比如说，人脸识别就有很多学术文献，如果你尝试搭建一个人脸识别设备，那么可以从现有大量学术文献为基础出发，一开始就搭建比较复杂的系统。但如果你第一次处理某个新问题，那我真的不鼓励你想太多，或者把第一个系统弄得太复杂。我建议你们构建一些快速而粗糙的实现，然后用来帮你找到改善系统要优先处理的方向。我见过很多机器学习项目，我觉得有些团队的解决方案想太多了，他们造出了过于复杂的系统。我也见过有限团队想的不够，然后造出过于简单的系统。平均来说，我见到更多的团队想太多，构建太复杂的系统。

所以我希望这些策略有帮助，如果你将机器学习算法应用到新的应用程序里，你的主要目标是弄出能用的系统，你的主要目标并不是发明全新的机器学习算法，这是完全不同的目标，那时你的目标应该是想出某种效果非常好的算法。所以我鼓励你们搭建快速而粗糙的实现，然后用它做偏差/方差分析，用它做错误分析，然后用分析结果确定下一步优先要做的方向。

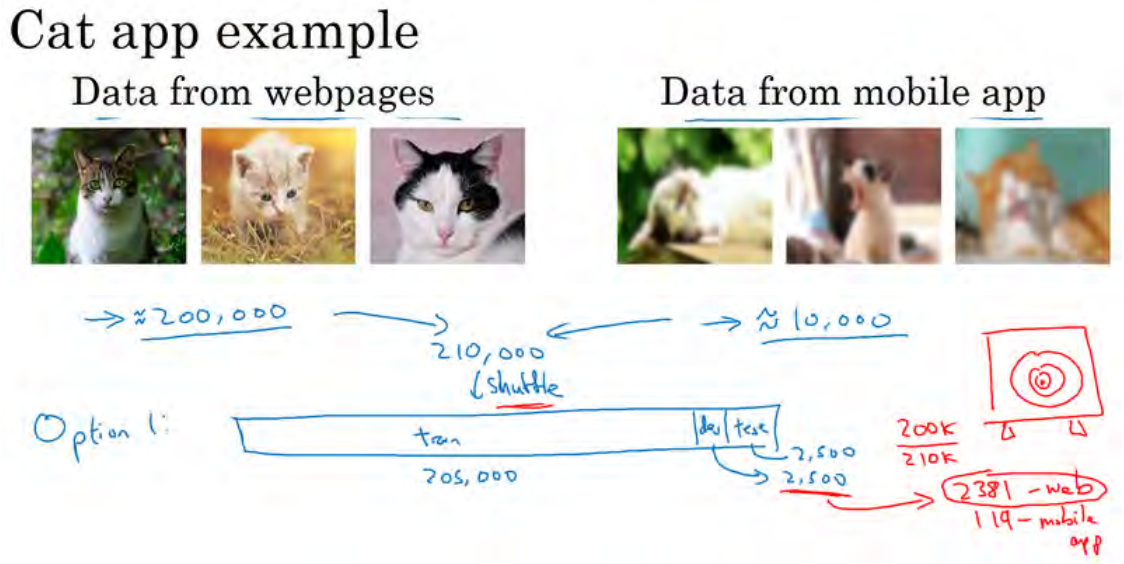
2.4 在不同的划分上进行训练并测试（Training and testing on different distributions）

深度学习算法对训练数据的胃口很大，当你收集到足够多带标签的数据构成训练集时，算法效果最好，这导致很多团队用尽一切办法收集数据，然后把它们堆到训练集里，让训练的数据量更大，即使有些数据，甚至是大部分数据都来自和开发集、测试集不同的分布。在深度学习时代，越来越多的团队都用来自和开发集、测试集分布不同的数据来训练，这里有一些微妙的地方，一些最佳做法来处理训练集和测试集存在差异的情况，我们来看看。

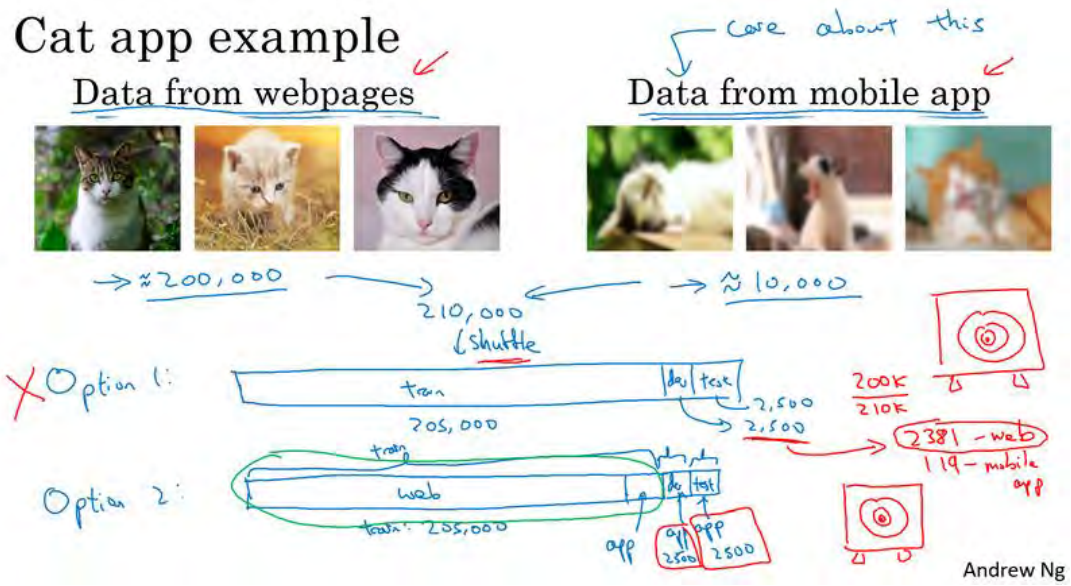
假设你在开发一个手机应用，用户会上传他们用手机拍摄的照片，你想识别用户从应用中上传的图片是不是猫。现在你有两个数据来源，一个是你真正关心的数据分布，来自应用上传的数据，比如右边的应用，这些照片一般更业余，取景不太好，有些甚至很模糊，因为它们都是业余用户拍的。另一个数据来源就是你可以用爬虫程序挖掘网页直接下载，就这个样本而言，可以下载很多取景专业、高分辨率、拍摄专业的猫图片。如果你的应用用户数还不多，也许你只收集到 10,000 张用户上传的照片，但通过爬虫挖掘网页，你可以下载到海量猫图，也许你从互联网上下载了超过 20 万张猫图。而你真正关心的算法表现是你的最终系统处理来自应用程序的这个图片分布时效果好不好，因为最后你的用户会上传类似右边这些图片，你的分类器必须在这个任务中表现良好。现在你就陷入困境了，因为你有一个相对小的数据集，只有 10,000 个样本来自那个分布，而你还有一个大得多的数据集来自另一个分布，图片的外观和你真正想要处理的并不一样。但你又不想直接用这 10,000 张图片，因为这样你的训练集就太小了，使用这 20 万张图片似乎有帮助。但是，困境在于，这 20 万张图片并不完全来自你想要的分布，那么你可以怎么做呢？

这里有一种选择，你可以做的一件事是将两组数据合并在一起，这样你就有 21 万张照片，你可以把这 21 万张照片随机分配到训练、开发和测试集中。为了说明观点，我们假设你已经确定开发集和测试集各包含 2500 个样本，所以你的训练集有 205000 个样本。现在这么设立你的数据集有一些好处，也有坏处。好处在于，你的训练集、开发集和测试集都来

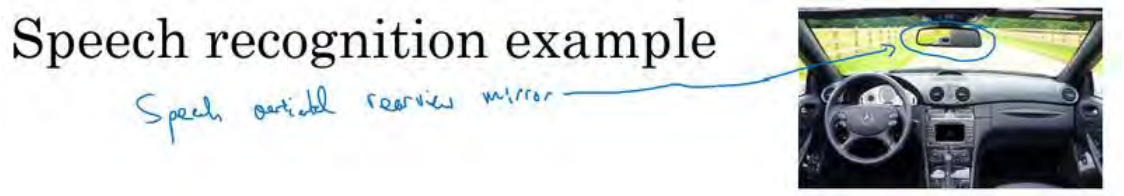
自同一分布，这样更好管理。但坏处在于，这坏处还不小，就是如果你观察开发集，看看这 2500 个样本其中很多图片都来自网页下载的图片，那并不是你真正关心的数据分布，你真正要处理的是来自手机的图片。



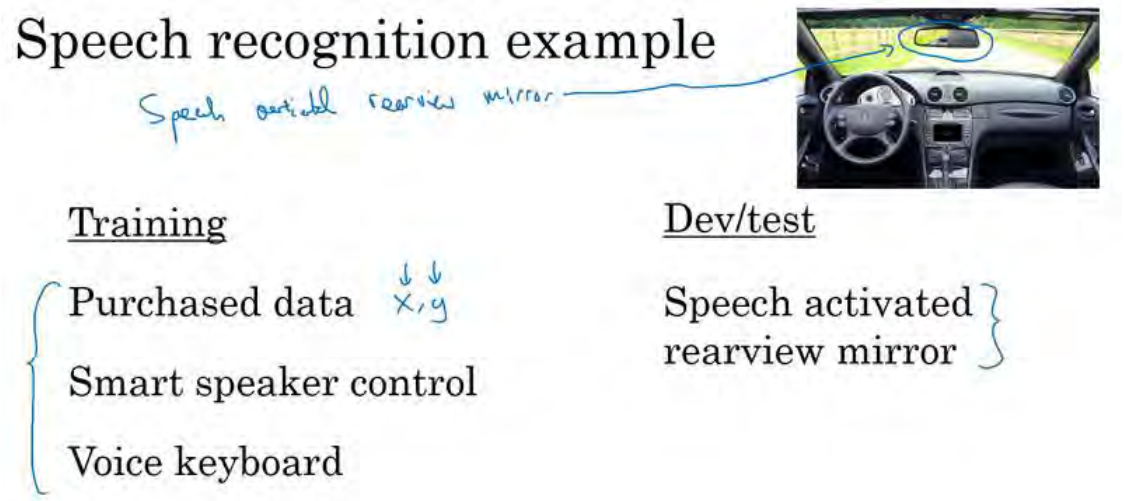
所以结果你的数据总量，这 200,000 个样本，我就用$200k$缩写表示，我把那些是从网页下载的数据总量写成$210k$，所以对于这 2500 个样本，数学期望值是：$2500 \times \frac{200k}{210k} = 2381$，有 2381 张图来自网页下载，这是期望值，确切数目会变化，取决于具体的随机分配操作。但平均而言，只有 119 张图来自手机上传。要记住，设立开发集的目的是告诉你的团队去瞄准的目标，而你瞄准目标的方式，你的大部分精力都用在优化来自网页下载的图片，这其实不是你想要的。所以我真的不建议使用第一个选项，因为这样设立开发集就是告诉你的团队，针对不同于你实际关心的数据分布去优化，所以不要这么做。



我建议你走另外一条路，就是这样，训练集，比如说还是 205,000 张图片，我们的训练集是来自网页下载的 200,000 张图片，然后如果需要的话，再加上 5000 张来自手机上传的图片。然后对于开发集和测试集，这数据集的大小是按比例画的，你的开发集和测试集都是手机图。而训练集包含了来自网页的 20 万张图片，还有 5000 张来自应用的图片，开发集就是 2500 张来自应用的图片，测试集也是 2500 张来自应用的图片。这样将数据分成训练集、开发集和测试集的好处在于，现在你瞄准的目标就是你想要处理的目标，你告诉你的团队，我的开发集包含的数据全部来自手机上传，这是你真正关心的图片分布。我们试试搭建一个学习系统，让系统在处理手机上传图片分布时效果良好。缺点在于，当然了，现在你的训练集分布和你的开发集、测试集分布并不一样。但事实证明，这样把数据分成训练、开发和测试集，在长期能给你带来更好的系统性能。我们以后会讨论一些特殊的技巧，可以处理 训练集的分布和开发集和测试集分布不一样的情况。



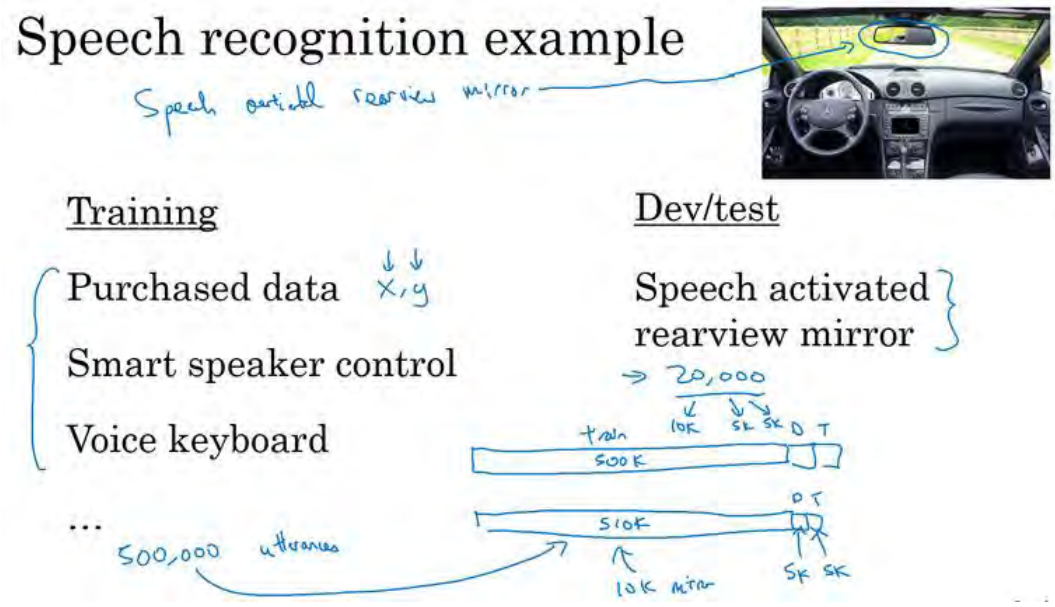
我们来看另一个样本，假设你正在开发一个全新的产品，一个语音激活汽车后视镜，这在中国是个真实存在的产品，它正在进入其他国家。但这就是造一个后视镜，把这个小东西换掉，现在你就可以和后视镜对话了，然后只需要说：“亲爱的后视镜，请帮我找找到最近的加油站的导航方向”，然后后视镜就会处理这个请求。所以这实际上是一个真正的产品，假设现在你要为你自己的国家研制这个产品，那么你怎么收集数据去训练这个产品语言识别模块呢？



也许你已经在语音识别领域上工作了很久，所以你有很多来自其他语音识别应用的数据，

它们并不是来自语音激活后视镜的数据。现在我讲讲如何分配训练集、开发集和测试集。对于你的训练集，你可以将你拥有的所有语音数据，从其他语音识别问题收集来的数据，比如这些年你从各种语音识别数据供应商买来的数据，今天你可以直接买到成x,y对的数据，其中x是音频剪辑，y是听写记录。或者也许你研究过智能音箱，语音激活音箱，所以你有一些数据，也许你做过语音激活键盘的开发之类的。

举例来说，也许你从这些来源收集了 500,000 段录音，对于你的开发集和测试集也许数据集小得多，比如实际上来自语音激活后视镜的数据。因为用户要查询导航信息或试图找到通往各个地方的路线，这个数据集可能会有很多街道地址，对吧？“请帮我导航到这个街道地址”，或者说：“请帮助我导航到这个加油站”，所以这个数据的分布和左边大不一样，但这真的是你关心的数据，因为这些数据是你的产品必须处理好的，所以你就应该把它设成你的开发和测试集。



在这个样本中，你应该这样设立你的训练集，左边有 500,000 段语音，然后你的开发集和测试集，我把它简写成D和T，可能每个集包含 10,000 段语音，是从实际的语音激活后视镜收集的。或者换种方式，如果你觉得不需要将 20,000 段来自语音激活后视镜的录音全部放进开发和测试集，也许你可以拿一半，把它放在训练集里，那么训练集可能是 51 万段语音，包括来自那里的 50 万段语音，还有来自后视镜的 1 万段语音，然后开发集和测试集也许各自有 5000 段语音。所以有 2 万段语音，也许 1 万段语音放入了训练集，5000 放入开发集，5000 放入测试集。所以这是另一种将你的数据分成训练、开发和测试的方式。这样你的训练集大得多，大概有 50 万段语音，比只用语音激活后视镜数据作为训练集要大得多。

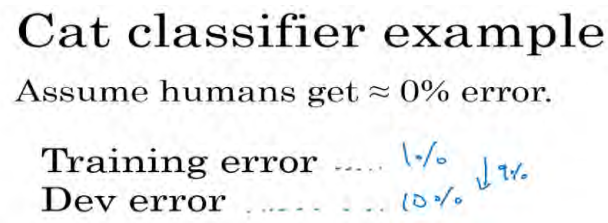
所以在这个视频中，你们见到几组样本，让你的训练集数据来自和开发集、测试集不同

的分布，这样你就可以有更多的训练数据。在这些样本中，这将改善你的学习算法。

现在你可能会问，是不是应该把收集到的数据都用掉？答案很微妙，不一定都是肯定的答案，我们在下段视频看看一个反例。

2.5 不匹配数据划分的偏差和方差（Bias and Variance with mismatched data distributions）

估计学习算法的偏差和方差真的可以帮你确定接下来应该优先做的方向，但是，当你的训练集来自和开发集、测试集不同分布时，分析偏差和方差的方式可能不一样，我们来看为什么。



我们继续用猫分类器为例，我们说人类在这个任务上能做到几乎完美，所以贝叶斯错误率或者说贝叶斯最优错误率，我们知道这个问题里几乎是 0%。所以要进行错误率分析，你通常需要看训练误差，也要看看开发集的误差。比如说，在这个样本中，你的训练集误差是 1%，你的开发集误差是 10%，如果你的开发集来自和训练集一样的分布，你可能会说，这里存在很大的方差问题，你的算法不能很好的从训练集出发泛化，它处理训练集很好，但处理开发集就突然间效果很差了。

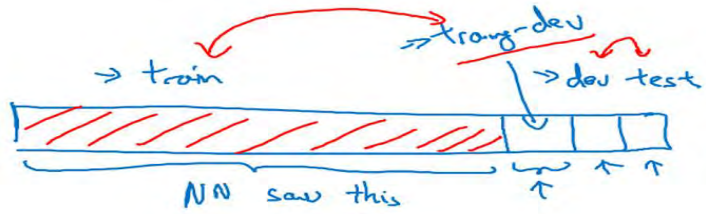
但如果你的训练数据和开发数据来自不同的分布，你就不能再放心下这个结论了。特别是，也许算法在开发集上做得不错，可能因为训练集很容易识别，因为训练集都是高分辨率图片，很清晰的图像，但开发集要难以识别得多。所以也许软件没有方差问题，这只不过反映了开发集包含更难准确分类的图片。所以这个分析的问题在于，当你看训练误差，再看开发误差，有两件事变了。首先算法只见过训练集数据，没见过开发集数据。第二，开发集数据来自不同的分布。而且因为你同时改变了两件事情，很难确认这增加的 9%误差率有多少是因为算法没看到开发集中的数据导致的，这是问题方差的部分，有多少是因为开发集数据就是不一样。

为了弄清楚哪个因素影响更大，如果你完全不懂这两种影响到底是什么，别担心我们马上会再讲一遍。但为了分辨清楚两个因素的影响，定义一组新的数据是有意义的，我们称之为训练-开发集，所以这是一个新的数据子集。我们应该从训练集的分布里挖出来，但你不会用来训练你的网络。

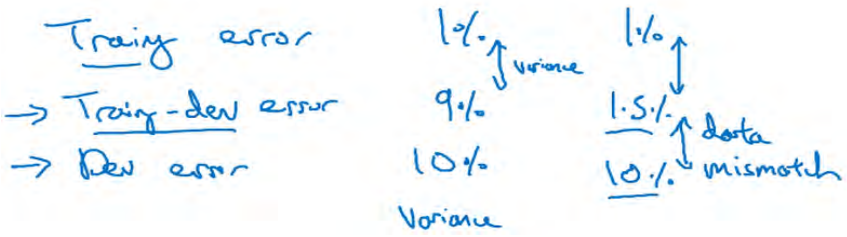
Training-dev set: Same distribution as training set, but not used for training

我的意思是我们已经设立过这样的训练集、开发集和测试集了，并且开发集和测试集来自相同的分布，但训练集来自不同的分布。

我们要做的是随机打散训练集，然后分出一部分训练集作为训练-开发集（training-dev），就像开发集和测试集来自同一分布，训练集、训练-开发集也来自同一分布。



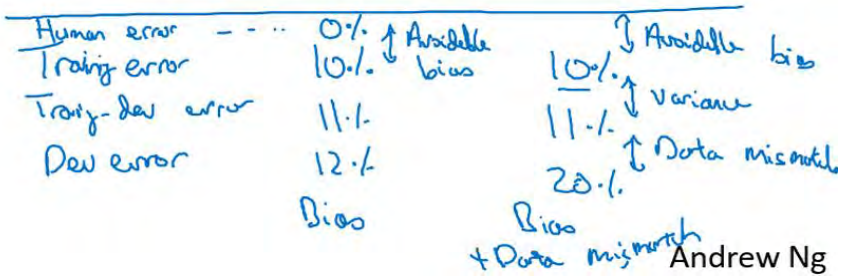
但不同的地方是，现在你只在训练集训练你的神经网络，你不会让神经网络在训练-开发集上跑后向传播。为了进行误差分析，你应该做的是看看分类器在训练集上的误差，训练-开发集上的误差，还有开发集上的误差。



比如说这个样本中，训练误差是 1%，我们说训练-开发集上的误差是 9%，然后开发集误差是 10%，和以前一样。你就可以从这里得到结论，当你从训练数据变到训练-开发集数据时，错误率真的上升了很多。而训练数据和训练-开发数据的差异在于，你的神经网络能看到第一部分数据并直接在上面做了训练，但没有在训练-开发集上直接训练，这就告诉你，算法存在方差问题，因为训练-开发集的错误率是在和训练集来自同一分布的数据中测得的。所以你知道，尽管你的神经网络在训练集中表现良好，但无法泛化到来自相同分布的训练-开发集里，它无法很好地泛化推广到来自同一分布，但以前没见过的数据中，所以在这个样

本中我们确实有一个方差问题。

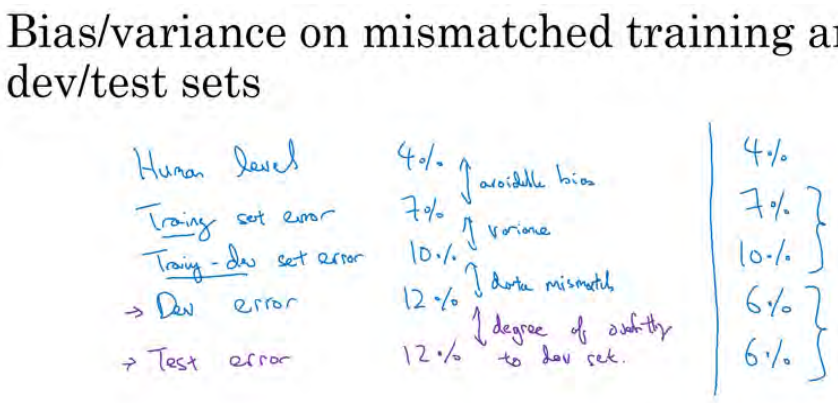
我们来看一个不同的样本，假设训练误差为 1%，训练-开发误差为 1.5%，但当你开始处理开发集时，错误率上升到 10%。现在你的方差问题就很小了，因为当你从见过的训练数据转到训练-开发集数据，神经网络还没有看到的数据，错误率只上升了一点点。但当你转到开发集时，错误率就大大上升了，所以这是数据不匹配的问题。因为你的学习算法没有直接在训练-开发集或者开发集训练过，但是这两个数据集来自不同的分布。但不管算法在学习什么，它在训练-开发集上做的很好，但开发集上做的不好，所以总之你的算法擅长处理和你关心的数据不同的分布，我们称之为数据不匹配的问题。



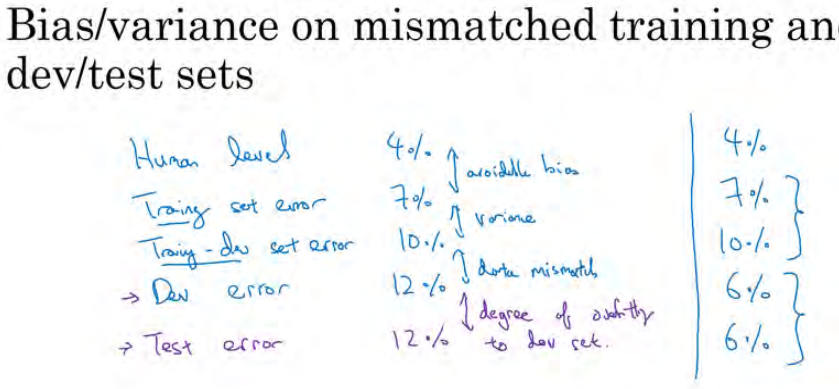
我们再来看几个样本，我会在下一行里写出来，因上面没空间了。所以训练误差、训练-开发误差、还有开发误差，我们说训练误差是 10%，训练-开发误差是 11%，开发误差为 12%，要记住，人类水平对贝叶斯错误率的估计大概是 0%，如果你得到了这种等级的表现，那就真的存在偏差问题了。存在可避免偏差问题，因为算法做的比人类水平差很多，所以这里的偏差真的很高。

最后一个例子，如果你的训练集错误率是 10%，你的训练-开发错误率是 11%，开发错误率是 20%，那么这其实有两个问题。第一，可避免偏差相当高，因为你在训练集上都没有做得很好，而人类能做到接近 0%错误率，但你的算法在训练集上错误率为 10%。这里方差似乎很小，但数据不匹配问题很大。所以对于这个样本，我说，如果你有很大的偏差或者可避免偏差问题，还有数据不匹配问题。

我们看看这张幻灯片里做了什么，然后写出一般的原则，我们要看的关键数据是人类水平错误率，你的训练集错误率，训练-开发集错误率，所以这分布和训练集一样，但你没有直接在上面训练。根据这些错误率之间差距有多大，你可以大概知道，可避免偏差、方差数据不匹配问题各自有多大。

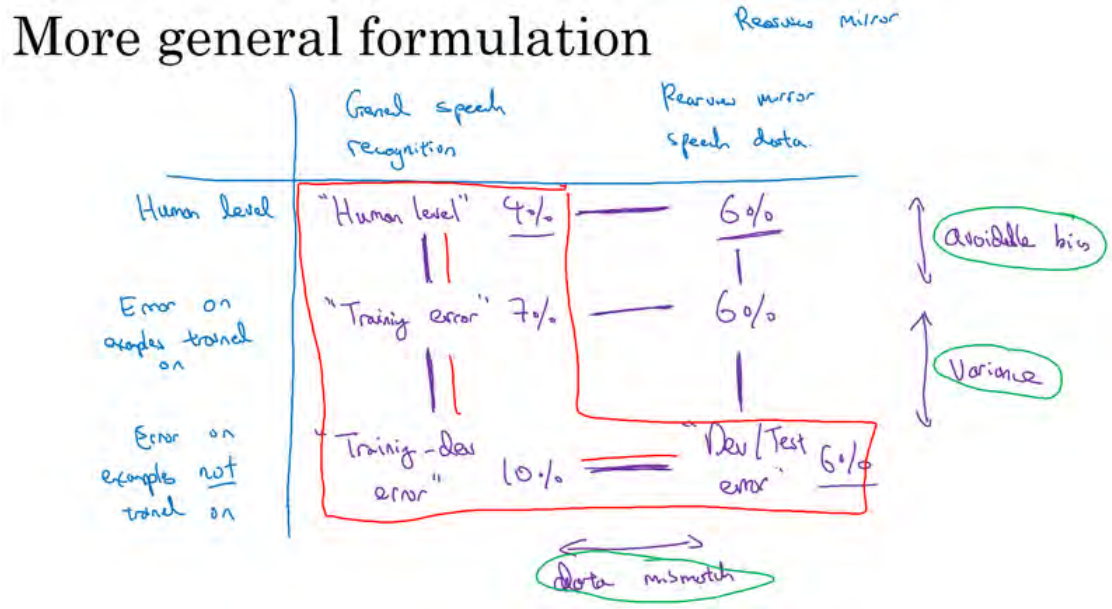


我们说人类水平错误率是 4%的话，你的训练错误率是 7%，而你的训练-开发错误率是 10%，而开发错误率是 12%，这样你就大概知道可避免偏差有多大。因为你知道，你希望你的算法至少要在训练集上的表现接近人类。而这大概表明了方差大小，所以你从训练集泛化推广到训练-开发集时效果如何？而这告诉你数据不匹配的问题大概有多大。技术上你还可以再加入一个数字，就是测试集表现，我们写成测试集错误率，你不应该在测试集上开发，因为你不希望对测试集过拟合。但如果你看看这个，那么这里的差距就说明你对开发集过拟合的程度。所以如果开发集表现和测试集表现有很大差距，那么你可能对开发集过拟合了，所以也许你需要一个更大的开发集，对吧？要记住，你的开发集和测试集来自同一分布，所以这里存在很大差距的话。如果算法在开发集上做的很好，比测试集好得多，那么你就可能对开发集过拟合了。如果是这种情况，那么你可能要往回退一步，然后收集更多开发集数据。现在我写出这些数字，这数字列表越往后数字越大。



这里还有个例子，其中数字并没有一直变大，也许人类的表现是 4%，训练错误率是 7%，训练-开发错误率是 10%。但我们看看开发集，你发现，很意外，算法在开发集上做的更好，也许是 6%。所以如果你见到这种现象，比如说在处理语音识别任务时发现这样，其中训练数据其实比你的开发集和测试集难识别得多。所以这两个（7%，10%）是从训练集分布评估的，而这两个（6%，6%）是从开发测试集分布评估的。所以有时候如果你的开发测试集分

布比你应用实际处理的数据要容易得多，那么这些错误率可能真的会下降。所以如果你看到这样的有趣的事情，可能需要比这个分析更普适的分析，我在下一张幻灯片里快速解释一下。

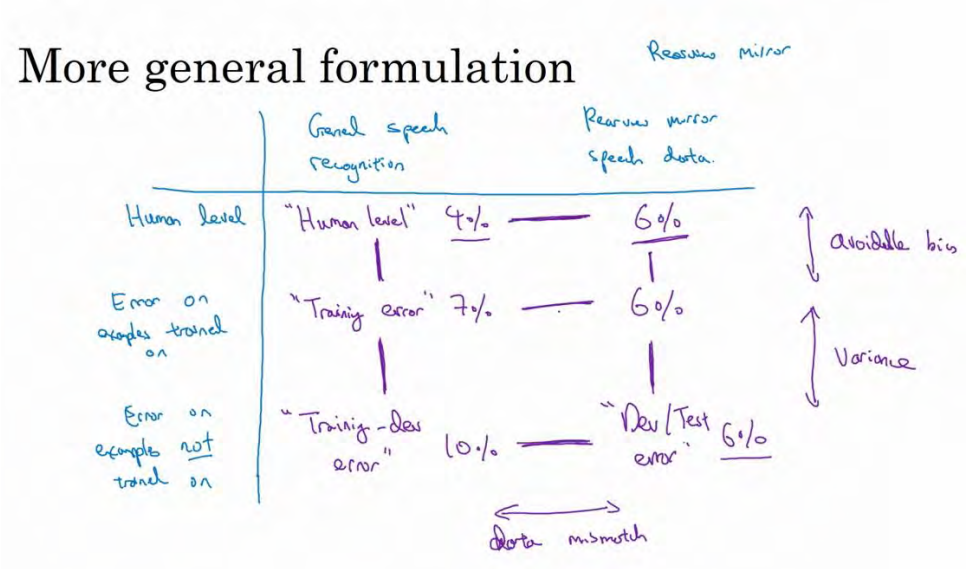


所以，我们就以语音激活后视镜为例子，事实证明，我们一直写出的数字可以放到一张表里，在水平轴上，我要放入不同的数据集。比如说，你可能从一般语音识别任务里得到很多数据，所以你可能会有一堆数据，来自小型智能音箱的语音识别问题的数据，你购买的数据等等。然后你收集了和后视镜有关的语音数据，在车里录的。所以这是表格的x轴，不同的数据集。在另一条轴上，我要标记处理数据不同的方式或算法。

首先，人类水平，人类处理这些数据集时准确度是多少。然后这是神经网络训练过的数据集上达到的错误率，然后还有神经网络没有训练过的数据集上达到的错误率。所以结果我们上一张幻灯片说是人类水平的错误率，数字填入这个单元格里（第二行第二列），人类对这一类数据处理得有多好，比如来自各种语音识别系统的数据，那些进入你的训练集的成千上万的语音片段，而上一张幻灯片中的例子是 4%。这个数字（7%），可能是我们的训练错误率，在上一张幻灯片中的例子中是 7%。是的，如果你的学习算法见过这个样本，在这个样本上跑过梯度下降，这个样本来自你的训练集分布或一般的语音识别数据分布，你的算法在训练过的数据中表现如何呢？然后这就是训练-开发集错误率，通常来自这个分布的错误率会高一点，一般的语音识别数据，如果你的算法没在来自这个分布的样本上训练过，它的表现如何呢？这就是我们说的训练-开发集错误率。

如果你移到右边去，这个单元格是开发集错误率，也可能是测试集错误，在刚刚的例子中是 6%。而开发集和测试集，实际上是两个数字，但都可以放入这个单元格里。如果你有来自后视镜的数据，来自从后视镜应用在车里实际录得的数据，但你的神经网络没有在这些

数据上做过反向传播，那么错误率是多少呢？

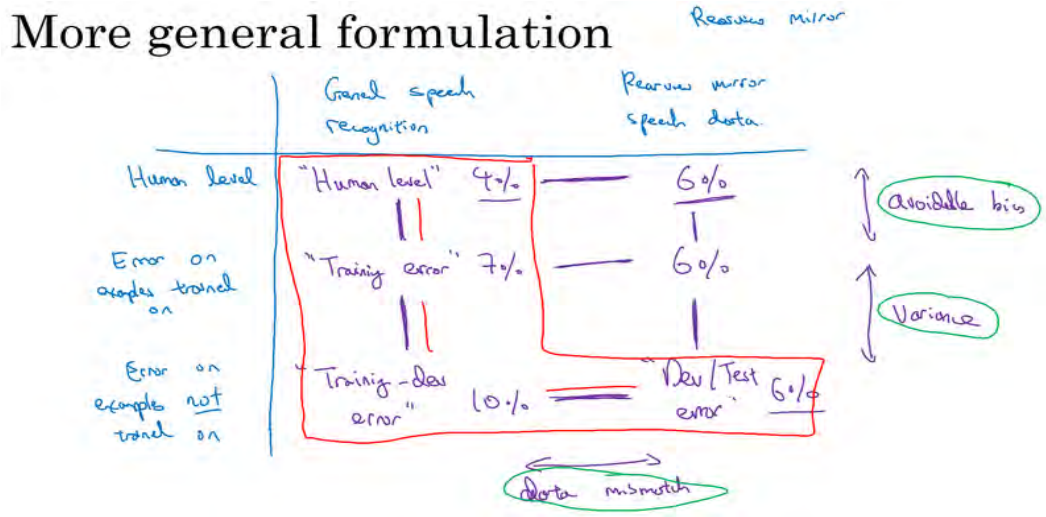


我们在上一张幻灯片作的分析是观察这两个数字之间的差异（**Human level 4%**和 **Training error 7%**），还有这两个数字之间（**Training error 7%**和 **Training-dev error 10%**），这两个数字之间（**Training-dev error 10%**和 **Dev/Test dev 6%**）。这个差距（**Human level 4%**和 **Training error 7%**）衡量了可避免偏差大小，这个差距 **Training error 7%**和 **Training-dev error 10%**）衡量了方差大小，而这个差距（**Training-dev error 10%**和 **Dev/Test dev 6%**）衡量了数据不匹配问题的大小。

事实证明，把剩下的两个数字（**rearview mirror speech data 6%**和 **Error on examples trained on 6%**），也放到这个表格里也是有用的。如果结果这也是 6%，那么你获得这个数字的方式是你让一些人自己标记他们的后视镜语音识别数据，看看人类在这个任务里能做多好，也许结果也是 6%。做法就是，你收集一些后视镜语音识别数据，把它放在训练集中，让神经网络去学习，然后测量那个数据子集上的错误率，但如果你得到这样的结果，好吧，那就是说你已经在后视镜语音数据上达到人类水平了，所以也许你对那个数据分布做的已经不错了。

当你继续进行更多分析时，分析并不一定会给你指明一条前进道路，但有时候你可能洞察到一些特征。比如比较这两个数字（**General speech recognition Human level 4%**和 **rearview mirror speech data 6%**），告诉我们对于人类来说，后视镜的语音数据实际上比一般语音识别更难，因为人类都有 6%的错误，而不是 4%的错误，但看看这个差值，你就可以了解到偏差和方差，还有数据不匹配这些问题的不同程度。所以更一般的分析方法是，我已经用过几次了。我还没用过，但对于很多问题来说检查这个子集的条目，看看这些差值，已经足够让

你往相对有希望的方向前进了。但有时候填满整个表格，你可能会洞察到更多特征。

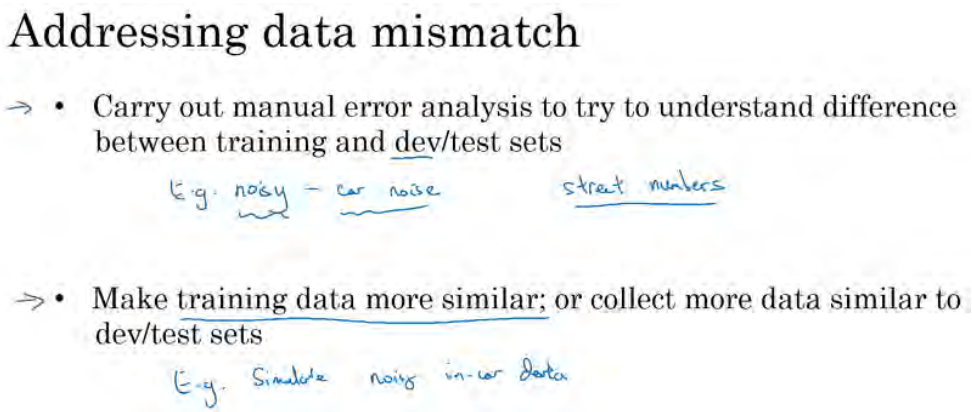


最后，我们以前讲过很多处理偏差的手段，讲过处理方差的手段，但怎么处理数据不匹配呢？特别是开发集、测试集和你的训练集数据来自不同分布时，这样可以用更多训练数据，真正帮你提高学习算法性能。但是，如果问题不仅来自偏差和方差，你现在又有了这个潜在的新问题，数据不匹配，有什么好办法可以处理数据不匹配的呢？实话说，并没有很通用，或者至少说是系统解决数据不匹配问题的方法，但你可以做一些尝试，可能会有帮助，我们在下一个视频里看看这些尝试。

所以我们讲了如何使用来自和开发集、测试集不同分布的训练数据，这可以给你提供更多训练数据，因此有助于提高你的学习算法的性能，但是，潜在问题就不只是偏差和方差问题，这样做会引入第三个潜在问题，数据不匹配。如果你做了错误分析，并发现数据不匹配是大量错误的来源，那么你怎么解决这个问题呢？但结果很不幸，并没有特别系统的方法去解决数据不匹配问题，但你可以做一些尝试，可能会有帮助，我们来看下一段视频。

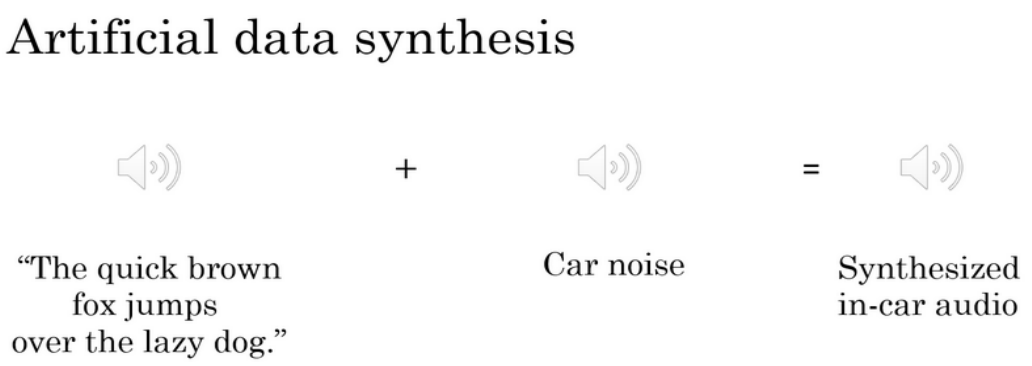
2.6 定位数据不匹配（Addressing data mismatch）

如果您的训练集来自和开发测试集不同的分布，如果错误分析显示你有一个数据不匹配的问题该怎么办？这个问题没有完全系统的解决方案，但我们可以看看一些可以尝试的事情。如果我发现有严重的数据不匹配问题，我通常会亲自做错误分析，尝试了解训练集和开发测试集的具体差异。技术上，为了避免对测试集过拟合，要做错误分析，你应该人工去看开发集而不是测试集。



但作为一个具体的例子，如果你正在开发一个语音激活的后视镜应用，你可能要看看……我想如果是语音的话，你可能要听一下来自开发集的样本，尝试弄清楚开发集和训练集到底有什么不同。所以，比如说你可能会发现很多开发集样本噪音很多，有很多汽车噪音，这是你的开发集和训练集差异之一。也许你还会发现其他错误，比如在你的车子里的语言激活后视镜，你发现它可能经常识别错误街道号码，因为那里有很多导航请求都有街道地址，所以得到正确的街道号码真的很重要。当你了解开发集误差的性质时，你就知道，开发集有可能跟训练集不同或者更难识别，那么你可以尝试把训练数据变得更像开发集一点，或者，你也可以收集更多类似你的开发集和测试集的数据。所以，比如说，如果你发现车辆背景噪音是主要的错误来源，那么你可以模拟车辆噪声数据，我会在下一张幻灯片里详细讨论这个问题。或者你发现很难识别街道号码，也许你可以有意识地收集更多人们说数字的音频数据，加到你的训练集里。

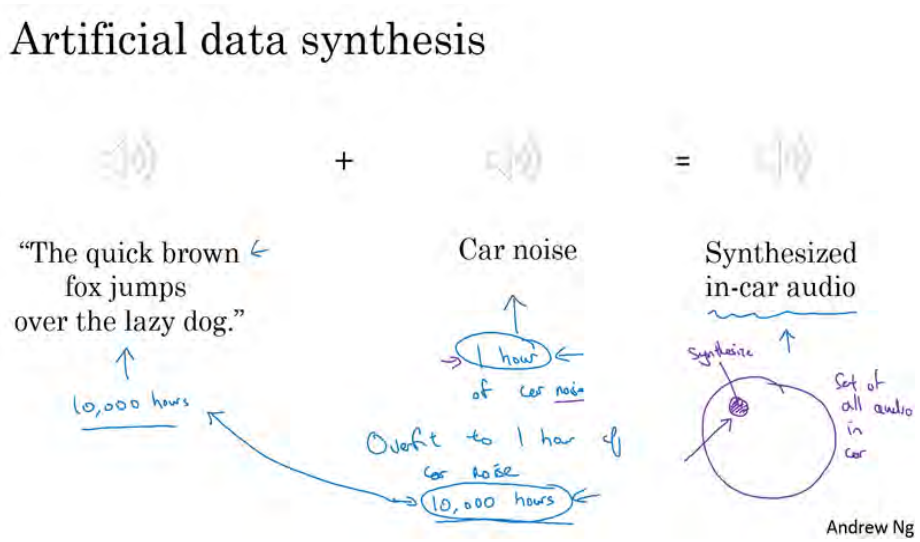
现在我知道这张幻灯片只给出了粗略的指南，列出一些你可以做的尝试，这不是一个系统化的过程，我想，这不能保证你一定能取得进展。但我发现这种人工见解，我们可以一起尝试收集更多和真正重要的场合相似的数据，这通常有助于解决很多问题。所以，如果你的目标是让训练数据更接近你的开发集，那么你可以怎么做呢？



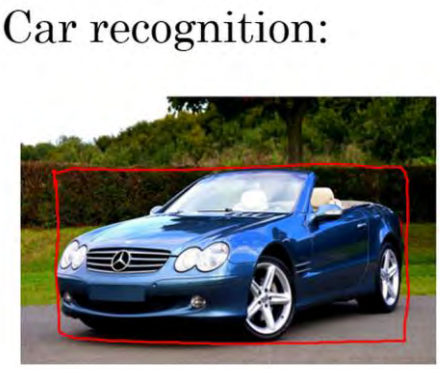
你可以利用的其中一种技术是人工合成数据（**artificial data synthesis**），我们讨论一下。在解决汽车噪音问题的场合，所以要建立语音识别系统。也许实际上你没那么多实际在汽车背景噪音下录得的音频，或者在高速公路背景噪音下录得的音频。但我们发现，你可以合成。所以假设你录制了大量清晰的音频，不带车辆背景噪音的音频，“**The quick brown fox jumps over the lazy dog**”（音频播放），所以，这可能是你的训练集里的一段音频，顺便说一下，这个句子在 AI 测试中经常使用，因为这个短句包含了从 a 到 z 所有字母，所以你会经常见到这个句子。但是，有了这个“**the quick brown fox jumps over the lazy dog**”这段录音之后，你也可以收集一段这样的汽车噪音，（播放汽车噪音音频）这就是汽车内部的背景噪音，如果你一言不发开车的话，就是这种声音。如果你把两个音频片段放到一起，你就可以合成出"**the quick brown fox jumps over the lazy dog**"（带有汽车噪声），在汽车背景噪音中的效果，听起来像这样，所以这是一个相对简单的音频合成例子。在实践中，你可能会合成其他音频效果，比如混响，就是声音从汽车内壁上反弹叠加的效果。

但是通过人工数据合成，你可以快速制造更多的训练数据，就像真的在车里录的那样，那就不需要花时间实际出去收集数据，比如说在实际行驶中的车子，录下上万小时的音频。所以，如果错误分析显示你应该尝试让你的数据听起来更像在车里录的，那么人工合成那种音频，然后喂给你的机器学习算法，这样做是合理的。

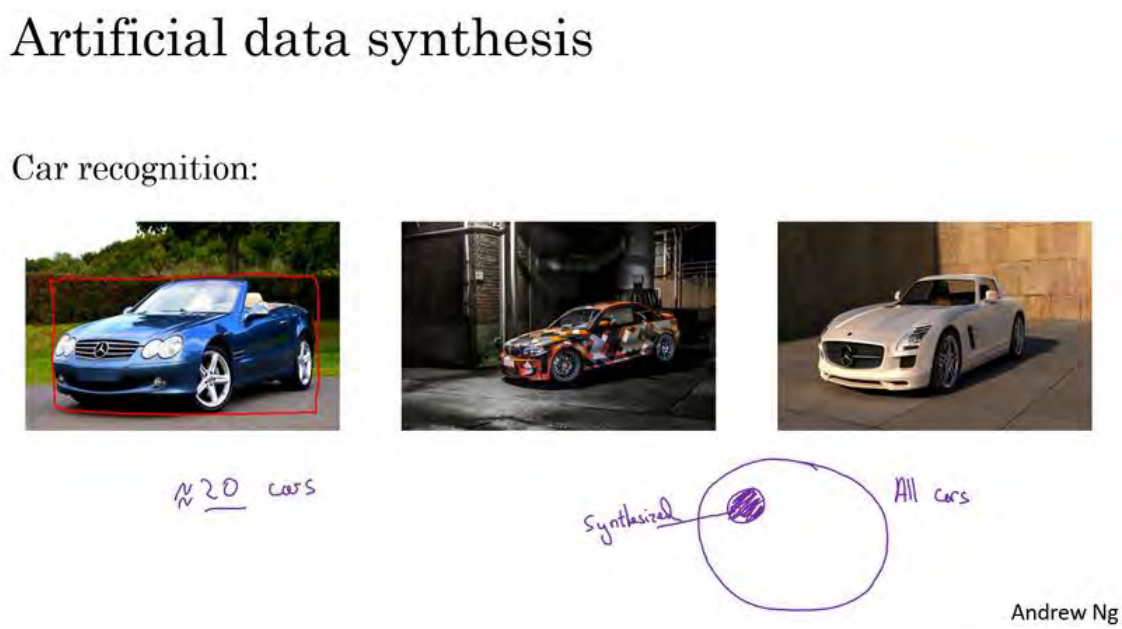
现在我们要提醒一下，人工数据合成有一个潜在问题，比如说，你在安静的背景里录得 10,000 小时音频数据，然后，比如说，你只录了一小时车辆背景噪音，那么，你可以这么做，将这 1 小时汽车噪音回放 10,000 次，并叠加到在安静的背景下录得的 10,000 小时数据。如果你这么做了，人听起来这个音频没什么问题。但是有一个风险，有可能你的学习算法对这 1 小时汽车噪音过拟合。特别是，如果这组汽车里录的音频可能是你可以想象的所有汽车噪音背景的集合，如果你只录了一小时汽车噪音，那你可能只模拟了全部数据空间的一小部分，你可能只从汽车噪音的很小的子集来合成数据。



而对于人耳来说，这些音频听起来没什么问题，因为一小时的车辆噪音对人耳来说，听起来和其他任意一小时车辆噪音是一样的。但你有可能从这整个空间很小的一个子集出发合成数据，神经网络最后可能对你这一小时汽车噪音过拟合。我不知道以较低成本收集 10,000 小时的汽车噪音是否可行，这样你就不用一遍又一遍地回放那 1 小时汽车噪音，你就有 10,000 个小时永不重复的汽车噪音来叠加到 10,000 小时安静背景下录得的永不重复的语音录音。这是可以做的，但不保证能做。但是使用 10,000 小时永不重复的汽车噪音，而不是 1 小时重复学习，算法有可能取得更好的性能。人工数据合成的挑战在于，人耳的话，人耳是无法分辨这 10,000 个小时听起来和那 1 小时没什么区别，所以你最后可能会制造出这个原始数据很少的，在一个小得多的空间子集合成的训练数据，但你自己没意识到。



这里有人工合成数据的另一个例子，假设你在研发无人驾驶汽车，你可能希望检测出这样的车，然后用这样的框包住它。很多人都讨论过的一个思路是，为什么不用计算机合成图像来模拟成千上万的车辆呢？事实上，这里有几张车辆照片（下图后两张图片），其实是用计算机合成的，我想这个合成是相当逼真的，我想通过这样合成图片，你可以训练出一个相当不错的计算机视觉系统来检测车子。



不幸的是，上一张幻灯片介绍的情况也会在这里出现，比如这是所有车的集合，如果你只合成这些车中很小的子集，对于人眼来说也许这样合成图像没什么问题，但你的学习算法可能会对合成的这一个小子集过拟合。特别是很多人都独立提出了一个想法，一旦你找到一个电脑游戏，里面车辆渲染的画面很逼真，那么就可以截图，得到数量巨大的汽车图片数据集。事实证明，如果你仔细观察一个视频游戏，如果这个游戏只有 20 辆独立的车，那么这游戏看起来还行。因为你是在游戏里开车，你只看到这 20 辆车，这个模拟看起来相当逼真。但现实世界里车辆的设计可不只 20 种，如果你用着 20 量独特的车合成的照片去训练系统，那么你的神经网络很可能对这 20 辆车过拟合，但人类很难分辨出来。即使这些图像看起来很逼真，你可能真的只用了所有可能出现的车辆的很小的子集。

所以，总而言之，如果你认为存在数据不匹配问题，我建议你做错误分析，或者看看训练集，或者看看开发集，试图找出，试图了解这两个数据分布到底有什么不同，然后看看是否有办法收集更多看起来像开发集的数据作训练。

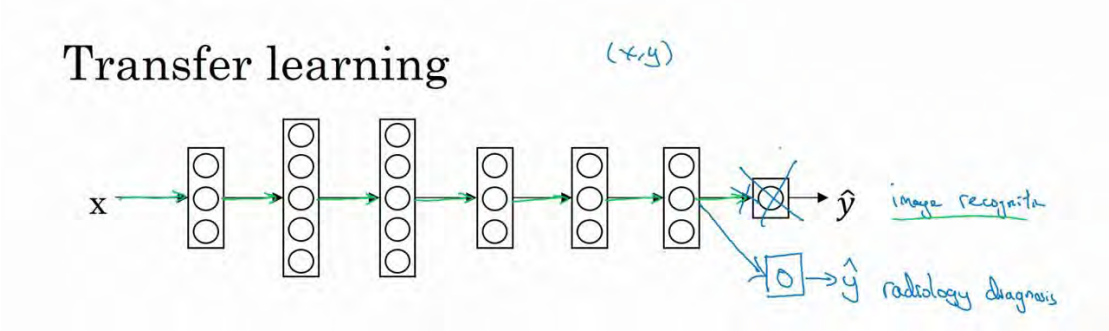
我们谈到其中一种办法是人工数据合成，人工数据合成确实有效。在语音识别中。我已经看到人工数据合成显著提升了已经非常好的语音识别系统的表现，所以这是可行的。但当你使用人工数据合成时，一定要谨慎，要记住你有可能从所有可能性的空间只选了很小一部分去模拟数据。

所以这就是如何处理数据不匹配问题，接下来，我想和你分享一些想法就是如何从多种类型的数据同时学习。

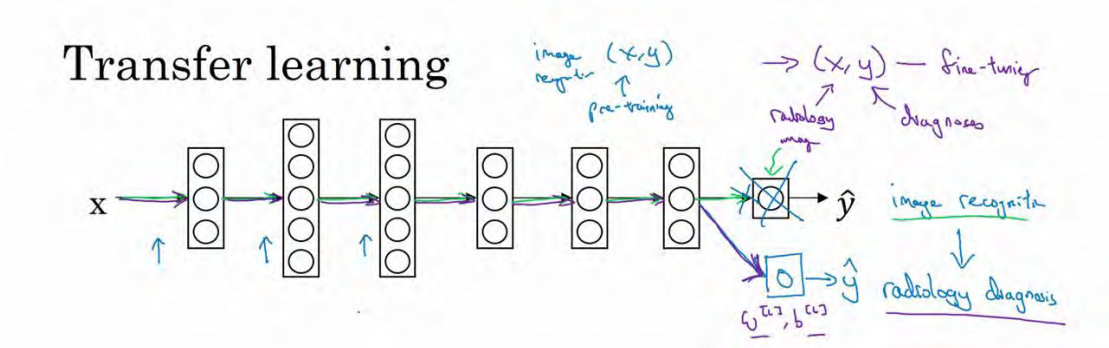
2.7 迁移学习（Transfer learning）

深度学习中，最强大的理念之一就是，有的时候神经网络可以从一个任务中习得知识，并将这些知识应用到另一个独立的任务中。所以例如，也许你已经训练好一个神经网络，能够识别像猫这样的对象，然后使用那些知识，或者部分习得的知识去帮助您更好地阅读 x 射线扫描图，这就是所谓的迁移学习。

我们来看看，假设你已经训练好一个图像识别神经网络，所以你首先用一个神经网络，并在(x,y)对上训练，其中x是图像，y是某些对象，图像是猫、狗、鸟或其他东西。如果你把这个神经网络拿来，然后让它适应或者说迁移，在不同任务中学到的知识，比如放射科诊断，就是说阅读X射线扫描图。你可以做的是把神经网络最后的输出层拿走，就把它删掉，还有进入到最后一层的权重删掉，然后为最后一层重新赋予随机权重，然后让它在放射诊断数据上训练。



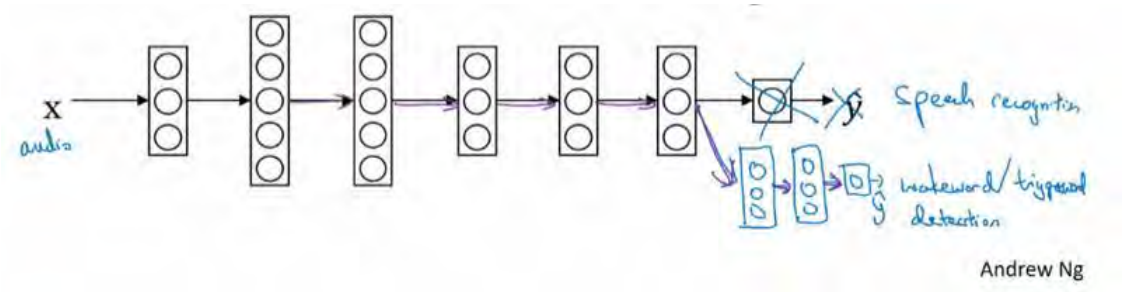
具体来说，在第一阶段训练过程中，当你进行图像识别任务训练时，你可以训练神经网络的所有常用参数，所有的权重，所有的层，然后你就得到了一个能够做图像识别预测的网络。在训练了这个神经网络后，要实现迁移学习，你现在要做的是，把数据集换成新的(x,y)对，现在这些变成放射科图像，而 y 是你想要预测的诊断，你要做的是初始化最后一层的权重，让我们称之为$w^{[L]}$和$b^{[L]}$随机初始化。



现在，我们在这个新数据集上重新训练网络，在新的放射科数据集上训练网络。要用放

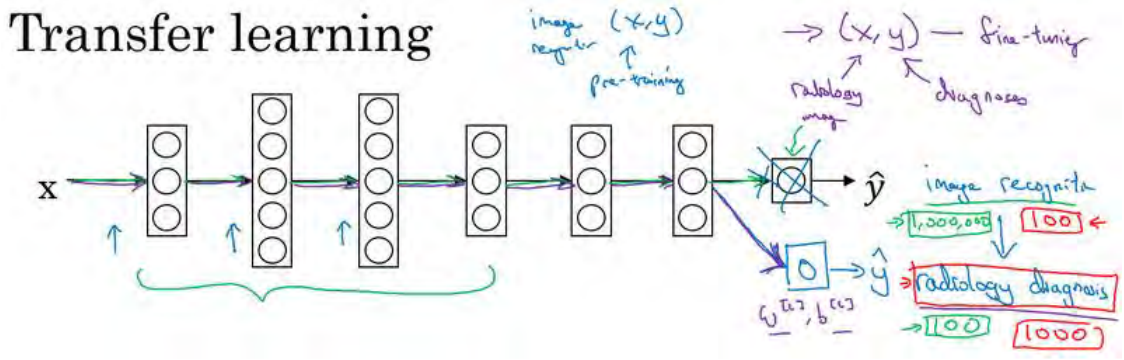
射科数据集重新训练神经网络有几种做法。你可能，如果你的放射科数据集很小，你可能只需要重新训练最后一层的权重，就是$w^{[l]}$和$b^{[l]}$并保持其他参数不变。如果你有足够多的数据，你可以重新训练神经网络中剩下的所有层。经验规则是，如果你有一个小数据集，就只训练输出层前的最后一层，或者也许是最后一两层。但是如果你有很多数据，那么也许你可以重新训练网络中的所有参数。如果你重新训练神经网络中的所有参数，那么这个在图像识别数据的初期训练阶段，有时称为预训练（**pre-training**），因为你在用图像识别数据去预先初始化，或者预训练神经网络的权重。然后，如果你以后更新所有权重，然后在放射科数据上训练，有时这个过程叫微调（**fine tuning**）。如果你在深度学习文献中看到预训练和微调，你就知道它们说的是这个意思，预训练和微调的权重来源于迁移学习。

在这个例子中你做的是，把图像识别中学到的知识应用或迁移到放射科诊断上来，为什么这样做有效果呢？有很多低层次特征，比如说边缘检测、曲线检测、阳性对象检测（**positive objects**），从非常大的图像识别数据库中习得这些能力可能有助于你的学习算法在放射科诊断中做得更好，算法学到了很多结构信息，图像形状的信息，其中一些知识可能会很有用，所以学会了图像识别，它就可能学到足够多的信息，可以了解不同图像的组成部分是怎样的，学到线条、点、曲线这些知识，也许对象的一小部分，这些知识有可能帮助你的放射科诊断网络学习更快一些，或者需要更少的学习数据。

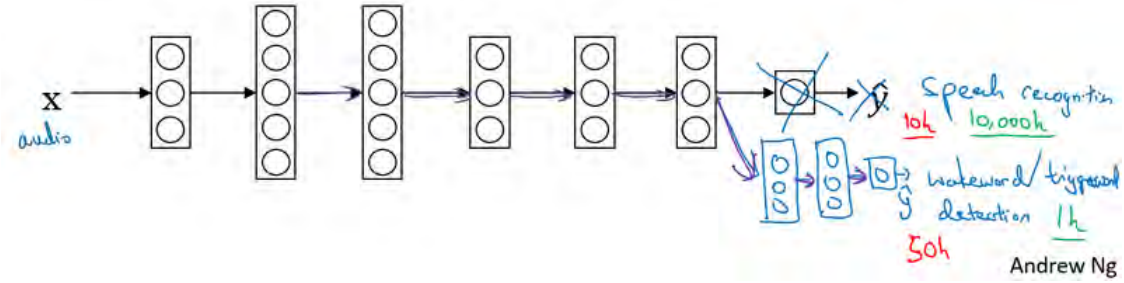


这里是另一个例子，假设你已经训练出一个语音识别系统，现在x是音频或音频片段输入，而y是听写文本，所以你已经训练了语音识别系统，让它输出听写文本。现在我们说你想搭建一个“唤醒词”或“触发词”检测系统，所谓唤醒词或触发词就是我们说的一句话，可以唤醒家里的语音控制设备，比如你说“**Alexa**”可以唤醒一个亚马逊 **Echo** 设备，或用“**OK Google**”来唤醒 **Google** 设备，用"**Hey Siri**"来唤醒苹果设备，用"你好百度"唤醒一个百度设备。要做到这点，你可能需要去掉神经网络的最后一层，然后加入新的输出节点，但有时你可以不只加入一个新节点，或者甚至往你的神经网络加入几个新层，然后把唤醒词检测问题的标签y喂进去训练。再次，这取决于你有多少数据，你可能只需要重新训练网络的新层，也许你需要重新训练神经网络中更多的层。

那么迁移学习什么时候是有意义的呢？迁移学习起作用的场合是，在迁移来源问题中你有很多数据，但迁移目标问题你没有那么多数据。例如，假设图像识别任务中你有 1 百万个样本，所以这里数据相当多。可以学习低层次特征，可以在神经网络的前面几层学到如何识别很多有用的特征。但是对于放射科任务，也许你只有一百个样本，所以你的放射学诊断问题数据很少，也许只有 100 次X射线扫描，所以你从图像识别训练中学到的很多知识可以迁移，并且真正帮你加强放射科识别任务的性能，即使你的放射科数据很少。



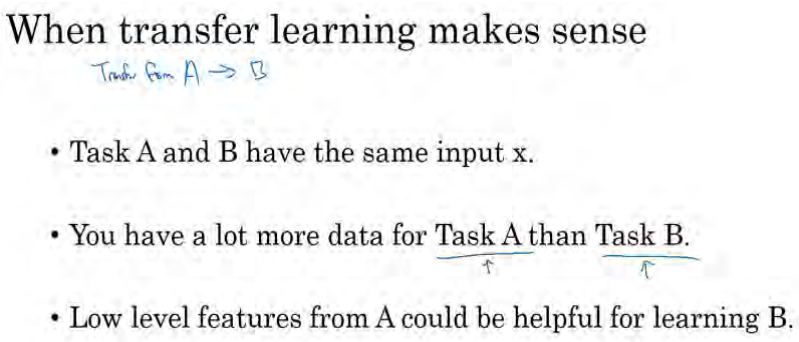
对于语音识别，也许你已经用 10,000 小时数据训练过你的语言识别系统，所以你从这 10,000 小时数据学到了很多人类声音的特征，这数据量其实很多了。但对于触发字检测，也许你只有 1 小时数据，所以这数据太小，不能用来拟合很多参数。所以在这种情况下，预先学到很多人类声音的特征人类语言的组成部分等等知识，可以帮你建立一个很好的唤醒字检测器，即使你的数据集相对较小。对于唤醒词任务来说，至少数据集要小得多。



所以在这两种情况下，你从数据量很多的问题迁移到数据量相对小的问题。然后反过来的话，迁移学习可能就没有意义了。比如，你用 100 张图训练图像识别系统，然后有 100 甚至 1000 张图用于训练放射科诊断系统，人们可能会想，为了提升放射科诊断的性能，假设你真的希望这个放射科诊断系统做得好，那么用放射科图像训练可能比使用猫和狗的图像更有价值，所以这里（100 甚至 1000 张图用于训练放射科诊断系统）的每个样本价值比这里（100 张图训练图像识别系统）要大得多，至少就建立性能良好的放射科系统而言是这样。所以，如果你的放射科数据更多，那么你这 100 张猫猫狗狗或者随机物体的图片肯定不会有太大帮助，因为来自猫狗识别任务中，每一张图的价值肯定不如一张X射线扫描图有价值，

对于建立良好的放射科诊断系统而言是这样。

所以，这是其中一个例子，说明迁移学习可能不会有害，但也别指望这么做可以带来有意义的增益。同样，如果你用 10 小时数据训练出一个语音识别系统。然后你实际上有 10 个小时甚至更多，比如说 50 个小时唤醒字检测的数据，你知道迁移学习有可能会有帮助，也可能不会，也许把这 10 小时数据迁移学习不会有太大坏处，但是你也别指望会得到有意义的增益。



所以总结一下，什么时候迁移学习是有意义的？如果你想从任务A学习并迁移一些知识到任务B，那么当任务A和任务B都有同样的输入x时，迁移学习是有意义的。在第一个例子中，A和B的输入都是图像，在第二个例子中，两者输入都是音频。当任务A的数据比任务B多得多时，迁移学习意义更大。所有这些假设的前提都是，你希望提高任务B的性能，因为任务B每个数据更有价值，对任务B来说通常任务A的数据量必须大得多，才有帮助，因为任务A里单个样本的价值没有比任务B单个样本价值大。然后如果你觉得任务A的低层次特征，可以帮助任务B的学习，那迁移学习更有意义一些。

而在这两个前面的例子中，也许学习图像识别教给系统足够多图像相关的知识，让它可以进行放射科诊断，也许学习语音识别教给系统足够多人类语言信息，能帮助你开发触发字或唤醒字检测器。

所以总结一下，迁移学习最有用的场合是，如果你尝试优化任务 B 的性能，通常这个任务数据相对较少，例如，在放射科中你知道很难收集很多X射线扫描图来搭建一个性能良好的放射科诊断系统，所以在这种情况下，你可能会找一个相关但不同的任务，如图像识别，其中你可能用 1 百万张图片训练过了，并从中学到很多低层次特征，所以那也许能帮助网络在任务B在放射科任务上做得更好，尽管任务B没有这么多数据。迁移学习什么时候是有意义的？它确实可以显著提高你的学习任务的性能，但我有时候也见过有些场合使用迁移学习时，任务A实际上数据量比任务B要少，这种情况下增益可能不多。

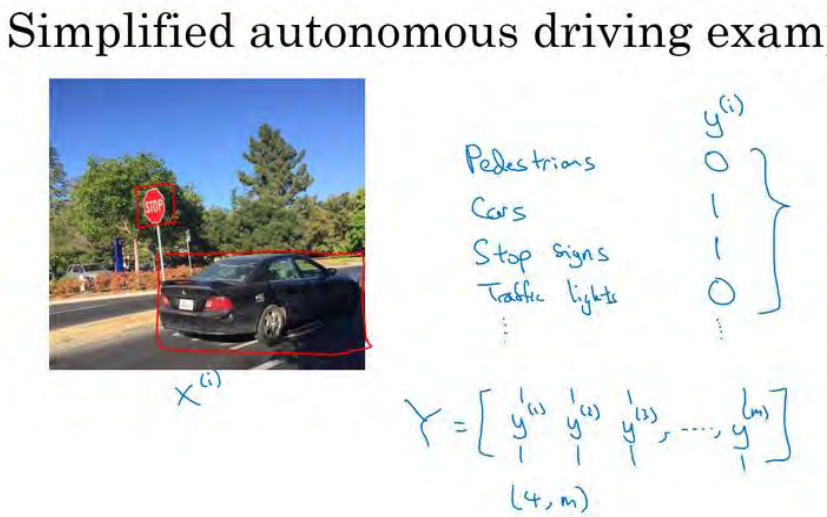
好，这就是迁移学习，你从一个任务中学习，然后尝试迁移到另一个不同任务中。从多

个任务中学习还有另外一个版本，就是所谓的多任务学习，当你尝试从多个任务中并行学习，而不是串行学习，在训练了一个任务之后试图迁移到另一个任务，所以在下一个视频中，让我们来讨论多任务学习。

2.8 多任务学习（Multi-task learning）

在迁移学习中，你的步骤是串行的，你从任务 A 里学习只是然后迁移到任务 B。在多任务学习中，你是同时开始学习的，试图让单个神经网络同时做几件事情，然后希望这里每个任务都能帮到其他所有任务。

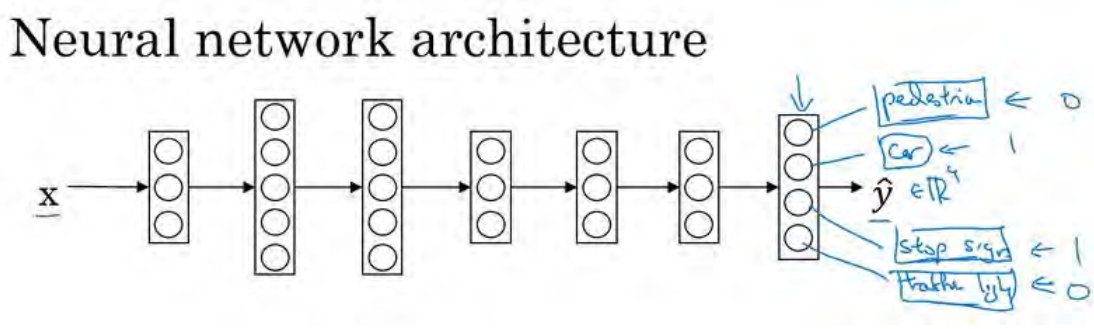
我们来看一个例子，假设你在研发无人驾驶车辆，那么你的无人驾驶车可能需要同时检测不同的物体，比如检测行人、车辆、停车标志，还有交通灯各种其他东西。比如在左边这个例子中，图像里有个停车标志，然后图像中有辆车，但没有行人，也没有交通灯。



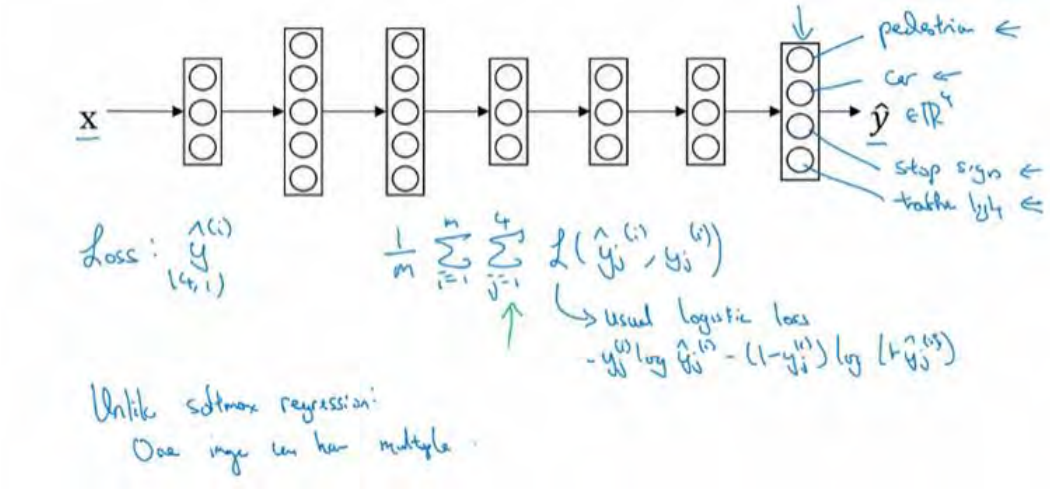
如果这是输入图像$x^{(i)}$，那么这里不再是一个标签 $y^{(i)}$，而是有 4 个标签。在这个例子中，没有行人，有一辆车，有一个停车标志，没有交通灯。然后如果你尝试检测其他物体，也许 $y^{(i)}$的维数会更高，现在我们就先用 4 个吧，所以 $y^{(i)}$是个 4×1 向量。如果你从整体来看这个训练集标签和以前类似，我们将训练集的标签水平堆叠起来，像这样$y^{(1)}$一直到$y^{(m)}$：

$$Y = \begin{bmatrix} | & | & | & \dots & | \\ y^{(1)} & y^{(2)} & y^{(3)} & \dots & y^{(m)} \\ | & | & | & \dots & | \end{bmatrix}$$

不过现在$y^{(i)}$是 4×1 向量，所以这些都是竖向的列向量，所以这个矩阵Y现在变成$4 \times m$矩阵。而之前，当y是单实数时，这就是$1 \times m$矩阵。



那么你现在可以做的是训练一个神经网络，来预测这些y值，你就得到这样的神经网络，输入x，现在输出是一个四维向量y。请注意，这里输出我画了四个节点，所以第一个节点就是我们想预测图中有没有行人，然后第二个输出节点预测的是有没有车，这里预测有没有停车标志，这里预测有没有交通灯，所以这里$\hat{y}$是四维的。



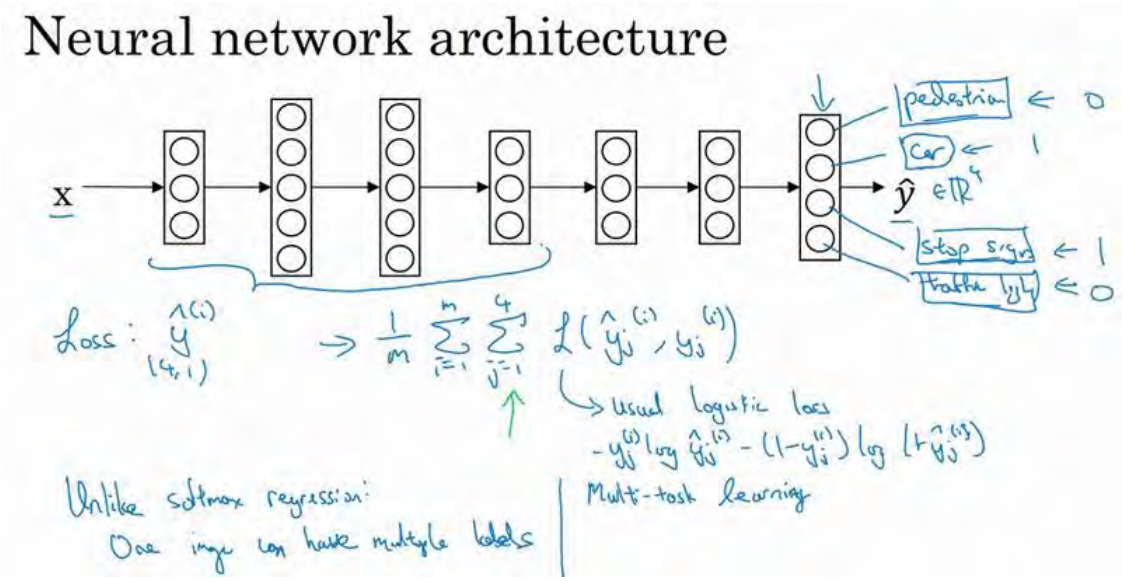
要训练这个神经网络，你现在需要定义神经网络的损失函数，对于一个输出$\hat{y}$，是个 4 维向量，对于整个训练集的平均损失：

$$\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} L(\hat{y}_j^{(i)}, y_j^{(i)})$$

$\sum_{j=1}^{4} L(\hat{y}_j^{(i)}, y_j^{(i)})$这些单个预测的损失，所以这就是对四个分量的求和，行人、车、停车标志、交通灯，而这个标志 L 指的是 **logistic 损失**，我们就这么写：

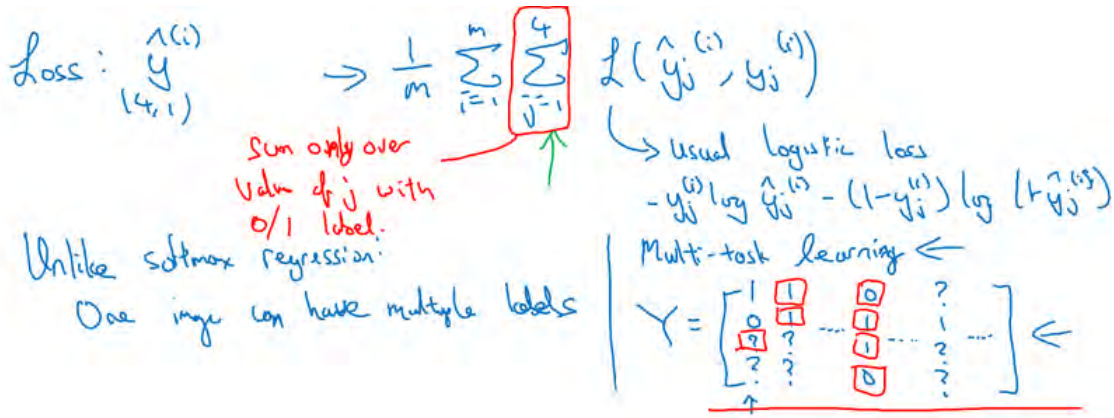
$$L(\hat{y}_j^{(i)}, y_j^{(i)}) = -y_j^{(i)} \log \hat{y}_j^{(i)} - (1 - y_j^{(i)}) log(1 - \hat{y}_j^{(i)})$$

整个训练集的平均损失和之前分类猫的例子主要区别在于，现在你要对$j = 1$到4求和，这与 **softmax** 回归的主要区别在于，与 **softmax** 回归不同，**softmax** 将单个标签分配给单个样本。



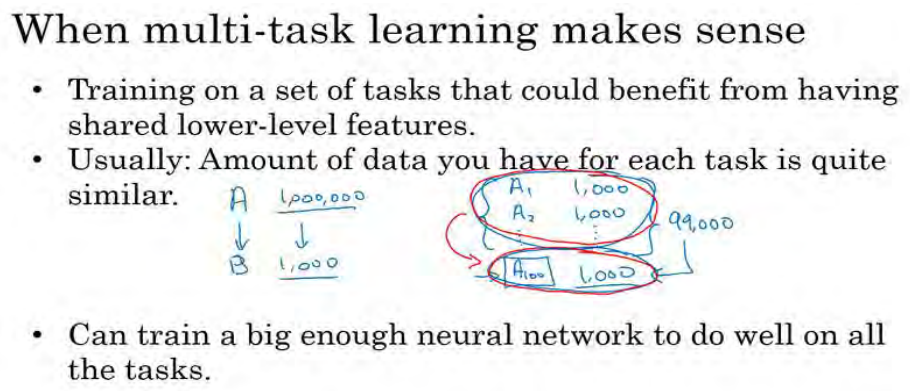
而这张图可以有很多不同的标签，所以不是说每张图都只是一张行人图片，汽车图片、停车标志图片或者交通灯图片。你要知道每张照片是否有行人、或汽车、停车标志或交通灯，多个物体可能同时出现在一张图里。实际上，在上一张幻灯片中，那张图同时有车和停车标志，但没有行人和交通灯，所以你不是只给图片一个标签，而是需要遍历不同类型，然后看看每个类型，那类物体有没有出现在图中。所以我就说在这个场合，一张图可以有多个标签。如果你训练了一个神经网络，试图最小化这个成本函数，你做的就是多任务学习。因为你现在做的是建立单个神经网络，观察每张图，然后解决四个问题，系统试图告诉你，每张图里面有没有这四个物体。另外你也可以训练四个不同的神经网络，而不是训练一个网络做四件事情。但神经网络一些早期特征，在识别不同物体时都会用到，然后你发现，训练一个神经网络做四件事情会比训练四个完全独立的神经网络分别做四件事性能要更好，这就是多任务学习的力量。

另一个细节，到目前为止，我是这么描述算法的，好像每张图都有全部标签。事实证明，多任务学习也可以处理图像只有部分物体被标记的情况。所以第一个训练样本，我们说有人，给数据贴标签的人告诉你里面有一个行人，没有车，但他们没有标记是否有停车标志，或者是否有交通灯。也许第二个例子中，有行人，有车。但是，当标记人看着那张图片时，他们没有加标签，没有标记是否有停车标志，是否有交通灯等等。也许有些样本都有标记，但也许有些样本他们只标记了有没有车，然后还有一些是问号。



即使是这样的数据集，你也可以在上面训练算法，同时做四个任务，即使一些图像只有一小部分标签，其他是问号或者不管是什么。然后你训练算法的方式，即使这里有些标签是问号，或者没有标记，这就是对j从 1 到 4 求和，你就只对带 0 和 1 标签的 j 值求和，所以当有问号的时候，你就在求和时忽略那个项，这样只对有标签的值求和，于是你就能利用这样的数据集。

那么多任务学习什么时候有意义呢？当三件事为真时，它就是有意义的。



第一，如果你训练的一组任务，可以共用低层次特征。对于无人驾驶的例子，同时识别交通灯、汽车和行人是有道理的，这些物体有相似的特征，也许能帮你识别停车标志，因为这些都是道路上的特征。

第二，这个准则没有那么绝对，所以不一定是对的。但我从很多成功的多任务学习案例中看到，如果每个任务的数据量很接近，你还记得迁移学习时，你从A任务学到知识然后迁移到B任务，所以如果任务A有 1 百万个样本，任务B只有 1000 个样本，那么你从这 1 百万个样本学到的知识，真的可以帮你增强对更小数据集任务B的训练。那么多任务学习又怎么样呢？在多任务学习中，你通常有更多任务而不仅仅是两个，所以也许你有，以前我们有 4 个任务，但比如说你要完成 100 个任务，而你要做多任务学习，尝试同时识别 100 种不同类型的物体。你可能会发现，每个任务大概有 1000 个样本。所以如果你专注加强单个任务的

性能，比如我们专注加强第 100 个任务的表现，我们用$A100$表示，如果你试图单独去做这个最后的任务，你只有 1000 个样本去训练这个任务，这是 100 项任务之一，而通过在其他 99 项任务的训练，这些加起来可以一共有 99000 个样本，这可能大幅提升算法性能，可以提供很多知识来增强这个任务的性能。不然对于任务$A100$，只有 1000 个样本的训练集，效果可能会很差。如果有对称性，这其他 99 个任务，也许能提供一些数据或提供一些知识来帮到这 100 个任务中的每一个任务。所以第二点不是绝对正确的准则，但我通常会看的是如果你专注于单项任务，如果想要从多任务学习得到很大性能提升，那么其他任务加起来必须要有比单个任务大得多的数据量。要满足这个条件，其中一种方法是，比如右边这个例子这样，或者如果每个任务中的数据量很相近，但关键在于，如果对于单个任务你已经有 1000 个样本了，那么对于所有其他任务，你最好有超过 1000 个样本，这样其他任务的知识才能帮你改善这个任务的性能。

最后多任务学习往往在以下场合更有意义，当你可以训练一个足够大的神经网络，同时做好所有的工作，所以多任务学习的替代方法是为每个任务训练一个单独的神经网络。所以不是训练单个神经网络同时处理行人、汽车、停车标志和交通灯检测。你可以训练一个用于行人检测的神经网络，一个用于汽车检测的神经网络，一个用于停车标志检测的神经网络和一个用于交通信号灯检测的神经网络。那么研究员 **Rich Carona** 几年前发现的是什么呢？多任务学习会降低性能的唯一情况，和训练单个神经网络相比性能更低的情况就是你的神经网络还不够大。但如果你可以训练一个足够大的神经网络，那么多任务学习肯定不会或者很少会降低性能，我们都希望它可以提升性能，比单独训练神经网络来单独完成各个任务性能要更好。

所以这就是多任务学习，在实践中，多任务学习的使用频率要低于迁移学习。我看到很多迁移学习的应用，你需要解决一个问题，但你的训练数据很少，所以你需要找一个数据很多的相关问题来预先学习，并将知识迁移到这个新问题上。但多任务学习比较少见，就是你需要同时处理很多任务，都要做好，你可以同时训练所有这些任务，也许计算机视觉是一个例子。在物体检测中，我们看到更多使用多任务学习的应用，其中一个神经网络尝试检测一大堆物体，比分别训练不同的神经网络检测物体更好。但我说，平均来说，目前迁移学习使用频率更高，比多任务学习频率要高，但两者都可以成为你的强力工具。

所以总结一下，多任务学习能让你训练一个神经网络来执行许多任务，这可以给你更高的性能，比单独完成各个任务更高的性能。但要注意，实际上迁移学习比多任务学习使用频率更高。我看到很多任务都是，如果你想解决一个机器学习问题，但你的数据集相对较小，

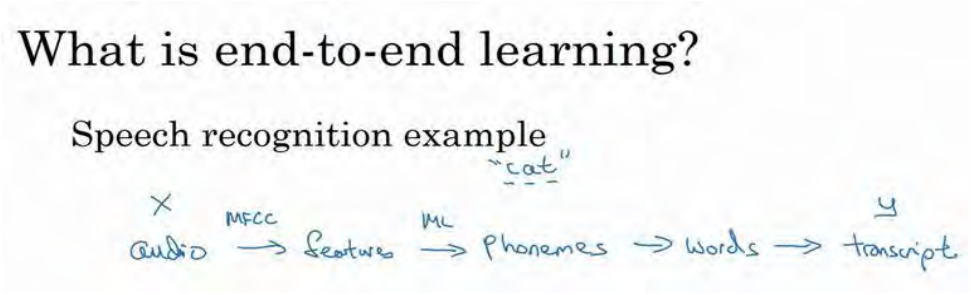
那么迁移学习真的能帮到你，就是如果你找到一个相关问题，其中数据量要大得多，你就能以它为基础训练你的神经网络，然后迁移到这个数据量很少的任务上来。

今天我们学到了很多和迁移学习有关的问题，还有一些迁移学习和多任务学习的应用。但多任务学习，我觉得使用频率比迁移学习要少得多，也许其中一个例外是计算机视觉，物体检测。在那些任务中，人们经常训练一个神经网络同时检测很多不同物体，这比训练单独的神经网络来检测视觉物体要更好。但平均而言，我认为即使迁移学习和多任务学习工作方式类似。实际上，我看到用迁移学习比多任务学习要更多，我觉得这是因为你很难找到那么多相似且数据量对等的任务可以用单一神经网络训练。再次，在计算机视觉领域，物体检测这个例子是最显著的例外情况。

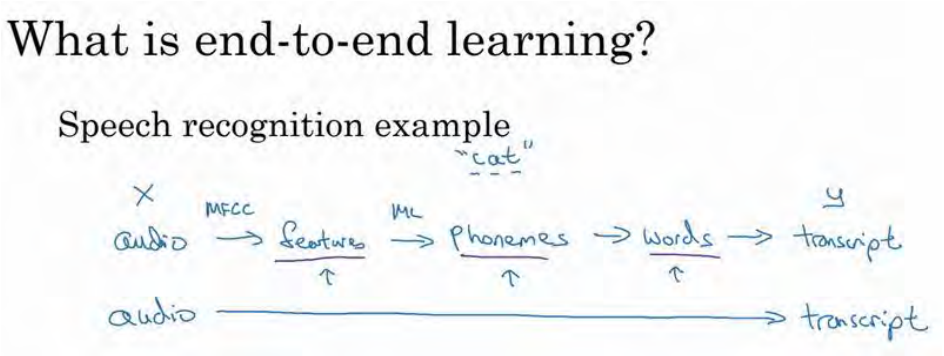
所以这就是多任务学习，多任务学习和迁移学习都是你的工具包中的重要工具。最后，我想继续讨论端到端深度学习，所以我们来看下一个视频来讨论端到端学习。

2.9 什么是端到端的深度学习？（What is end-to-end deep learning?）

深度学习中最令人振奋的最新动态之一就是端到端深度学习的兴起，那么端到端学习到底是什么呢？简而言之，以前有一些数据处理系统或者学习系统，它们需要多个阶段的处理。那么端到端深度学习就是忽略所有这些不同的阶段，用单个神经网络代替它。

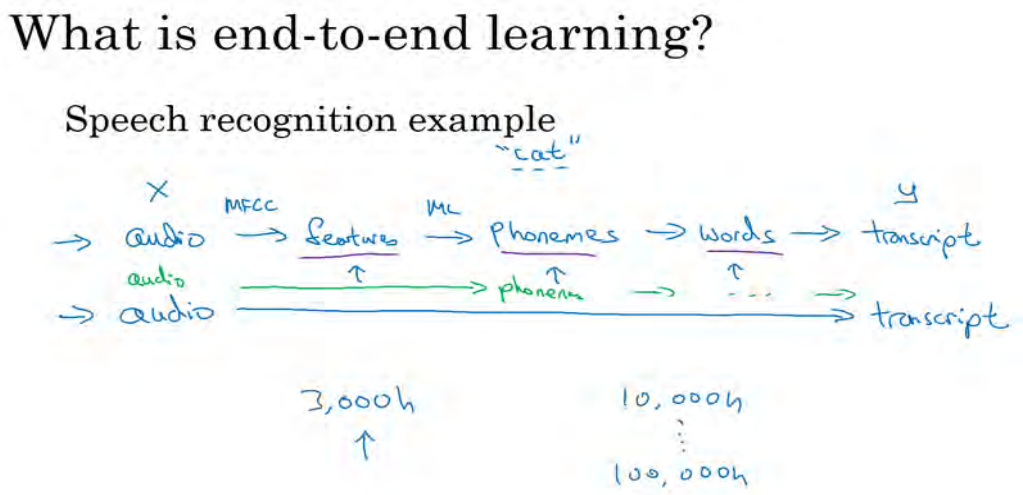


我们来看一些例子，以语音识别为例，你的目标是输入x，比如说一段音频，然后把它映射到一个输出y，就是这段音频的听写文本。所以传统上，语音识别需要很多阶段的处理。首先你会提取一些特征，一些手工设计的音频特征，也许你听过 **MFCC**，这种算法是用来从音频中提取一组特定的人工设计的特征。在提取出一些低层次特征之后，你可以应用机器学习算法在音频片段中找到音位，所以音位是声音的基本单位，比如说**“Cat”**这个词是三个音节构成的，**Cu-**、**Ah-**和 **Tu-**，算法就把这三个音位提取出来，然后你将音位串在一起构成独立的词，然后你将词串起来构成音频片段的听写文本。



所以和这种有很多阶段的流水线相比，端到端深度学习做的是，你训练一个巨大的神经网络，输入就是一段音频，输出直接是听写文本。**AI** 的其中一个有趣的社会学效应是，随着端到端深度学习系统表现开始更好，有一些花了大量时间或者整个事业生涯设计出流水线各个步骤的研究员，还有其他领域的研究员，不只是语言识别领域的，也许是计算机视觉，还有其他领域，他们花了大量的时间，写了很多论文，有些甚至整个职业生涯的一大部分都投

入到开发这个流水线的功能或者其他构件上去了。而端到端深度学习就只需要把训练集拿过来，直接学到了x和y之间的函数映射，直接绕过了其中很多步骤。对一些学科里的人来说，这点相当难以接受，他们无法接受这样构建 AI 系统，因为有些情况，端到端方法完全取代了旧系统，某些投入了多年研究的中间组件也许已经过时了。

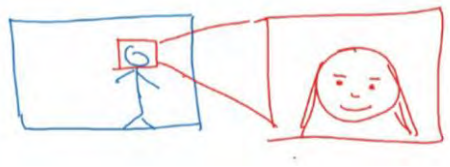


事实证明，端到端深度学习的挑战之一是，你可能需要大量数据才能让系统表现良好，比如，你只有 3000 小时数据去训练你的语音识别系统，那么传统的流水线效果真的很好。但当你拥有非常大的数据集时，比如 10,000 小时数据或者 100,000 小时数据，这样端到端方法突然开始很厉害了。所以当你的数据集较小的时候，传统流水线方法其实效果也不错，通常做得更好。你需要大数据集才能让端到端方法真正发出耀眼光芒。如果你的数据量适中，那么也可以用中间件方法，你可能输入还是音频，然后绕过特征提取，直接尝试从神经网络输出音位，然后也可以在其他阶段用，所以这是往端到端学习迈出的一小步，但还没有到那里。

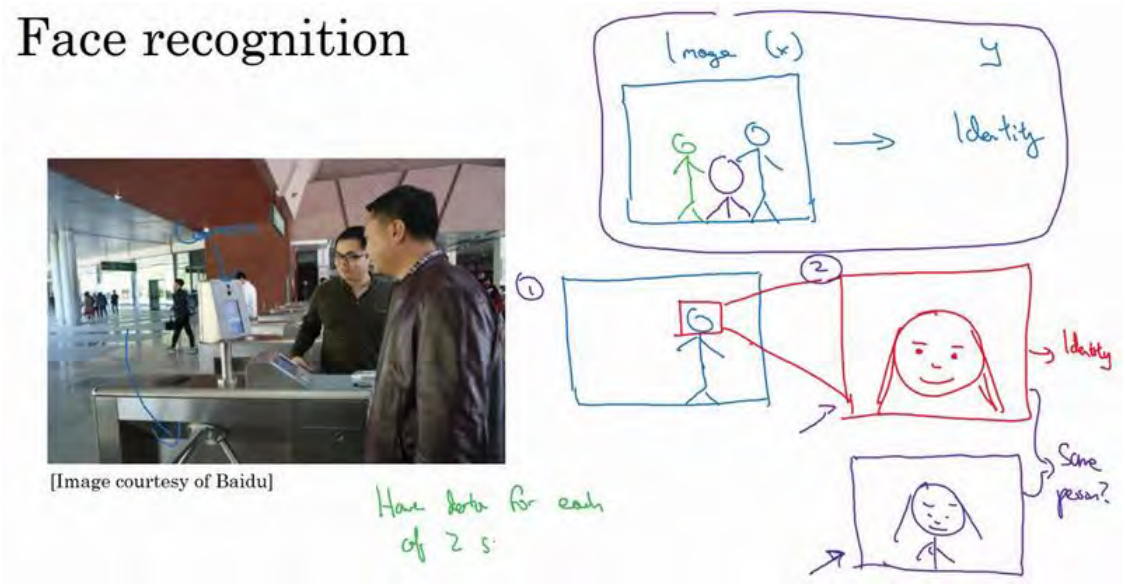
[Image courtesy of Baidu]

这张图上是一个研究员做的人脸识别门禁，是百度的林元庆研究员做的。这是一个相机，它会拍下接近门禁的人，如果它认出了那个人，门禁系统就自动打开，让他通过，所以你不需要刷一个 **RFID** 工卡就能进入这个设施。系统部署在越来越多的中国办公室，希望在其他国家也可以部署更多，你可以接近门禁，如果它认出你的脸，它就直接让你通过，你不需要带 **RFID** 工卡。

那么，怎么搭建这样的系统呢？你可以做的第一件事是，看看相机拍到的照片，对吧？我想我画的不太好，但也许这是相机照片，你知道，有人接近门禁了，所以这可能是相机拍到的图像x。有件事你可以做，就是尝试直接学习图像x到人物y身份的函数映射，事实证明这不是最好的方法。其中一个问题是，人可以从很多不同的角度接近门禁，他们可能在绿色位置，可能在蓝色位置。有时他们更靠近相机，所以他们看起来更大，有时候他们非常接近相机，那照片中脸就很大了。在实际研制这些门禁系统时，他不是直接将原始照片喂到一个神经网络，试图找出一个人的身份。

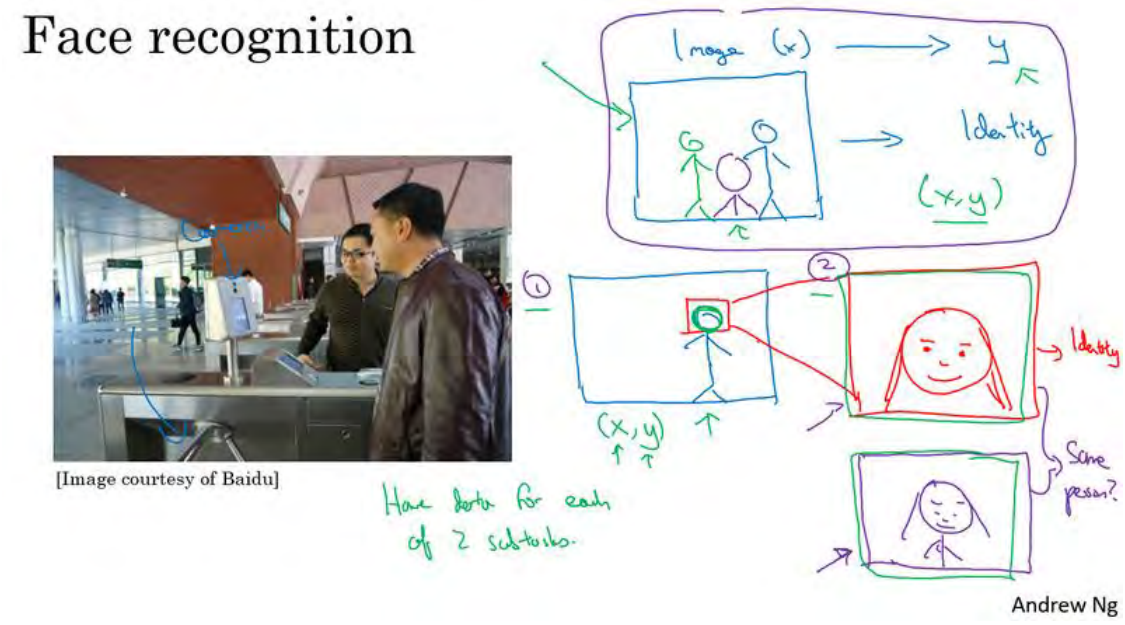


相反，迄今为止最好的方法似乎是一个多步方法，首先，你运行一个软件来检测人脸，所以第一个检测器找的是人脸位置，检测到人脸，然后放大图像的那部分，并裁剪图像，使人脸居中显示，然后就是这里红线框起来的照片，再喂到神经网络里，让网络去学习，或估计那人的身份。



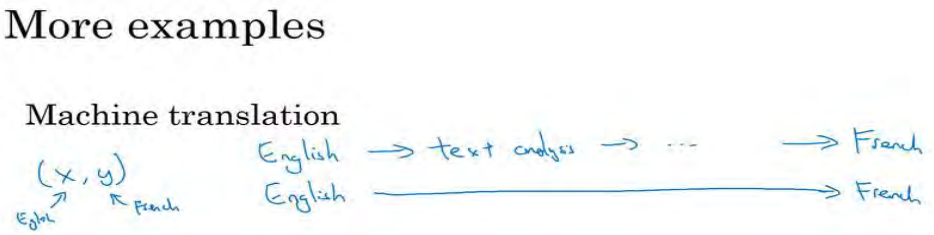
研究人员发现，比起一步到位，一步学习，把这个问题分解成两个更简单的步骤。首先，是弄清楚脸在哪里。第二步是看着脸，弄清楚这是谁。这第二种方法让学习算法，或者说两个学习算法分别解决两个更简单的任务，并在整体上得到更好的表现。

顺便说一句，如果你想知道第二步实际是怎么工作的，我这里其实省略了很多。训练第二步的方式，训练网络的方式就是输入两张图片，然后你的网络做的就是将输入的两张图比较一下，判断是否是同一个人。比如你记录了 10,000 个员工 **ID**，你可以把红色框起来的图像快速比较……也许是全部 10,000 个员工记录在案的 **ID**，看看这张红线内的照片，是不是那 10000 个员工之一，来判断是否应该允许其进入这个设施或者进入这个办公楼。这是一个门禁系统，允许员工进入工作场所的门禁。

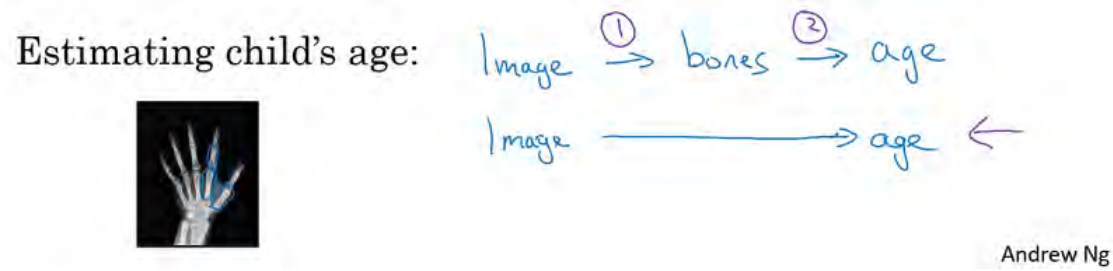


为什么两步法更好呢？实际上有两个原因。一是，你解决的两个问题，每个问题实际上要简单得多。但第二，两个子任务的训练数据都很多。具体来说，有很多数据可以用于人脸识别训练，对于这里的任务 1 来说，任务就是观察一张图，找出人脸所在的位置，把人脸图像框出来，所以有很多数据，有很多标签数据(x, y)，其中x是图片，y是表示人脸的位置，你可以建立一个神经网络，可以很好地处理任务 1。然后任务 2，也有很多数据可用，今天，业界领先的公司拥有，比如说数百万张人脸照片，所以输入一张裁剪得很紧凑的照片，比如这张红色照片，下面这个，今天业界领先的人脸识别团队有至少数亿的图像，他们可以用来观察两张图片，并试图判断照片里人的身份，确定是否同一个人，所以任务 2 还有很多数据。相比之下，如果你想一步到位，这样(x, y)的数据对就少得多，其中 x 是门禁系统拍摄的图像，y是那人的身份，因为你没有足够多的数据去解决这个端到端学习问题，但你却有足够多的数据来解决子问题 1 和子问题 2。

实际上，把这个分成两个子问题，比纯粹的端到端深度学习方法，达到更好的表现。不过如果你有足够多的数据来做端到端学习，也许端到端方法效果更好。但在今天的实践中，并不是最好的方法。



我们再来看几个例子，比如机器翻译。传统上，机器翻译系统也有一个很复杂的流水线，比如英语机翻得到文本，然后做文本分析，基本上要从文本中提取一些特征之类的，经过很多步骤，你最后会将英文文本翻译成法文。因为对于机器翻译来说的确有很多(英文,法文)的数据对，端到端深度学习在机器翻译领域非常好用，那是因为在今天可以收集$x - y$对的大数据集，就是英文句子和对应的法语翻译。所以在这个例子中，端到端深度学习效果很好。



最后一个例子，比如说你希望观察一个孩子手部的 X 光照片，并估计一个孩子的年龄。你知道，当我第一次听到这个问题的时候，我以为这是一个非常酷的犯罪现场调查任务，你

可能悲剧的发现了一个孩子的骨架，你想弄清楚孩子在生时是怎么样的。事实证明，这个问题的典型应用，从 X 射线图估计孩子的年龄，是我想太多了，没有我想象的犯罪现场调查脑洞那么大，结果这是儿科医生用来判断一个孩子的发育是否正常。

处理这个例子的一个非端到端方法，就是照一张图，然后分割出每一块骨头，所以就是分辨出那段骨头应该在哪里，那段骨头在哪里，那段骨头在哪里，等等。然后，知道不同骨骼的长度，你可以去查表，查到儿童手中骨头的平均长度，然后用它来估计孩子的年龄，所以这种方法实际上很好。

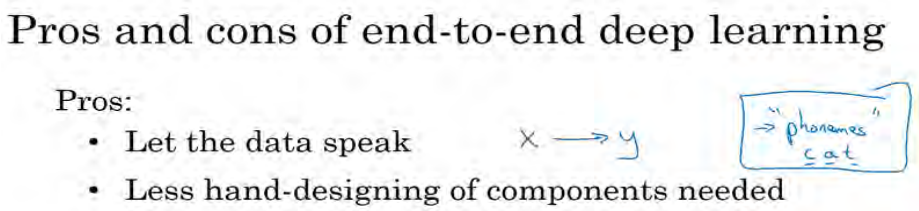
相比之下，如果你直接从图像去判断孩子的年龄，那么你需要大量的数据去直接训练。据我所知，这种做法今天还是不行的，因为没有足够的数据来用端到端的方式来训练这个任务。

你可以想象一下如何将这个问题分解成两个步骤，第一步是一个比较简单的问题，也许你不需要那么多数据，也许你不需要许多 X 射线图像来切分骨骼。而任务二，收集儿童手部的骨头长度的统计数据，你不需要太多数据也能做出相当准确的估计，所以这个多步方法看起来很有希望，也许比端对端方法更有希望，至少直到你能获得更多端到端学习的数据之前。

所以端到端深度学习系统是可行的，它表现可以很好，也可以简化系统架构，让你不需要搭建那么多手工设计的单独组件，但它也不是灵丹妙药，并不是每次都能成功。在下一个视频中，我想与你分享一个更系统的描述，什么时候你应该使用或者不应该使用端到端的深度学习，以及如何组装这些复杂的机器学习系统。

2.10 是否要使用端到端的深度学习？（Whether to use end-to-end learning?）

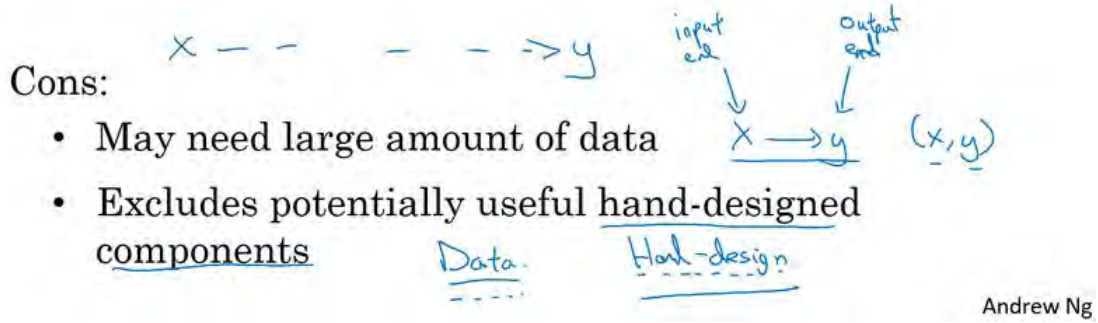
假设你正在搭建一个机器学习系统，你要决定是否使用端对端方法，我们来看看端到端深度学习的一些优缺点，这样你就可以根据一些准则，判断你的应用程序是否有希望使用端到端方法。



这里是应用端到端学习的一些好处，首先端到端学习真的只是让数据说话。所以如果你有足够多的(x, y)数据，那么不管从 x 到 y 最适合的函数映射是什么，如果你训练一个足够大的神经网络，希望这个神经网络能自己搞清楚，而使用纯机器学习方法，直接从x到y输入去训练的神经网络，可能更能够捕获数据中的任何统计信息，而不是被迫引入人类的成见。

例如，在语音识别领域，早期的识别系统有这个音位概念，就是基本的声音单元，如 cat 单词的“cat”的 Cu-、Ah-和 Tu-，我觉得这个音位是人类语言学家生造出来的，我实际上认为音位其实是语音学家的幻想，用音位描述语言也还算合理。但是不要强迫你的学习算法以音位为单位思考，这点有时没那么明显。如果你让你的学习算法学习它想学习的任意表示方式，而不是强迫你的学习算法使用音位作为表示方式，那么其整体表现可能会更好。

端到端深度学习的第二个好处就是这样，所需手工设计的组件更少，所以这也许能够简化你的设计工作流程，你不需要花太多时间去手工设计功能，手工设计这些中间表示方式。

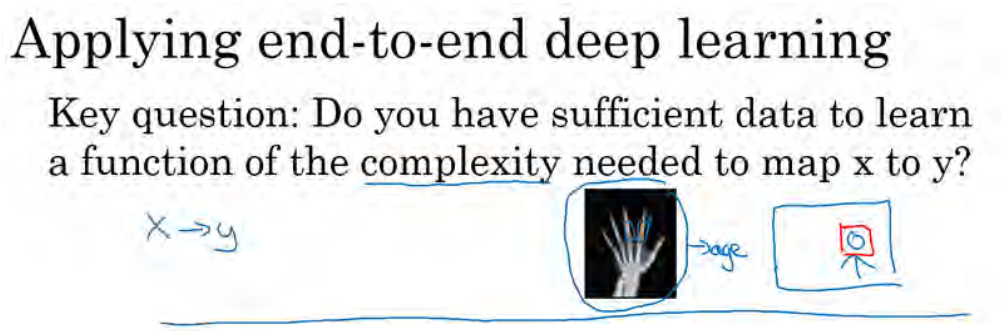


那么缺点呢？这里有一些缺点，首先，它可能需要大量的数据。要直接学到这个x到y的映射，你可能需要大量(x, y)数据。我们在以前的视频里看过一个例子，其中你可以收集大量子任务数据，比如人脸识别，我们可以收集很多数据用来分辨图像中的人脸，当你找到一

张脸后，也可以找得到很多人脸识别数据。但是对于整个端到端任务，可能只有更少的数据可用。所以x这是端到端学习的输入端，y是输出端，所以你需要很多这样的(x, y)数据，在输入端和输出端都有数据，这样可以训练这些系统。这就是为什么我们称之为端到端学习，因为你直接学习出从系统的一端到系统的另一端。

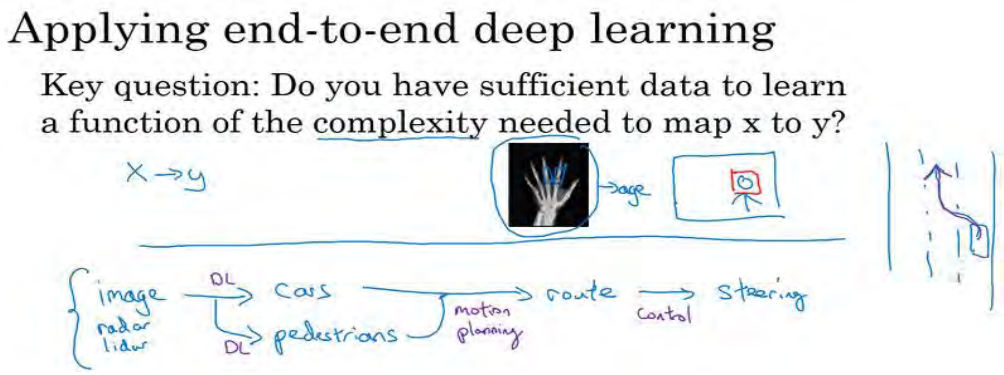
另一个缺点是，它排除了可能有用的手工设计组件。机器学习研究人员一般都很鄙视手工设计的东西，但如果你没有很多数据，你的学习算法就没办法从很小的训练集数据中获得洞察力。所以手工设计组件在这种情况，可能是把人类知识直接注入算法的途径，这总不是一件坏事。我觉得学习算法有两个主要的知识来源，一个是数据，另一个是你手工设计的任何东西，可能是组件，功能，或者其他东西。所以当你有大量数据时，手工设计的东西就不太重要了，但是当你没有太多的数据时，构造一个精心设计的系统，实际上可以将人类对这个问题的很多认识直接注入到问题里，进入算法里应该挺有帮助的。

所以端到端深度学习的弊端之一是它把可能有用的人工设计的组件排除在外了，精心设计的人工组件可能非常有用，但它们也有可能真的伤害到你的算法表现。例如，强制你的算法以音位为单位思考，也许让算法自己找到更好的表示方法更好。所以这是一把双刃剑，可能有坏处，可能有好处，但往往好处更多，手工设计的组件往往在训练集更小的时候帮助更大。



如果你在构建一个新的机器学习系统，而你在尝试决定是否使用端到端深度学习，我认为关键的问题是，你有足够的数据能够直接学到从x映射到y足够复杂的函数吗？我还没有正式定义过这个词“必要复杂度（**complexity needed**）”。但直觉上，如果你想从x到y的数据学习出一个函数，就是看着这样的图像识别出图像中所有骨头的位置，那么也许这像是识别图中骨头这样相对简单的问题，也许系统不需要那么多数据来学会处理这个任务。或给出一张人物照片，也许在图中把人脸找出来不是什么难事，所以你也许不需要太多数据去找到人脸，或者至少你可以找到足够数据去解决这个问题。相对来说，把手的 X 射线照片直接映射到孩子的年龄，直接去找这种函数，直觉上似乎是更为复杂的问题。如果你用纯端到端方法，

需要很多数据去学习。



视频最后我讲一个更复杂的例子，你可能知道我一直在花时间帮忙主攻无人驾驶技术的公司 drive.ai，无人驾驶技术的发展其实让我相当激动，你怎么造出一辆自己能行驶的车呢？好，这里你可以做一件事，这不是端到端的深度学习方法，你可以把你车前方的雷达、激光雷达或者其他传感器的读数看成是输入图像。但是为了说明起来简单，我们就说拍一张车前方或者周围的照片，然后驾驶要安全的话，你必须能检测到附近的车，你也需要检测到行人，你需要检测其他的东西，当然，我们这里提供的是高度简化的例子。

弄清楚其他车和形如的位置之后，你就需要计划你自己的路线。所以换句话说，当你看到其他车子在哪，行人在哪里，你需要决定如何摆方向盘在接下来的几秒钟内引导车子的路径。如果你决定了要走特定的路径，也许这是道路的俯视图，这是你的车，也许你决定了要走那条路线，这是一条路线，那么你就需要摆动你的方向盘到合适的角度，还要发出合适的加速和制动指令。所以从传感器或图像输入到检测行人和车辆，深度学习可以做得很好，但一旦知道其他车辆和行人的位置或者动向，选择一条车要走的路，这通常用的不是深度学习，而是用所谓的运动规划软件完成的。如果你学过机器人课程，你一定知道运动规划，然后决定了你的车子要走的路径之后。还会有一些其他算法，我们说这是一个控制算法，可以产生精确的决策确定方向盘应该精确地转多少度，油门或刹车上应该用多少力。

所以这个例子就表明了，如果你想使用机器学习或者深度学习来学习某些单独的组件，那么当你应用监督学习时，你应该仔细选择要学习的x到y映射类型，这取决于那些任务你可以收集数据。相比之下，谈论纯端到端深度学习方法是很激动人心的，你输入图像，直接得出方向盘转角，但是就目前能收集到的数据而言，还有我们今天能够用神经网络学习的数据

类型而言，这实际上不是最有希望的方法，或者说这个方法并不是团队想出的最好用的方法。而我认为这种纯粹的端到端深度学习方法，其实前景不如这样更复杂的多步方法。因为目前能收集到的数据，还有我们现在训练神经网络的能力是有局限的。

这就是端到端的深度学习，有时候效果拔群。但你也要注意应该在什么时候使用端到端深度学习。最后，谢谢你，恭喜你坚持到现在，如果你学完了上周的视频和本周的视频，那么我认为你已经变得更聪明，更具战略性，并能够做出更好的优先分配任务的决策，更好地推动你的机器学习项目，也许比很多机器学习工程师，还有和我在硅谷看到的研究人员都强。所以恭喜你学到这里，我希望你能看看本周的作业，应该能再给你一个机会去实践这些理念，并确保你掌握它们。

第四门课 卷积神经网络（Convolutional Neural Networks）

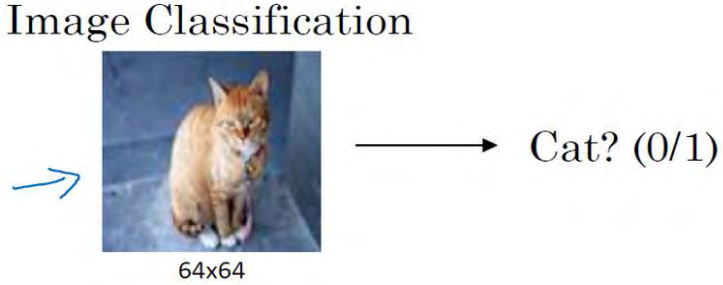
第一周 卷积神经网络（Foundations of Convolutional Neural Networks）

1.1 计算机视觉（Computer vision）

欢迎参加这次的卷积神经网络课程，计算机视觉是一个飞速发展的一个领域，这多亏了深度学习。深度学习与计算机视觉可以帮助汽车，查明周围的行人和汽车，并帮助汽车避开它们。还使得人脸识别技术变得更加效率和精准，你们即将能够体验到或早已体验过仅仅通过刷脸就能解锁手机或者门锁。当你解锁了手机，我猜手机上一定有很多分享图片的应用。在上面，你能看到美食，酒店或美丽风景的图片。有些公司在这些应用上使用了深度学习技术来向你展示最为生动美丽以及与你最为相关的图片。机器学习甚至还催生了新的艺术类型。深度学习之所以让我兴奋有下面两个原因，我想你们也是这么想的。

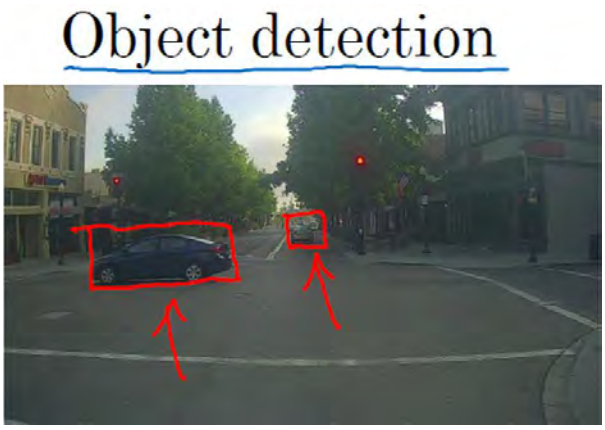
第一，计算机视觉的高速发展标志着新型应用产生的可能，这是几年前，人们所不敢想象的。通过学习使用这些工具，你也许能够创造出新的产品和应用。

其次，即使到头来你未能在计算机视觉上有所建树，但我发现，人们对于计算机视觉的研究是如此富有想象力和创造力，由此衍生出新的神经网络结构与算法，这实际上启发人们去创造出计算机视觉与其他领域的交叉成果。举个例子，之前我在做语音识别的时候，我经常从计算机视觉领域中寻找灵感， 并将其应用于我的文献当中。所以即使你在计算机视觉方面没有做出成果，我也希望你也可以将所学的知识应用到其他算法和结构。就介绍到这儿，让我们开始学习吧。

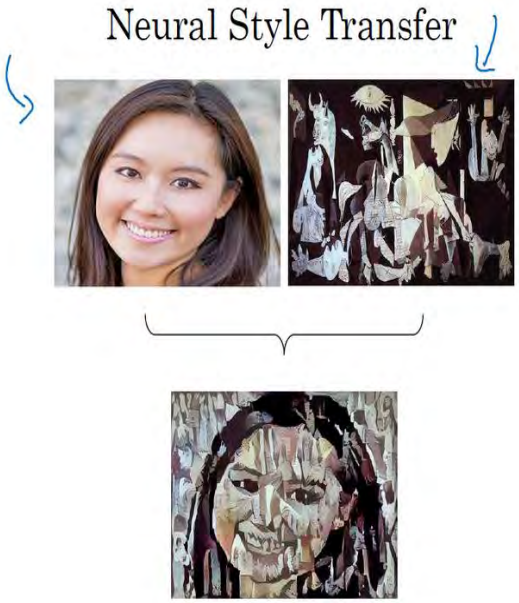


这是我们本节课将要学习的一些问题，你应该早就听说过图片分类，或者说图片识别。

比如给出这张 64×64 的图片，让计算机去分辨出这是一只猫。



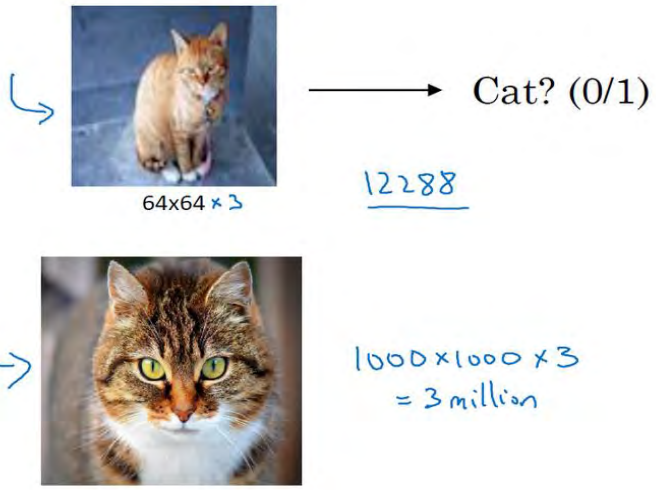
还有一个例子，在计算机视觉中有个问题叫做目标检测，比如在一个无人驾驶项目中，你不一定非得识别出图片中的物体是车辆，但你需要计算出其他车辆的位置，以确保自己能够避开它们。所以在目标检测项目中，首先需要计算出图中有哪些物体，比如汽车，还有图片中的其他东西，再将它们模拟成一个个盒子，或用一些其他的技术识别出它们在图片中的位置。注意在这个例子中，在一张图片中同时有多个车辆，每辆车相对与你来说都有一个确切的距离。



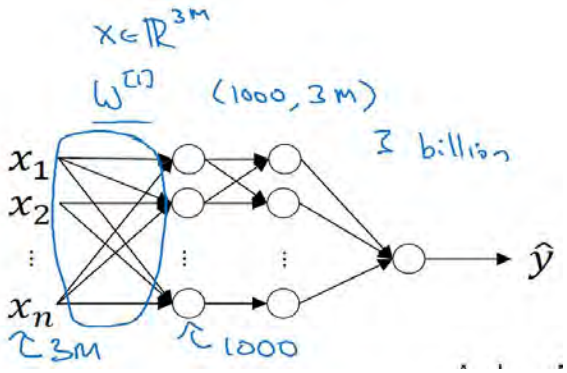
还有一个更有趣的例子，就是神经网络实现的图片风格迁移，比如说你有一张图片，但你想将这张图片转换为另外一种风格。所以图片风格迁移，就是你有一张满意的图片和一张风格图片，实际上右边这幅画是毕加索的画作，而你可以利用神经网络将它们融合到一起，描绘出一张新的图片。它的整体轮廓来自于左边，却是右边的风格，最后生成下面这张图片。这种神奇的算法创造出了新的艺术风格，所以在这门课程中，你也能通过学习做到这样的事

情。

但在应用计算机视觉时要面临一个挑战，就是数据的输入可能会非常大。举个例子，在过去的课程中，你们一般操作的都是 64×64 的小图片，实际上，它的数据量是 64×64×3，因为每张图片都有 3 个颜色通道。如果计算一下的话，可得知数据量为 12288，所以我们的特征向量x维度为 12288。这其实还好，因为 64×64 真的是很小的一张图片。



如果你要操作更大的图片，比如一张 1000×1000 的图片，它足有 1 兆那么大，但是特征向量的维度达到了 1000×1000×3，因为有 3 个 **RGB** 通道，所以数字将会是 300 万。如果你在尺寸很小的屏幕上观察，可能察觉不出上面的图片只有 64×64 那么大，而下面一张是 1000×1000 的大图。



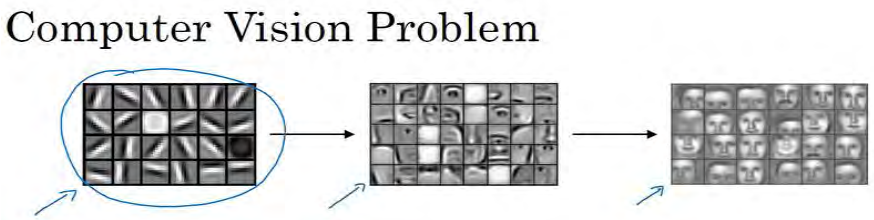
如果你要输入 300 万的数据量，这就意味着，特征向量x的维度高达 300 万。所以在第一隐藏层中，你也许会有 1000 个隐藏单元，而所有的权值组成了矩阵 $W^{[1]}$。如果你使用了标准的全连接网络，就像我们在第一门和第二门的课程里说的，这个矩阵的大小将会是 1000×300 万。因为现在x的维度为$3m$，$3m$通常用来表示 300 万。这意味着矩阵$W^{[1]}$会有 30 亿个参数，这是个非常巨大的数字。在参数如此大量的情况下，难以获得足够的数据来防止神经网络发生过拟合和竞争需求，要处理包含 30 亿参数的神经网络，巨大的内存需求让人

不太能接受。

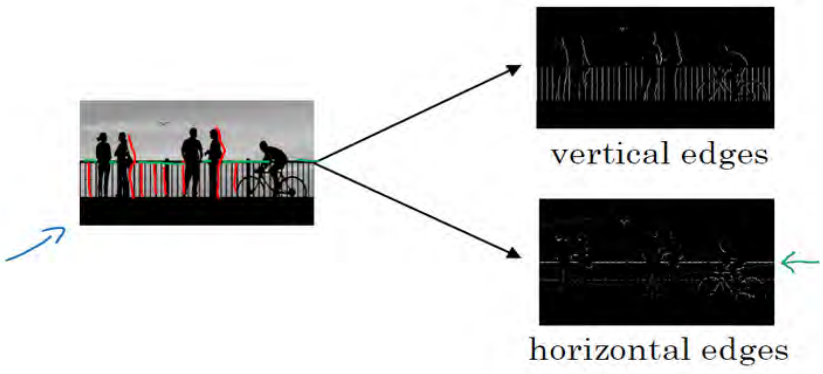
但对于计算机视觉应用来说，你肯定不想它只处理小图片，你希望它同时也要能处理大图。为此，你需要进行卷积计算，它是卷积神经网络中非常重要的一块。下节课中，我会为你介绍如何进行这种运算，我将用边缘检测的例子来向你说明卷积的含义。

1.2 边缘检测示例（Edge detection example）

卷积运算是卷积神经网络最基本的组成部分，使用边缘检测作为入门样例。在这个视频中，你会看到卷积是如何进行运算的。

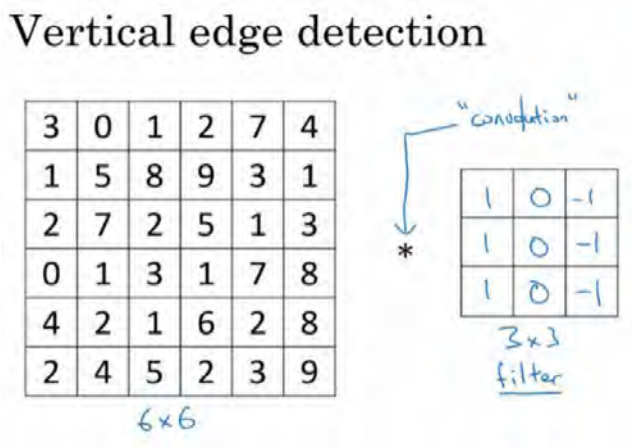


在之前的视频中，我说过神经网络的前几层是如何检测边缘的，然后，后面的层有可能检测到物体的部分区域，更靠后的一些层可能检测到完整的物体，这个例子中就是人脸。在这个视频中，你会看到如何在一张图片中进行边缘检测。

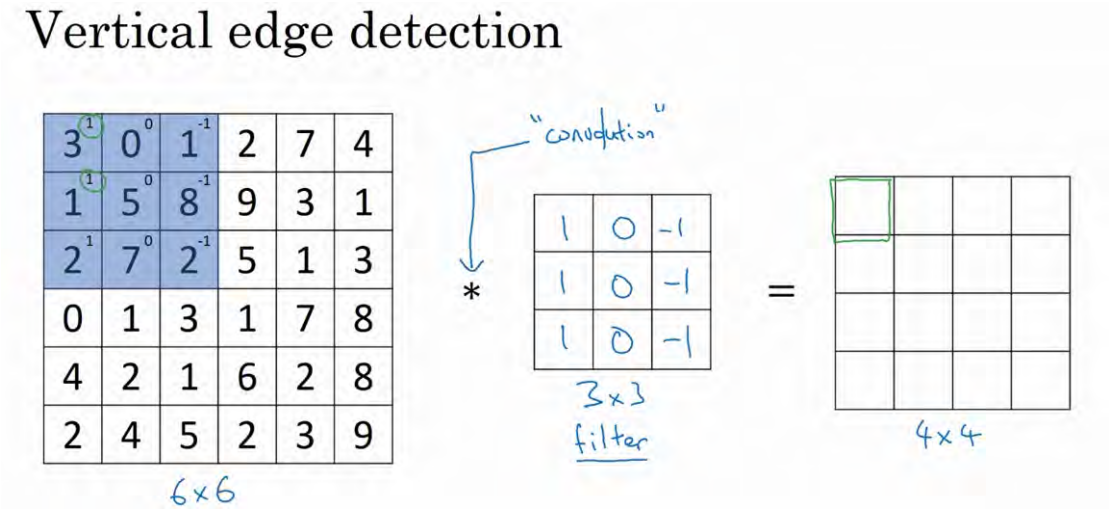


让我们举个例子，给了这样一张图片，让电脑去搞清楚这张照片里有什么物体，你可能做的第一件事是检测图片中的垂直边缘。比如说，在这张图片中的栏杆就对应垂直线，与此同时，这些行人的轮廓线某种程度上也是垂线，这些线是垂直边缘检测器的输出。同样，你可能也想检测水平边缘，比如说这些栏杆就是很明显的水平线，它们也能被检测到，结果在这。所以如何在图像中检测这些边缘？

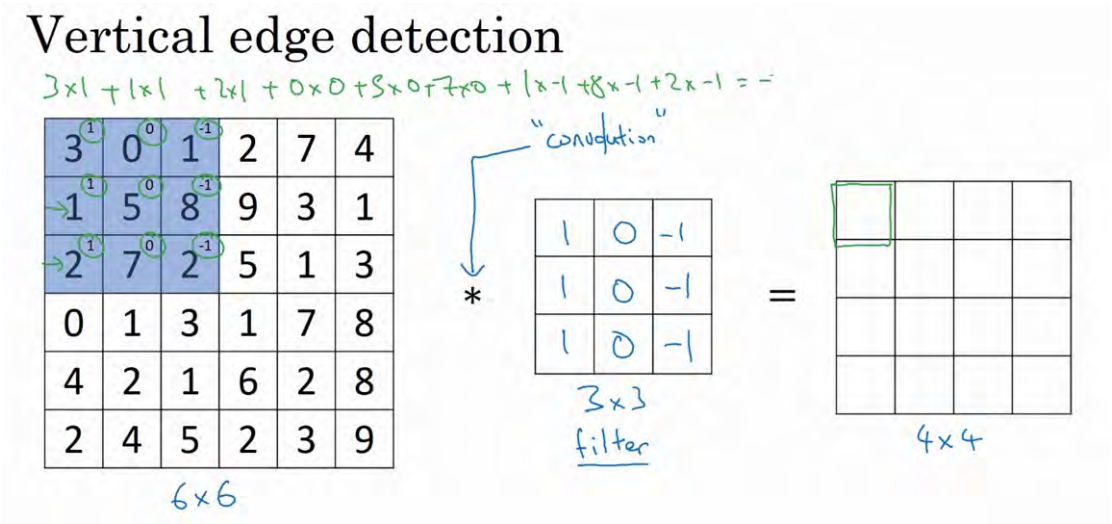
看一个例子，这是一个 6×6 的灰度图像。因为是灰度图像，所以它是 6×6×1 的矩阵，而不是 6×6×3 的，因为没有 **RGB** 三通道。为了检测图像中的垂直边缘，你可以构造一个 3×3 矩阵。在共用习惯中，在卷积神经网络的术语中，它被称为过滤器。我要构造一个 3×3 的过滤器，像这样$\begin{bmatrix}1 & 0 & -1\\1 & 0 & -1\\1 & 0 & -1\end{bmatrix}$。在论文它有时候会被称为核，而不是过滤器，但在这个视频中，我将使用过滤器这个术语。对这个 6×6 的图像进行卷积运算，卷积运算用“∗”来表示，用 3×3 的过滤器对其进行卷积。



关于符号表示，有一些问题，在数学中“∗”就是卷积的标准标志，但是在 **Python** 中，这个标识常常被用来表示乘法或者元素乘法。所以这个“∗”有多层含义，它是一个重载符号，在这个视频中，当“∗”表示卷积的时候我会特别说明。

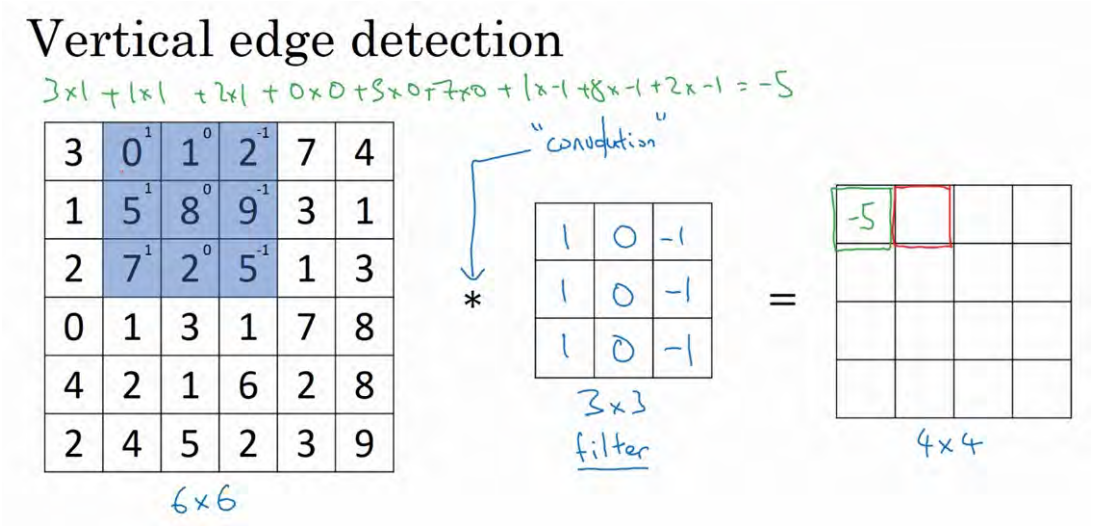


这个卷积运算的输出将会是一个 4×4 的矩阵，你可以将它看成一个 4×4 的图像。下面来说明是如何计算得到这个 4×4 矩阵的。为了计算第一个元素，在 4×4 左上角的那个元素，使用 3×3 的过滤器，将其覆盖在输入图像，如下图所示。然后进行元素乘法（**element-wise products**）运算，所以$\begin{bmatrix} 3\times1 & 0\times0 & 1\times(1) \\ 1\times1 & 5\times0 & 8\times(-1) \\ 2\times1 & 7\times0 & 2\times(-1) \end{bmatrix}=\begin{bmatrix} 3 & 0 & -1 \\ 1 & 0 & -8 \\ 2 & 0 & -2 \end{bmatrix}$，然后将该矩阵每个元素相加得到最左上角的元素，即$3+1+2+0+0+0+(-1)+(-8)+(-2)=-5$。

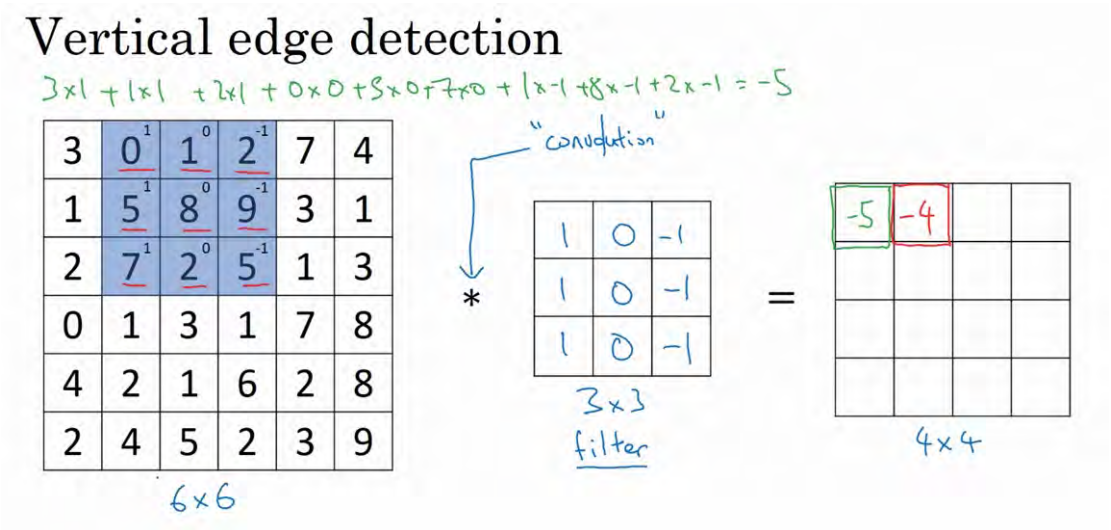


把这 9 个数加起来得到-5，当然，你可以把这 9 个数按任何顺序相加，我只是先写了第一列，然后第二列，第三列。

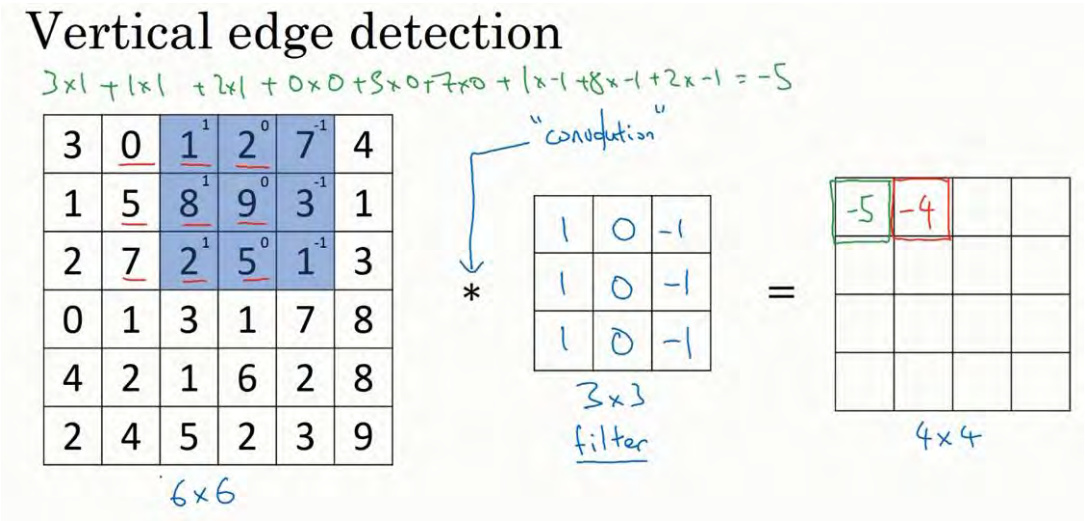
接下来，为了弄明白第二个元素是什么，你要把蓝色的方块，向右移动一步，像这样，把这些绿色的标记去掉：



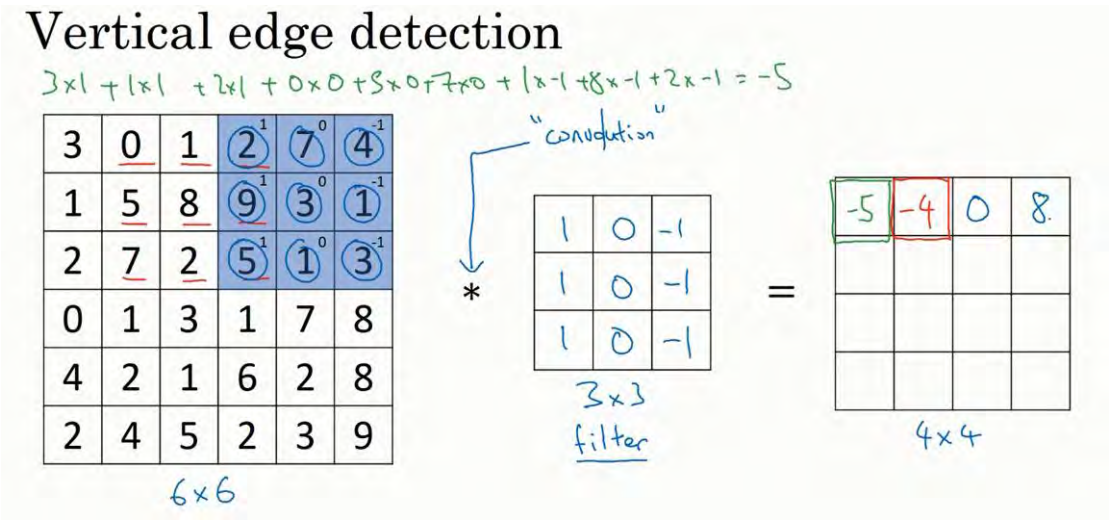
继续做同样的元素乘法，然后加起来，所以是 $0\times1+5\times1+7\times1+1\times0+8\times0+2\times0+2\times(-1)+9\times(-1)+5\times(-1)=-4$。



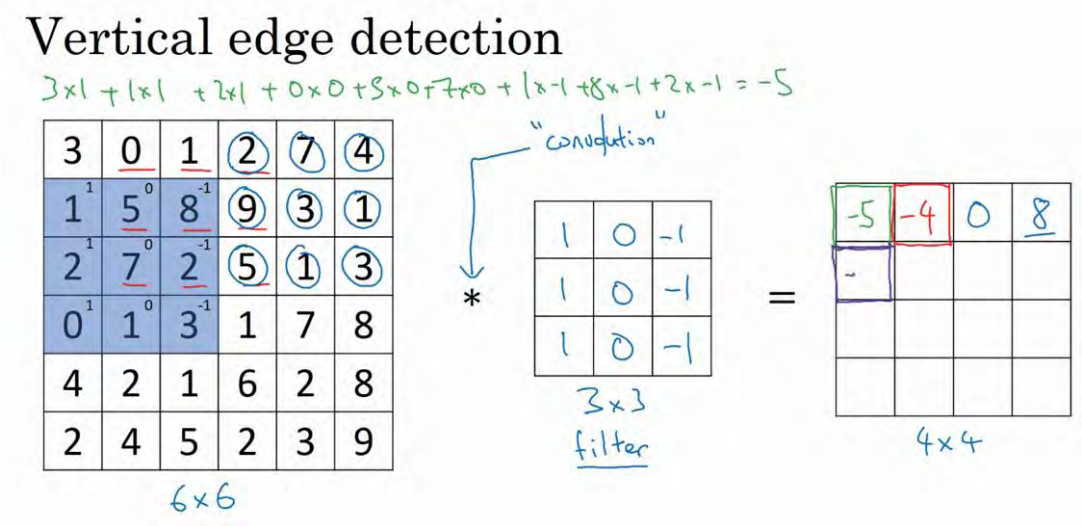
接下来也是一样，继续右移一步，把 9 个数的点积加起来得到 0。



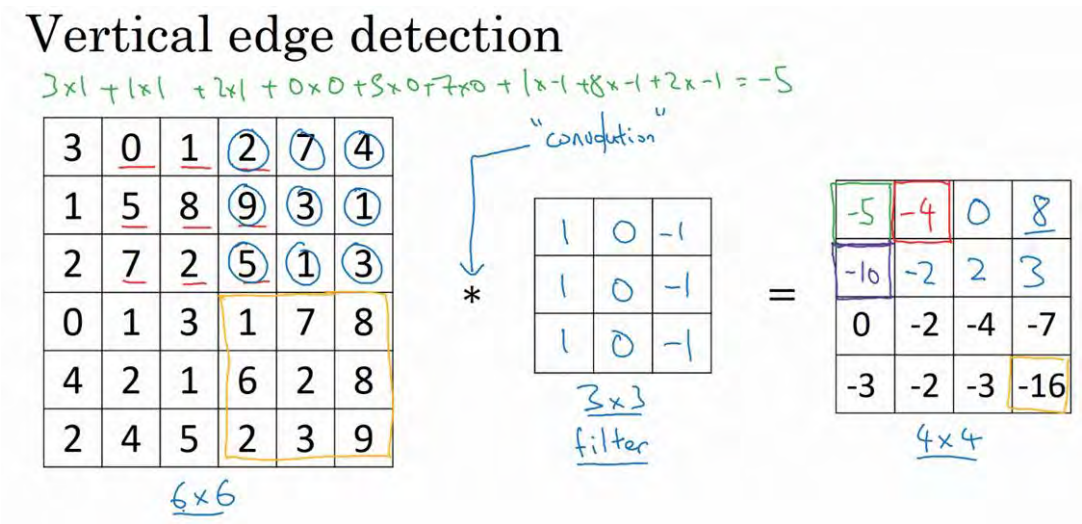
继续移得到 8，验证一下：$2\times1+9\times1+5\times1+7\times0+3\times0+1\times0+4\times(-1)+1\times(-1)+3\times(-1)=8$。



接下来为了得到下一行的元素，现在把蓝色块下移，现在蓝色块在这个位置：



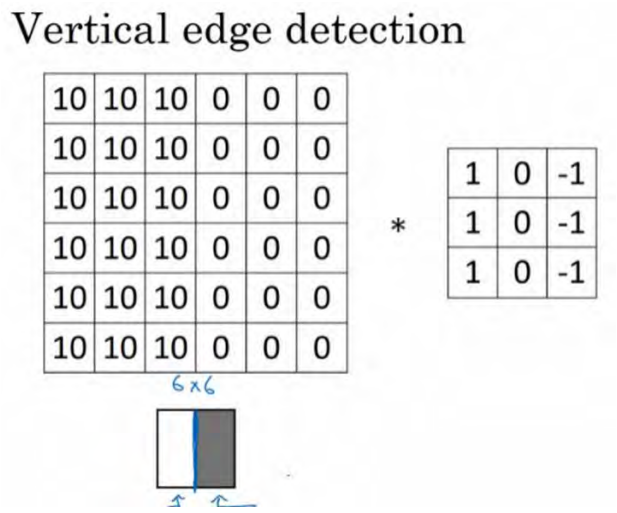
重复进行元素乘法，然后加起来。通过这样做得到-10。再将其右移得到-2，接着是 2，3。以此类推，这样计算完矩阵中的其他元素。



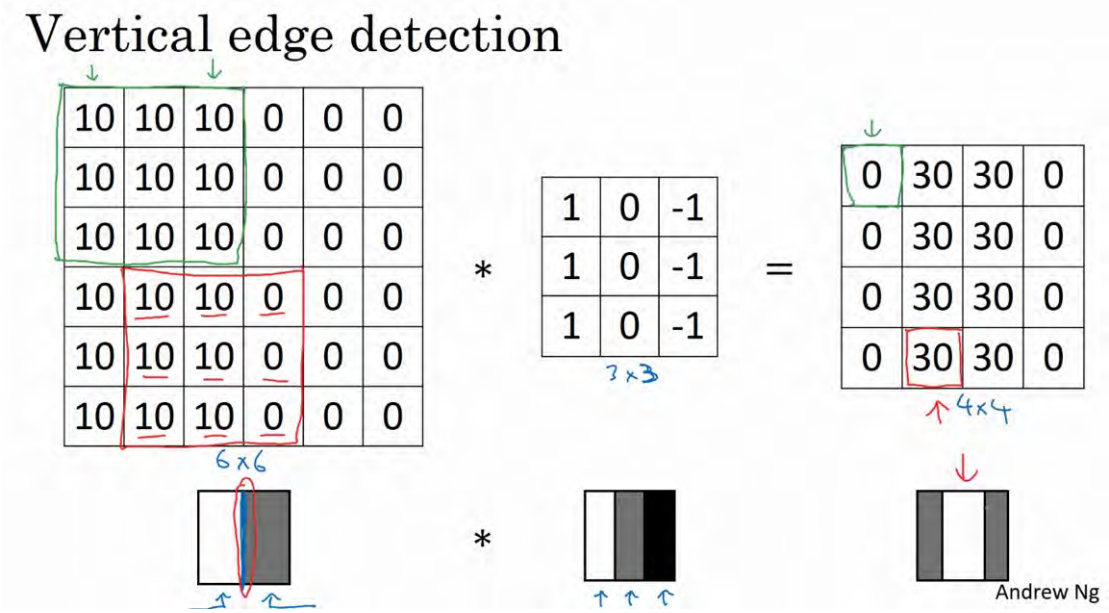
为了说得更清楚一点，这个-16 是通过底部右下角的 3×3 区域得到的。

因此 6×6 矩阵和 3×3 矩阵进行卷积运算得到 4×4 矩阵。这些图片和过滤器是不同维度的矩阵，但左边矩阵容易被理解为一张图片，中间的这个被理解为过滤器，右边的图片我们可以理解为另一张图片。这个就是垂直边缘检测器，下一页中你就会明白。

在往下讲之前，多说一句，如果你要使用编程语言实现这个运算，不同的编程语言有不同的函数，而不是用“*”来表示卷积。所以在编程练习中，你会使用一个叫 **conv_forward** 的函数。如果在 **tensorflow** 下，这个函数叫 **tf.conv2d**。在其他深度学习框架中，在后面的课程中，你将会看到 **Keras** 这个框架，在这个框架下用 **Conv2D** 实现卷积运算。所有的编程框架都有一些函数来实现卷积运算。

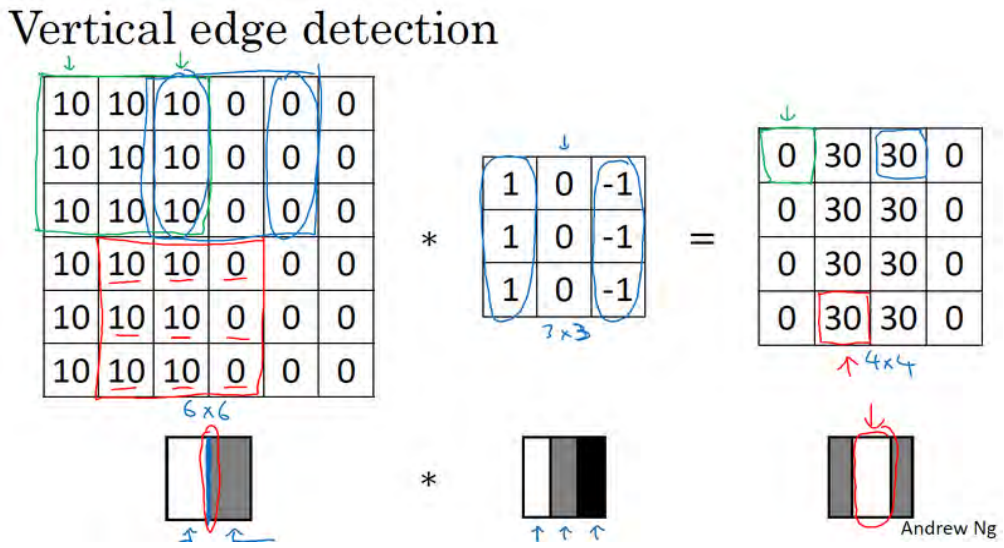


为什么这个可以做垂直边缘检测呢？让我们来看另外一个例子。为了讲清楚，我会用一个简单的例子。这是一个简单的 6×6 图像，左边的一半是 10，右边一般是 0。如果你把它当成一个图片，左边那部分看起来是白色的，像素值 10 是比较亮的像素值，右边像素值比较暗，我使用灰色来表示 0，尽管它也可以被画成黑的。图片里，有一个特别明显的垂直边缘在图像中间，这条垂直线是从黑到白的过渡线，或者从白色到深色。



所以，当你用一个 3×3 过滤器进行卷积运算的时候，这个 3×3 的过滤器可视化为下面这个样子，在左边有明亮的像素，然后有一个过渡，0 在中间，然后右边是深色的。卷积运算后，你得到的是右边的矩阵。如果你愿意，可以通过数学运算去验证。举例来说，最左上角的元素 0，就是由这个 3×3 块（绿色方框标记）经过元素乘积运算再求和得到的，$10 \times 1 + 10 \times 1 + 10 \times 1 + 10 \times 0 + 10 \times 0 + 10 \times 0 + 10 \times (-1) + 10 \times (-1) + 10 \times (-1) = 0$。相反这个 30 是由这个（红色方框标记）得到的，

$10 \times 1 + 10 \times 1 + 10 \times 1 + 10 \times 0 + 10 \times 0 + 10 \times 0 + 0 \times (-1) + 0 \times (-1) + 0 \times (-1) = 30$。

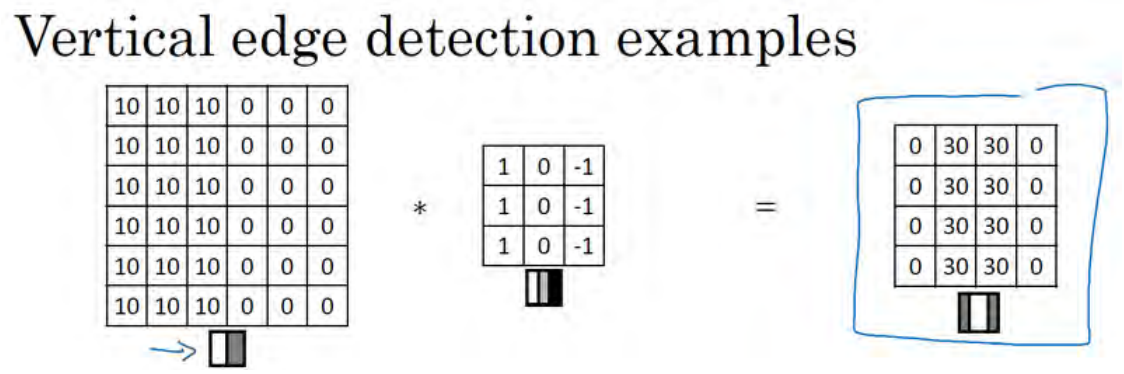


如果把最右边的矩阵当成图像，它是这个样子。在中间有段亮一点的区域，对应检查到这个 6×6 图像中间的垂直边缘。这里的维数似乎有点不正确，检测到的边缘太粗了。因为在这个例子中，图片太小了。如果你用一个 1000×1000 的图像，而不是 6×6 的图片，你会发现其会很好地检测出图像中的垂直边缘。在这个例子中，在输出图像中间的亮处，表示在图像中间有一个特别明显的垂直边缘。从垂直边缘检测中可以得到的启发是，因为我们使用 3×3 的矩阵（过滤器），所以垂直边缘是一个 3×3 的区域，左边是明亮的像素，中间的并不需要考虑，右边是深色像素。在这个 6×6 图像的中间部分，明亮的像素在左边，深色的像素在右边，就被视为一个垂直边缘，卷积运算提供了一个方便的方法来发现图像中的垂直边缘。

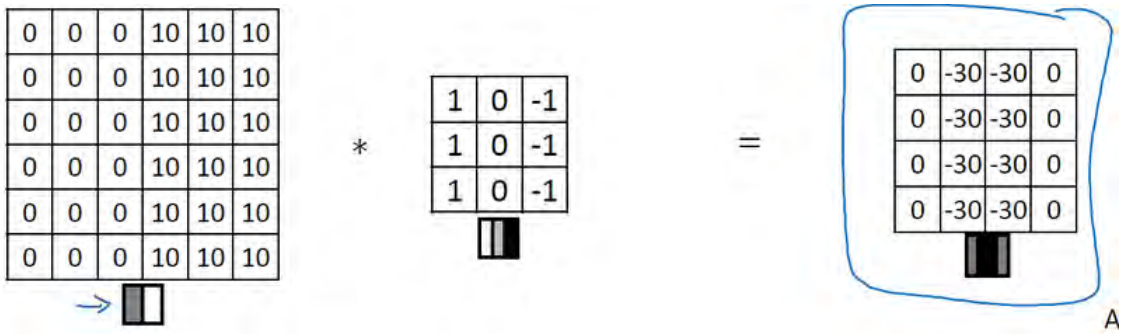
所以你已经了解卷积是怎么工作的，在下一个视频中，你将会看到如何使用卷积运算作为卷积神经网络的基本模块的。

1.3 更多边缘检测内容（More edge detection）

你已经见识到用卷积运算实现垂直边缘检测，在本视频中，你将学习如何区分正边和负边，这实际就是由亮到暗与由暗到亮的区别，也就是边缘的过渡。你还能了解到其他类型的边缘检测以及如何去实现这些算法，而不要总想着去自己编写一个边缘检测程序，让我们开始吧。



还是上一个视频中的例子，这张 6×6 的图片，左边较亮，而右边较暗，将它与垂直边缘检测滤波器进行卷积，检测结果就显示在了右边这幅图的中间部分。

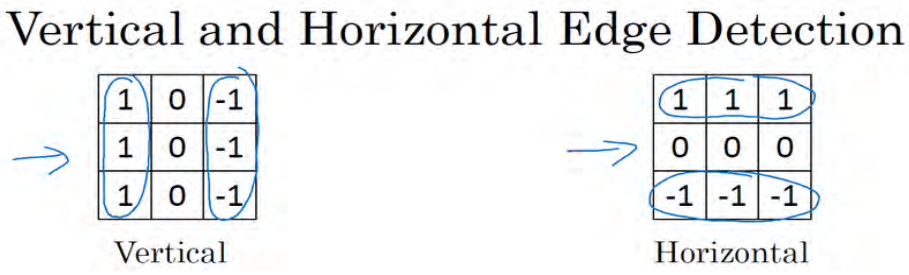


现在这幅图有什么变化呢？它的颜色被翻转了，变成了左边比较暗，而右边比较亮。现在亮度为 10 的点跑到了右边，为 0 的点则跑到了左边。如果你用它与相同的过滤器进行卷积，最后得到的图中间会是-30，而不是 30。如果你将矩阵转换为图片，就会是该矩阵下面图片的样子。现在中间的过渡部分被翻转了，之前的 30 翻转成了-30，表明是由暗向亮过渡，而不是由亮向暗过渡。

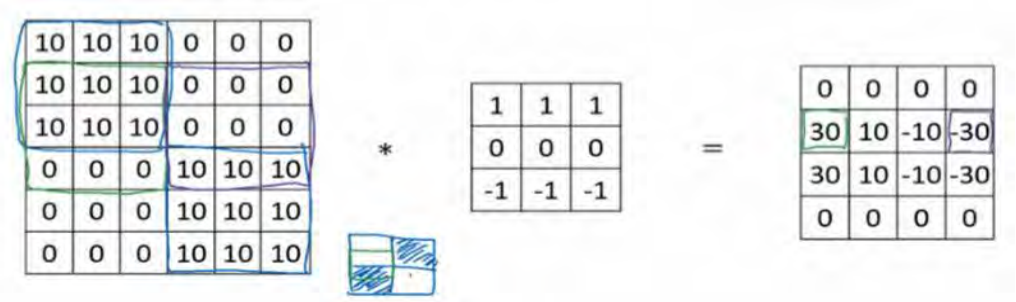
如果你不在乎这两者的区别，你可以取出矩阵的绝对值。但这个特定的过滤器确实可以为我们区分这两种明暗变化的区别。

再来看看更多的边缘检测的例子，我们已经见过这个 3×3 的过滤器，它可以检测出垂直的边缘。所以，看到右边这个过滤器，我想你应该猜出来了，它能让你检测出水平的边缘。提醒一下，一个垂直边缘过滤器是一个 3×3 的区域，它的左边相对较亮，而右边相对较暗。

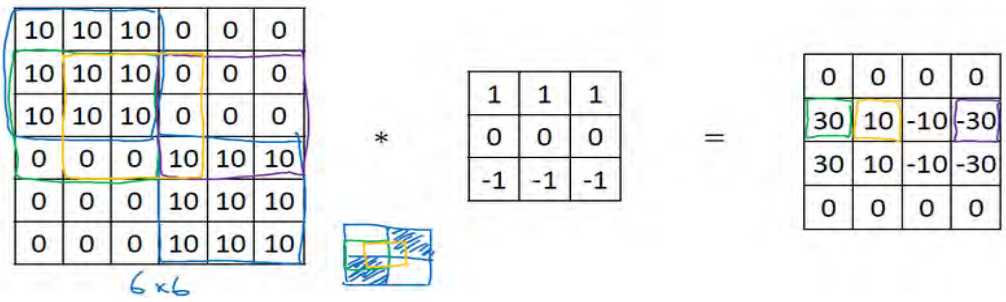
相似的，右边这个水平边缘过滤器也是一个 3×3 的区域，它的上边相对较亮，而下方相对较暗。



这里还有个更复杂的例子，左上方和右下方都是亮度为 10 的点。如果你将它绘成图片，右上角是比较暗的地方，这边都是亮度为 0 的点，我把这些比较暗的区域都加上阴影。而左上方和右下方都会相对较亮。如果你用这幅图与水平边缘过滤器卷积，就会得到右边这个矩阵。



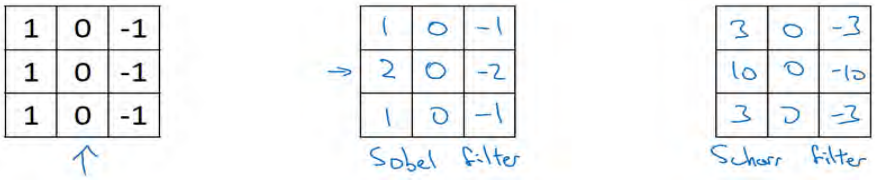
再举个例子，这里的 30（右边矩阵中绿色方框标记元素）代表了左边这块 3×3 的区域（左边矩阵绿色方框标记部分），这块区域确实是上边比较亮，而下边比较暗的，所以它在这里发现了一条正边缘。而这里的-30（右边矩阵中紫色方框标记元素）又代表了左边另一块区域（左边矩阵紫色方框标记部分），这块区域确实是底部比较亮，而上边则比较暗，所以在这里它是一条负边。



再次强调，我们现在所使用的都是相对很小的图片，仅有 6×6。但这些中间的数值，比如说这个 10（右边矩阵中黄色方框标记元素）代表的是左边这块区域（左边 6×6 矩阵中黄色方框标记的部分）。这块区域左边两列是正边，右边一列是负边，正边和负边的值加在一

起得到了一个中间值。但假如这个一个非常大的 1000×1000 的类似这样棋盘风格的大图，就不会出现这些亮度为 10 的过渡带了，因为图片尺寸很大，这些中间值就会变得非常小。

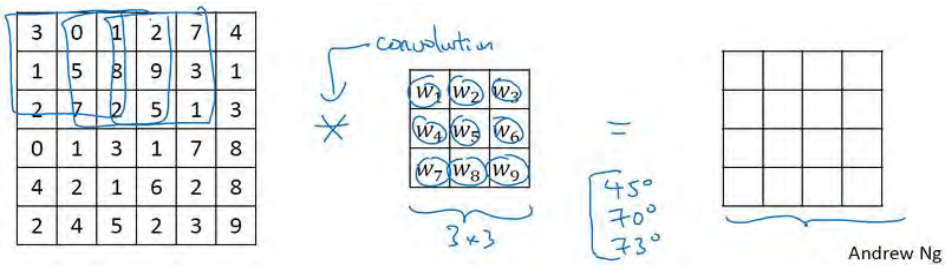
总而言之，通过使用不同的过滤器，你可以找出垂直的或是水平的边缘。但事实上，对于这个 3×3 的过滤器来说，我们使用了其中的一种数字组合。



但在历史上，在计算机视觉的文献中，曾公平地争论过怎样的数字组合才是最好的，所以你还可以使用这种：$\begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix}$，叫做 **Sobel** 的过滤器，它的优点在于增加了中间一行元素的权重，这使得结果的鲁棒性会更高一些。

但计算机视觉的研究者们也会经常使用其他的数字组合，比如这种：$\begin{bmatrix}3 & 0 & -3\\ 10 & 0 & -10\\ 3 & 0 & -3\end{bmatrix}$，这叫做 **Scharr** 过滤器，它有着和之前完全不同的特性，实际上也是一种垂直边缘检测，如果你将其翻转 90 度，你就能得到对应水平边缘检测。

随着深度学习的发展，我们学习的其中一件事就是当你真正想去检测出复杂图像的边缘，你不一定要去使用那些研究者们所选择的这九个数字，但你可以从中获益匪浅。把这矩阵中的 9 个数字当成 9 个参数，并且在之后你可以学习使用反向传播算法，其目标就是去理解这 9 个参数。



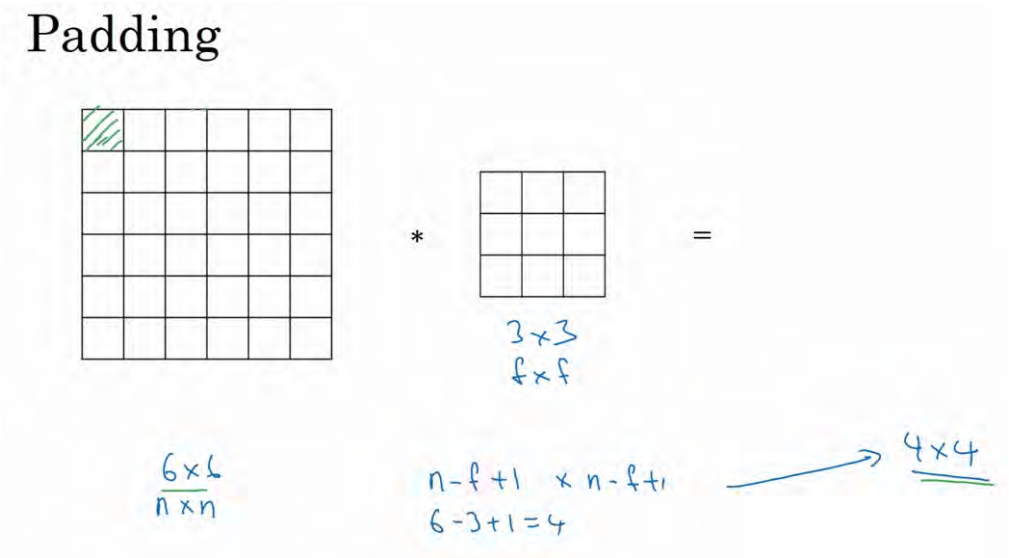
当你得到左边这个 6×6 的图片，将其与这个 3×3 的过滤器进行卷积，将会得到一个出色的边缘检测。这就是你在下节视频中将会看到的，把这 9 个数字当成参数的过滤器，通过反向传播，你可以学习这种$\begin{bmatrix}1 & 0 & -1\\ 1 & 0 & -1\\ 1 & 0 & -1\end{bmatrix}$的过滤器，或者 Sobel 过滤器和 Scharr 过滤器。还有另一种过滤器，这种过滤器对于数据的捕捉能力甚至可以胜过任何之前这些手写的过滤器。相比这种单纯的垂直边缘和水平边缘，它可以检测出 45°或 70°或 73°，甚至是任何角度的边缘。

所以将矩阵的所有数字都设置为参数，通过数据反馈，让神经网络自动去学习它们，我们会发现神经网络可以学习一些低级的特征，例如这些边缘的特征。尽管比起那些研究者们，我们要更费劲一些，但确实可以动手写出这些东西。不过构成这些计算的基础依然是卷积运算，使得反向传播算法能够让神经网络学习任何它所需要的 3×3 的过滤器，并在整幅图片上去应用它。这里，这里，还有这里（左边矩阵蓝色方框标记部分），去输出这些，任何它所检测到的特征，不管是垂直的边缘，水平的边缘，还有其他奇怪角度的边缘，甚至是其它的连名字都没有的过滤器。

所以这种将这 9 个数字当成参数的思想，已经成为计算机视觉中最为有效的思想之一。在接下来的课程中，也就是下个星期，我们将详细去探讨如何使用反向传播去让神经网络学习这 9 个数字。但在此之前，我们需要先讨论一些其它细节，比如一些基础的卷积运算的变量。在下面两节视频中，我将与你们讨论如何去使用 **padding**，以及卷积各种不同的发展，这两节内容将会是卷积神经网络中卷积模块的重要组成部分，所以我们下节视频再见。

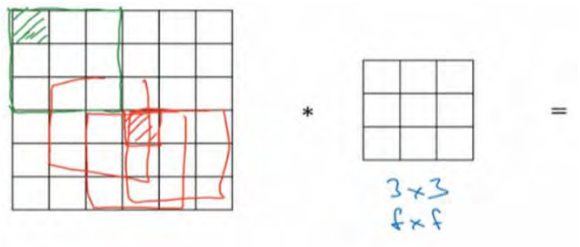
1.4 Padding

为了构建深度神经网络，你需要学会使用的一个基本的卷积操作就是 **padding**，让我们来看看它是如何工作的。

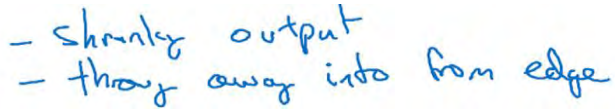


我们在之前视频中看到，如果你用一个 3×3 的过滤器卷积一个 6×6 的图像，你最后会得到一个 4×4 的输出，也就是一个 4×4 矩阵。那是因为你的 3×3 过滤器在 6×6 矩阵中，只可能有 4×4 种可能的位置。这背后的数学解释是，如果我们有一个$n \times n$的图像，用$f \times f$的过滤器做卷积，那么输出的维度就是$(n-f+1) \times (n-f+1)$。在这个例子里是$6-3+1=4$，因此得到了一个 4×4 的输出。

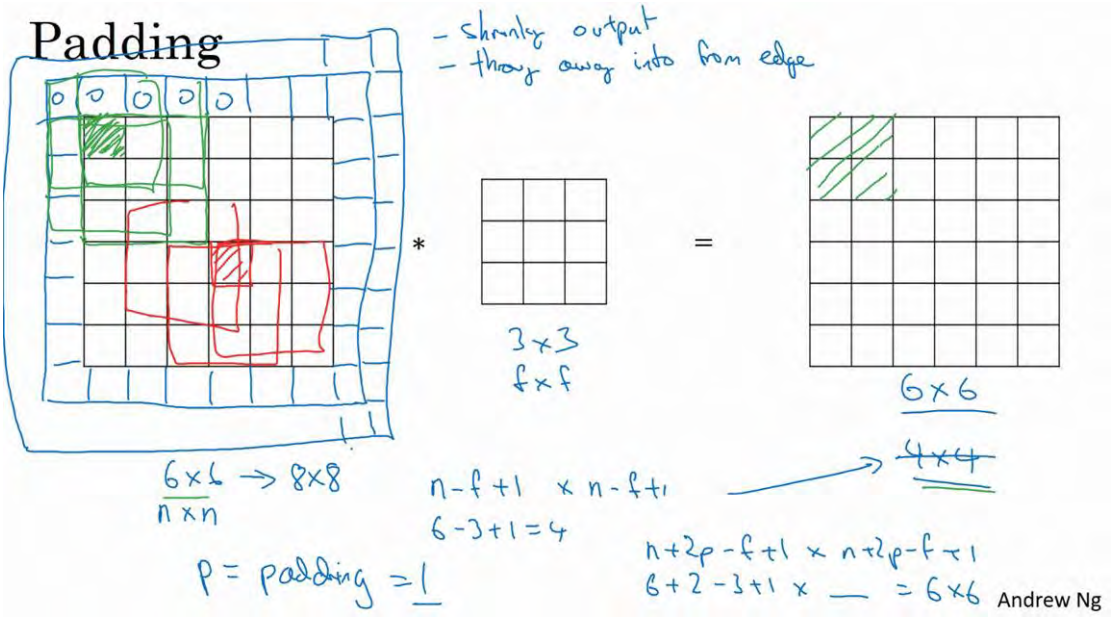
这样的话会有两个缺点，第一个缺点是每次做卷积操作，你的图像就会缩小，从 6×6 缩小到 4×4，你可能做了几次之后，你的图像就会变得很小了，可能会缩小到只有 1×1 的大小。你可不想让你的图像在每次识别边缘或其他特征时都缩小，这就是第一个缺点。



第二个缺点时，如果你注意角落边缘的像素，这个像素点（绿色阴影标记）只被一个输出所触碰或者使用，因为它位于这个 3×3 的区域的一角。但如果是在中间的像素点，比如这个（红色方框标记），就会有许多 3×3 的区域与之重叠。所以那些在角落或者边缘区域的像素点在输出中采用较少，意味着你丢掉了图像边缘位置的许多信息。

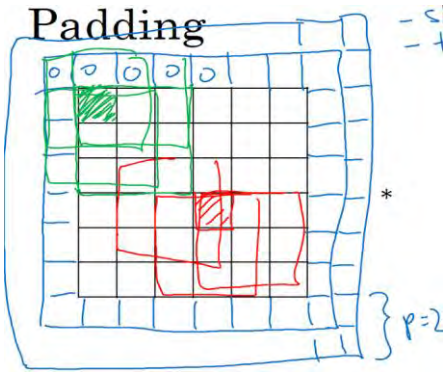


为了解决这两个问题，一是输出缩小。当我们建立深度神经网络时，你就会知道你为什么不希望每进行一步操作图像都会缩小。比如当你有 100 层深层的网络，如果图像每经过一层都缩小的话，经过 100 层网络后，你就会得到一个很小的图像，所以这是个问题。另一个问题是图像边缘的大部分信息都丢失了。



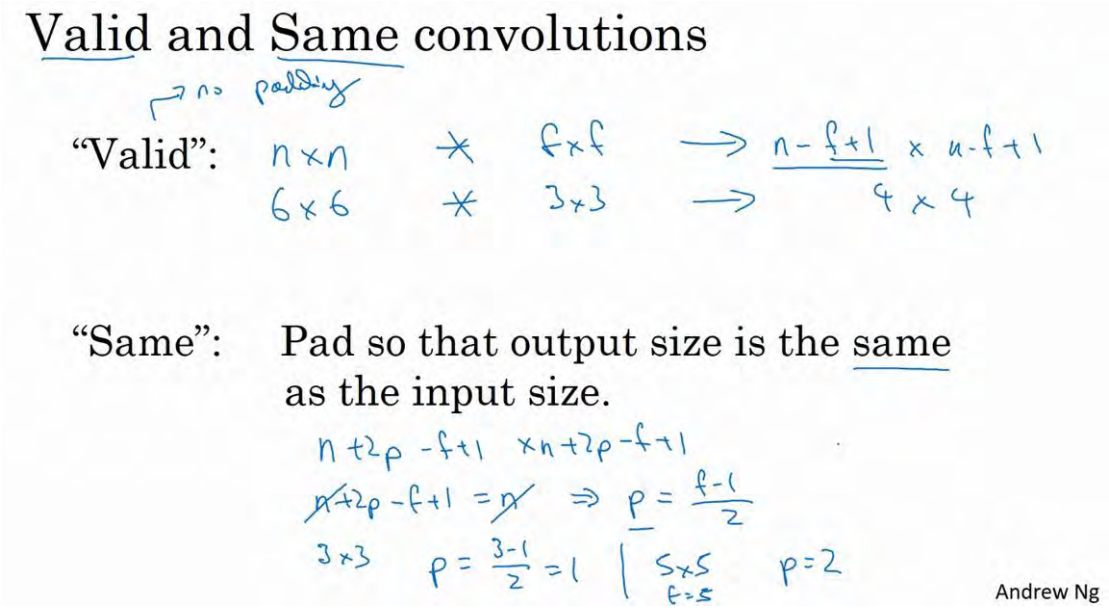
为了解决这些问题，你可以在卷积操作之前填充这幅图像。在这个案例中，你可以沿着图像边缘再填充一层像素。如果你这样操作了，那么 6×6 的图像就被你填充成了一个 8×8 的图像。如果你用 3×3 的图像对这个 8×8 的图像卷积，你得到的输出就不是 4×4 的，而是 6×6 的图像，你就得到了一个尺寸和原始图像 6×6 的图像。习惯上，你可以用 0 去填充，如果p是填充的数量，在这个案例中，$p = 1$，因为我们在周围都填充了一个像素点，输出也就变成了$(n + 2p - f + 1) \times (n + 2p - f + 1)$，所以就变成了$(6 + 2 \times 1 - 3 + 1) \times (6 + 2 \times 1 - 3 + 1) = 6 \times 6$，和输入的图像一样大。这个涂绿的像素点（左边矩阵）影响了输出中的这些格子（右边矩阵）。这样一来，丢失信息或者更准确来说角落或图像边缘的信息发挥的作用较小的这一缺点就被削弱了。

刚才我已经展示过用一个像素点来填充边缘，如果你想的话，也可以填充两个像素点，也就是说在这里填充一层。实际上你还可以填充更多像素。我这里画的这种情况，填充后$p = 2$。



至于选择填充多少像素，通常有两个选择，分别叫做 **Valid** 卷积和 **Same** 卷积，名字不怎么样。

Valid 卷积意味着不填充，这样的话，如果你有一个$n \times n$的图像，用一个$f \times f$的过滤器卷积，它将会给你一个$(n-f+1) \times (n-f+1)$维的输出。这类似于我们在前面的视频中展示的例子，有一个 6×6 的图像，通过一个 3×3 的过滤器，得到一个 4×4 的输出。



另一个经常被用到的填充方法叫做 Same 卷积，那意味你填充后，你的输出大小和输入大小是一样的。根据这个公式$n-f+1$，当你填充p个像素点，n就变成了$n+2p$，最后公式变为$n+2p-f+1$。因此如果你有一个$n \times n$的图像，用p个像素填充边缘，输出的大小就是这样的$(n+2p-f+1) \times (n+2p-f+1)$。如果你想让$n+2p-f+1=n$的话，使得输出和输入大小相等，如果你用这个等式求解p，那么$p=(f-1)/2$。所以当f是一个奇数的时候，只要选择相应的填充尺寸，你就能确保得到和输入相同尺寸的输出。这也是为什么前面的例子，当过滤器是 3×3 时，和上一张幻灯片的例子一样，使得输出尺寸等于输入尺寸，所需要的填充是(3-1)/2，也就是 1 个像素。另一个例子，当你的过滤器是 5×5，如果$f=5$，然后代入那个式子，你就会发现需要 2 层填充使得输出和输入一样大，这是过滤器 5×5 的情

况。

习惯上，计算机视觉中，f通常是奇数，甚至可能都是这样。你很少看到一个偶数的过滤器在计算机视觉里使用，我认为有两个原因。

其中一个可能是，如果f是一个偶数，那么你只能使用一些不对称填充。只有f是奇数的情况下，**Same** 卷积才会有自然的填充，我们可以以同样的数量填充四周，而不是左边填充多一点，右边填充少一点，这样不对称的填充。

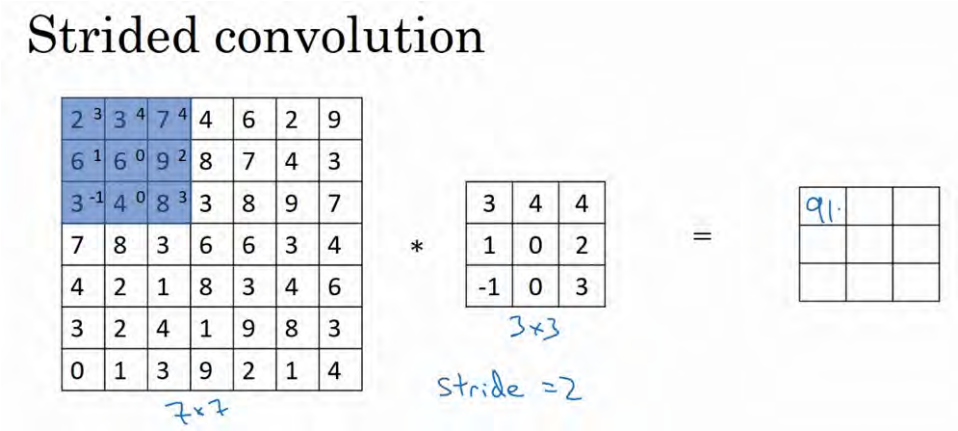
第二个原因是当你有一个奇数维过滤器，比如 3×3 或者 5×5 的，它就有一个中心点。有时在计算机视觉里，如果有一个中心像素点会更方便，便于指出过滤器的位置。

也许这些都不是为什么f通常是奇数的充分原因，但如果你看了卷积的文献，你经常会看到 3×3 的过滤器，你也可能会看到一些 5×5，7×7 的过滤器。后面我们也会谈到 1×1 的过滤器，以及什么时候它是有意义的。但是习惯上，我推荐你只使用奇数的过滤器。我想如果你使用偶数 f 也可能会得到不错的表现，如果遵循计算机视觉的惯例，我通常使用奇数值的f。

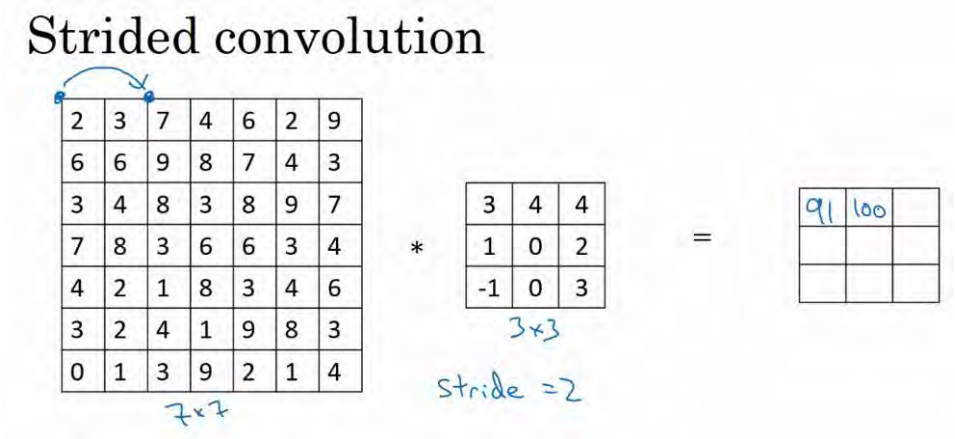
你已经看到如何使用 **padding** 卷积，为了指定卷积操作中的 **padding**，你可以指定p的值。也可以使用 **Valid** 卷积，也就是$p = 0$。也可使用 **Same** 卷积填充像素，使你的输出和输入大小相同。以上就是 **padding**，在接下来的视频中我们讨论如何在卷积中设置步长。

1.5 卷积步长（Strided convolutions）

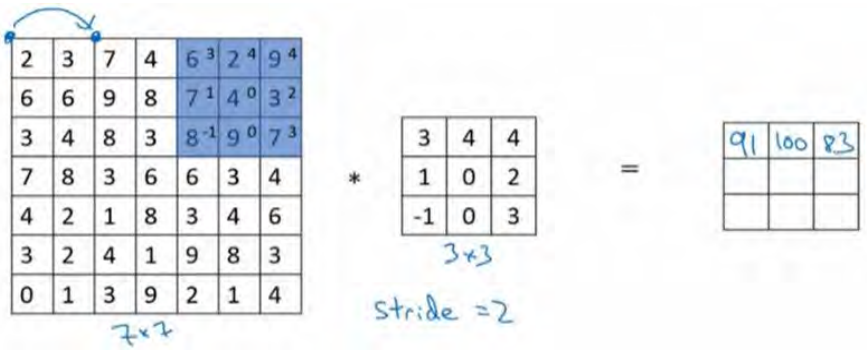
卷积中的步幅是另一个构建卷积神经网络的基本操作，让我向你展示一个例子。



如果你想用 3×3 的过滤器卷积这个 7×7 的图像，和之前不同的是，我们把步幅设置成了 2。你还和之前一样取左上方的 3×3 区域的元素的乘积，再加起来，最后结果为 91。

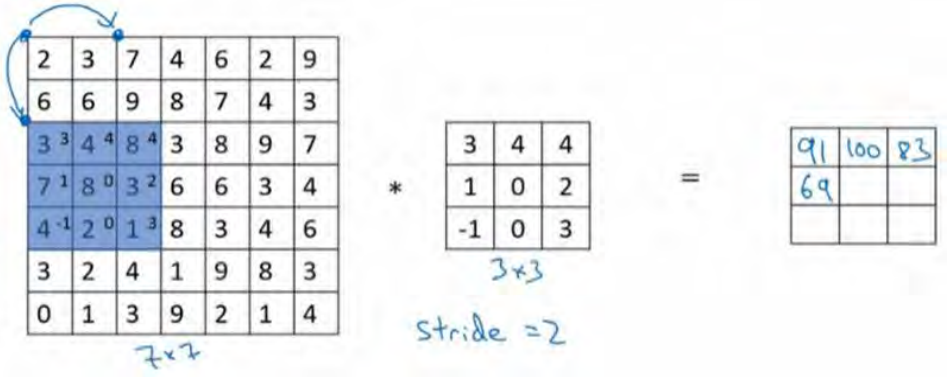


只是之前我们移动蓝框的步长是 1，现在移动的步长是 2，我们让过滤器跳过 2 个步长，注意一下左上角，这个点移动到其后两格的点，跳过了一个位置。然后你还是将每个元素相乘并求和，你将会得到的结果是 100。

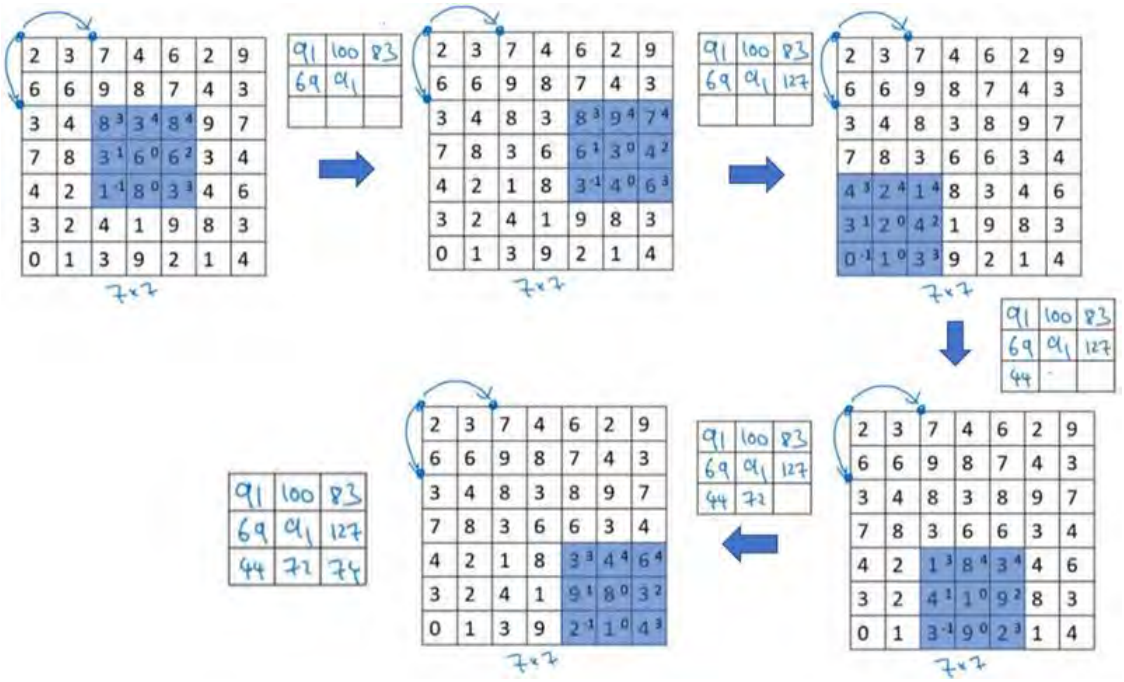


现在我们继续，将蓝色框移动两个步长，你将会得到 83 的结果。当你移动到下一行的

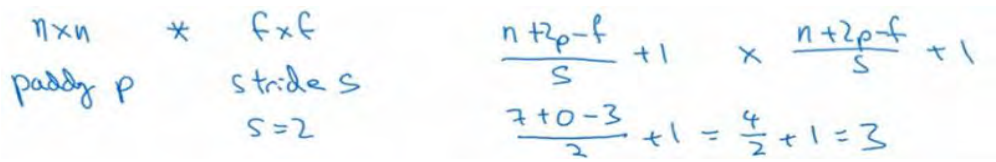
时候，你也是使用步长 2 而不是步长 1，所以我们将蓝色框移动到这里：



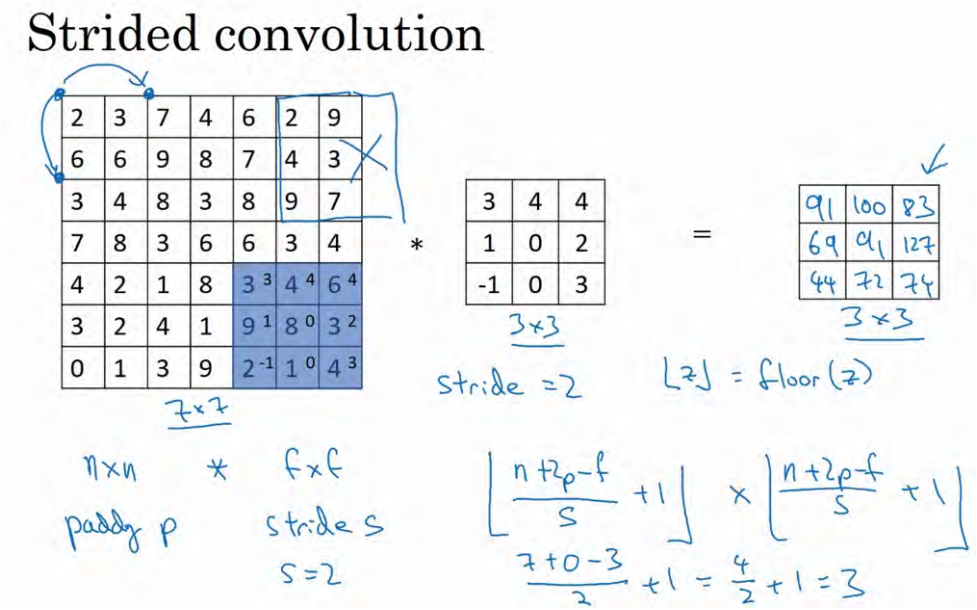
注意到我们跳过了一个位置，得到 69 的结果，现在你继续移动两个步长，会得到 91，127，最后一行分别是 44，72，74。



所以在这个例子中，我们用 3×3 的矩阵卷积一个 7×7 的矩阵，得到一个 3×3 的输出。输入和输出的维度是由下面的公式决定的。如果你用一个$f \times f$的过滤器卷积一个$n \times n$的图像，你的 **padding** 为p，步幅为s，在这个例子中$s = 2$，你会得到一个输出，因为现在你不是一次移动一个步子，而是一次移动s个步子，输出于是变为$\frac{n+2p-f}{s} + 1 \times \frac{n+2p-f}{s} + 1$



在我们的这个例子里，$n=7$，$p=0$，$f=3$，$s=2$，$\frac{7+0-3}{2}+1=3$，即 3×3 的输出。



现在只剩下最后的一个细节了，如果商不是一个整数怎么办？在这种情况下，我们向下取整。[ ]这是向下取整的符号，这也叫做对z进行地板除(**floor**)，这意味着z向下取整到最近的整数。这个原则实现的方式是，你只在蓝框完全包括在图像或填充完的图像内部时，才对它进行运算。如果有任意一个蓝框移动到了外面，那你就不要进行相乘操作，这是一个惯例。你的 3×3 的过滤器必须完全处于图像中或者填充之后的图像区域内才输出相应结果，这就是惯例。因此正确计算输出维度的方法是向下取整，以免$\frac{n+2p-f}{s}$不是整数。

Summary of convolutions

$n \times n$ image　　$f \times f$ filter

padding p　　stride s

Output Size:

$$\left\lfloor \frac{n+2p-f}{s}+1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s}+1 \right\rfloor$$

总结一下维度情况，如果你有一个$n \times n$的矩阵或者$n \times n$的图像，与一个$f \times f$的矩阵卷积，或者说$f \times f$的过滤器。**Padding** 是p，步幅为s没输出尺寸就是这样：

Summary of convolutions

$n \times n$ image　　$f \times f$ filter

padding p　　stride s

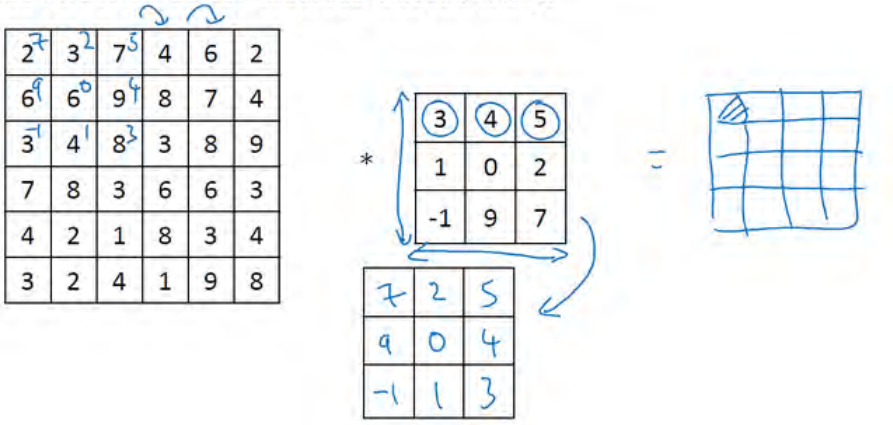
Output Size:

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

可以选择所有的数使结果是整数是挺不错的，尽管一些时候，你不必这样做，只要向下取整也就可以了。你也可以自己选择一些n，f，p和s的值来验证这个输出尺寸的公式是对的。

Technical note on cross-correlation vs. convolution

Convolution in math textbook:



在讲下一部分之前，这里有一个关于互相关和卷积的技术性建议，这不会影响到你构建卷积神经网络的方式，但取决于你读的是数学教材还是信号处理教材，在不同的教材里符号可能不一致。如果你看的是一本典型的数学教科书，那么卷积的定义是做元素乘积求和，实际上还有一个步骤是你首先要做的，也就是在把这个 6×6 的矩阵和 3×3 的过滤器卷积之前，首先你将 3×3 的过滤器沿水平和垂直轴翻转，所以$\begin{bmatrix} 3 & 4 & 5 \\ 1 & 0 & 2 \\ -1 & 9 & 7 \end{bmatrix}$变为$\begin{bmatrix} 7 & 2 & 5 \\ 9 & 0 & 4 \\ -1 & 1 & 3 \end{bmatrix}$，这相当于将 3×3 的过滤器做了个镜像，在水平和垂直轴上（整理者注：此处应该是先顺时针旋转 90 得到$\begin{bmatrix} -1 & 1 & 3 \\ 9 & 0 & 4 \\ 7 & 2 & 5 \end{bmatrix}$，再水平翻转得到$\begin{bmatrix} 7 & 2 & 5 \\ 9 & 0 & 4 \\ -1 & 1 & 3 \end{bmatrix}$）。然后你再把这个翻转后的矩阵复制到这里（左边的图像矩阵），你要把这个翻转矩阵的元素相乘来计算输出的 4×4 矩阵左上角的元

素，如图所示。然后取这 9 个数字，把它们平移一个位置，再平移一格，以此类推。

所以我们在这些视频中定义卷积运算时，我们跳过了这个镜像操作。从技术上讲，我们实际上做的，我们在前面视频中使用的操作，有时被称为互相关（**cross-correlation**）而不是卷积（**convolution**）。但在深度学习文献中，按照惯例，我们将这（不进行翻转操作）叫做卷积操作。

总结来说，按照机器学习的惯例，我们通常不进行翻转操作。从技术上说，这个操作可能叫做互相关更好。但在大部分的深度学习文献中都把它叫做卷积运算，因此我们将在这些视频中使用这个约定。如果你读了很多机器学习文献的话，你会发现许多人都把它叫做卷积运算，不需要用到这些翻转。

事实证明在信号处理中或某些数学分支中，在卷积的定义包含翻转，使得卷积运算符拥有这个性质，即$(A * B) * C = A * (B * C)$，这在数学中被称为结合律。这对于一些信号处理应用来说很好，但对于深度神经网络来说它真的不重要，因此省略了这个双重镜像操作，就简化了代码，并使神经网络也能正常工作。

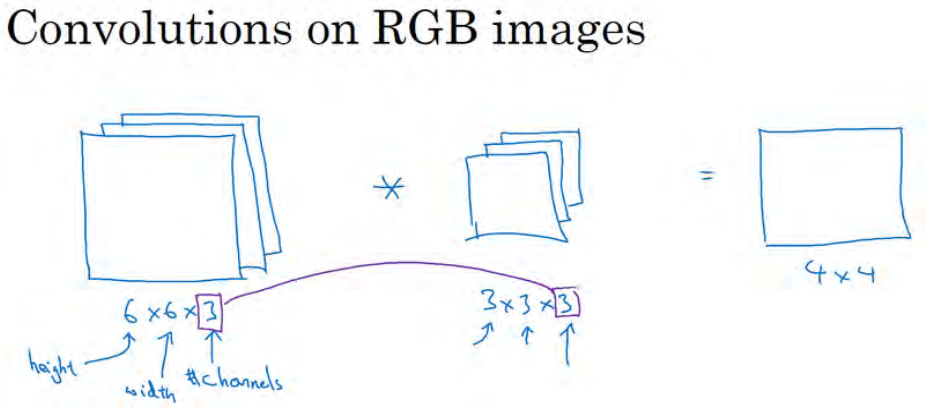
根据惯例，我们大多数人都叫它卷积，尽管数学家们更喜欢称之为互相关，但这不会影响到你在编程练习中要实现的任何东西，也不会影响你阅读和理解深度学习文献。

现在你已经看到了如何进行卷积，以及如何使用填充，如何在卷积中选择步幅。但到目前为止，我们所使用的是关于矩阵的卷积，例如 6×6 的矩阵。在下一集视频中，你将看到如何对立体进行卷积，这将会使你的卷积变得更加强大，让我们继续下一个视频。

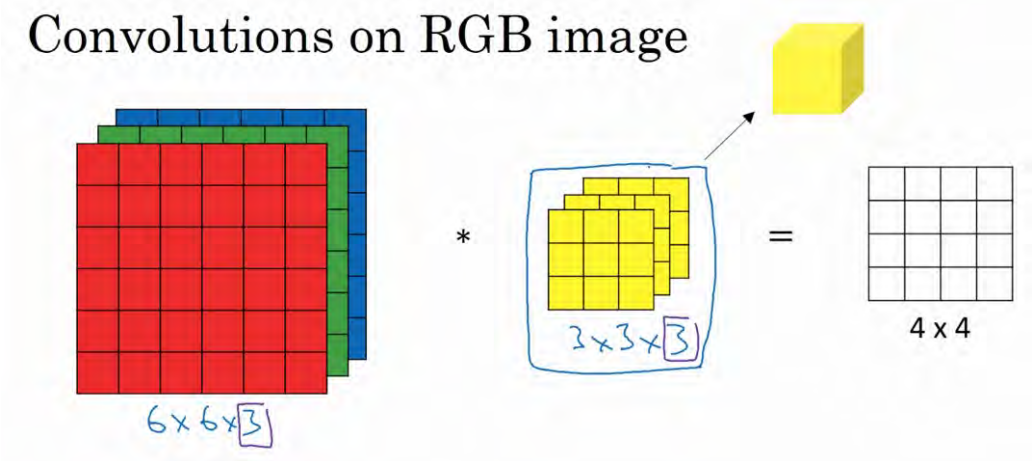
1.6 三维卷积（Convolutions over volumes）

你已经知道如何对二维图像做卷积了，现在看看如何执行卷积不仅仅在二维图像上，而是三维立体上。

我们从一个例子开始，假如说你不仅想检测灰度图像的特征，也想检测 **RGB** 彩色图像的特征。彩色图像如果是 6×6×3，这里的 3 指的是三个颜色通道，你可以把它想象成三个 6×6 图像的堆叠。为了检测图像的边缘或者其他的特征，不是把它跟原来的 3×3 的过滤器做卷积，而是跟一个三维的过滤器，它的维度是 3×3×3，这样这个过滤器也有三层，对应红绿、蓝三个通道。

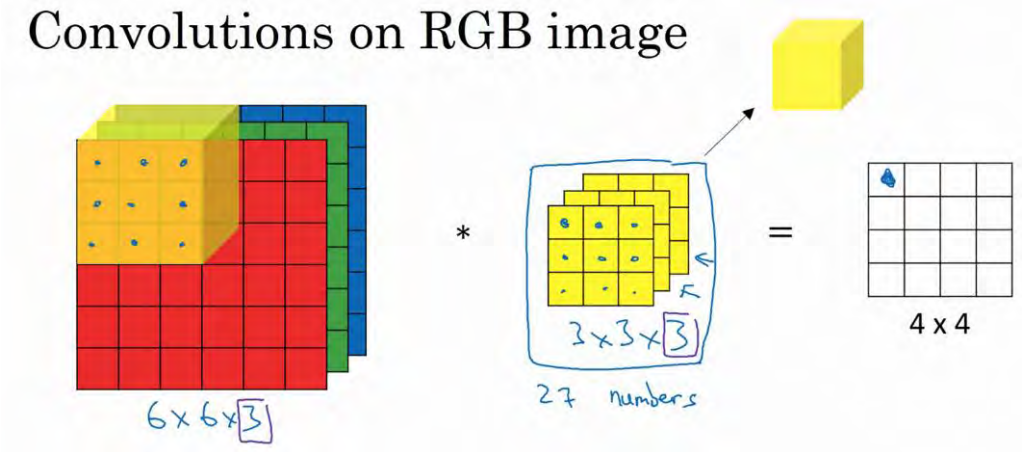


给这些命个名字（原图像），这里的第一个 6 代表图像高度，第二个 6 代表宽度，这个 3 代表通道的数目。同样你的过滤器也有一个高，宽和通道数，并且图像的通道数必须和过滤器的通道数匹配，所以这两个数（紫色方框标记的两个数）必须相等。下个幻灯片里，我们就会知道这个卷积操作是如何进行的了，这个的输出会是一个 4×4 的图像，注意是 4×4×1，最后一个数不是 3 了。



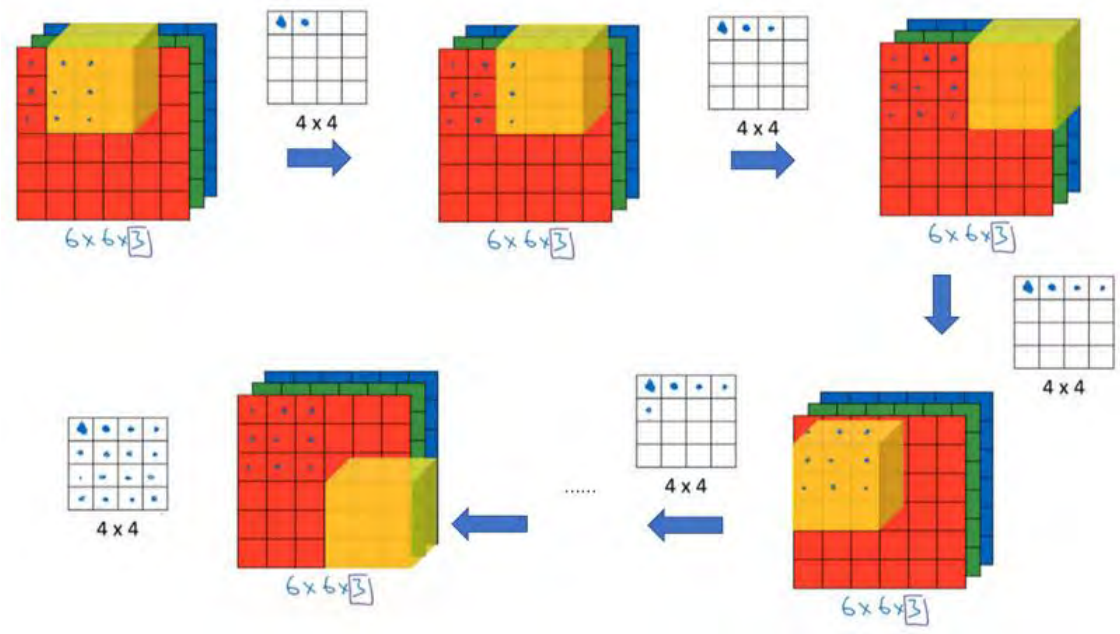
我们研究下这背后的细节，首先先换一张好看的图片。这个是 6×6×3 的图像，这个是

3×3×3 的过滤器，最后一个数字通道数必须和过滤器中的通道数相匹配。为了简化这个 3×3×3 过滤器的图像，我们不把它画成 3 个矩阵的堆叠，而画成这样，一个三维的立方体。



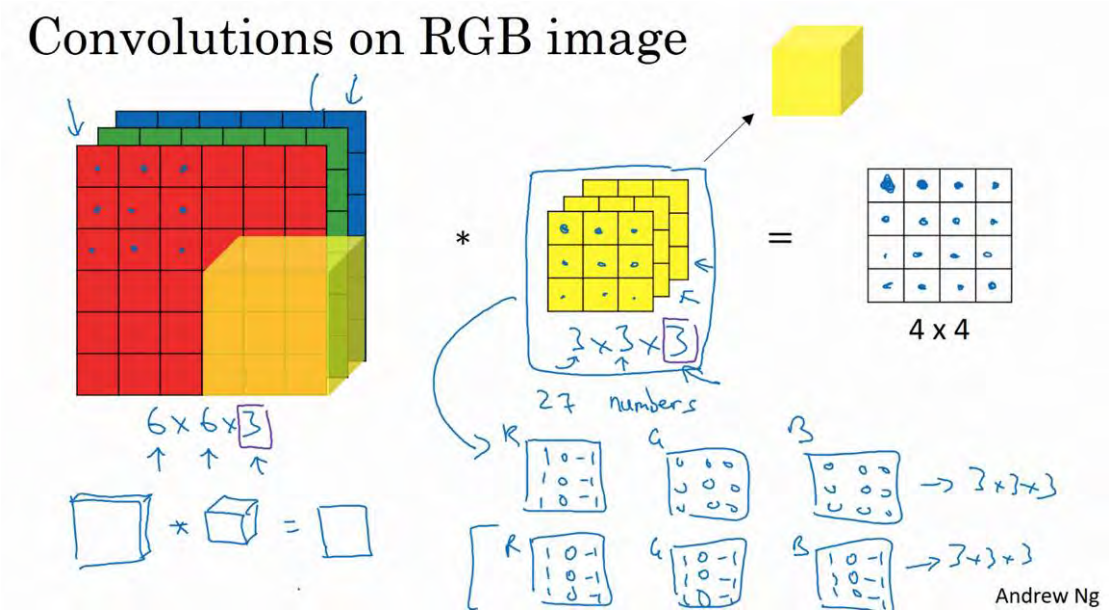
为了计算这个卷积操作的输出，你要做的就是把这个 3×3×3 的过滤器先放到最左上角的位置，这个 3×3×3 的过滤器有 27 个数，27 个参数就是 3 的立方。依次取这 27 个数，然后乘以相应的红绿蓝通道中的数字。先取红色通道的前 9 个数字，然后是绿色通道，然后再是蓝色通道，乘以左边黄色立方体覆盖的对应的 27 个数，然后把这些数都加起来，就得到了输出的第一个数字。

如果要计算下一个输出，你把这个立方体滑动一个单位，再与这 27 个数相乘，把它们都加起来，就得到了下一个输出，以此类推。



那么，这个能干什么呢？举个例子，这个过滤器是 3×3×3 的，如果你想检测图像红色通道的边缘，那么你可以将第一个过滤器设为 $\begin{bmatrix}1 & 0 & -1\\1 & 0 & -1\\1 & 0 & -1\end{bmatrix}$，和之前一样，而绿色通道全为 0，

$\begin{bmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix}$，蓝色也全为 0。如果你把这三个堆叠在一起形成一个 3×3×3 的过滤器，那么这就是一个检测垂直边界的过滤器，但只对红色通道有用。



或者如果你不关心垂直边界在哪个颜色通道里，那么你可以用一个这样的过滤器，

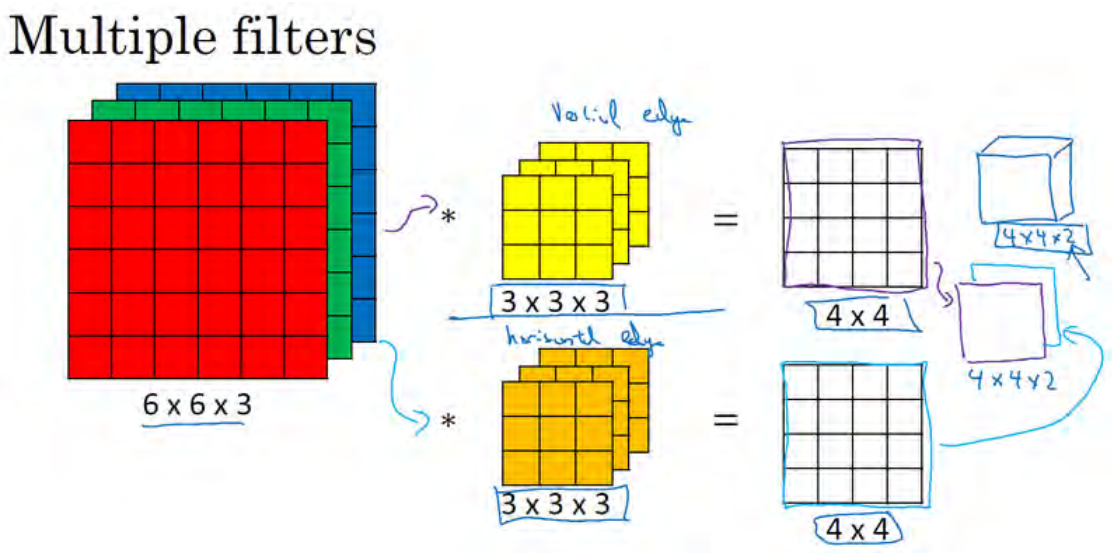
$\begin{bmatrix}1 & 0 & -1\\1 & 0 & -1\\1 & 0 & -1\end{bmatrix}$，$\begin{bmatrix}1 & 0 & -1\\1 & 0 & -1\\1 & 0 & -1\end{bmatrix}$，$\begin{bmatrix}1 & 0 & -1\\1 & 0 & -1\\1 & 0 & -1\end{bmatrix}$，所有三个通道都是这样。所以通过设置第二个过滤器参数，你就有了一个边界检测器，3×3×3 的边界检测器，用来检测任意颜色通道里的边界。参数的选择不同，你就可以得到不同的特征检测器，所有的都是 3×3×3 的过滤器。

按照计算机视觉的惯例，当你的输入有特定的高宽和通道数时，你的过滤器可以有不同的高，不同的宽，但是必须一样的通道数。理论上，我们的过滤器只关注红色通道，或者只关注绿色或者蓝色通道也是可行的。

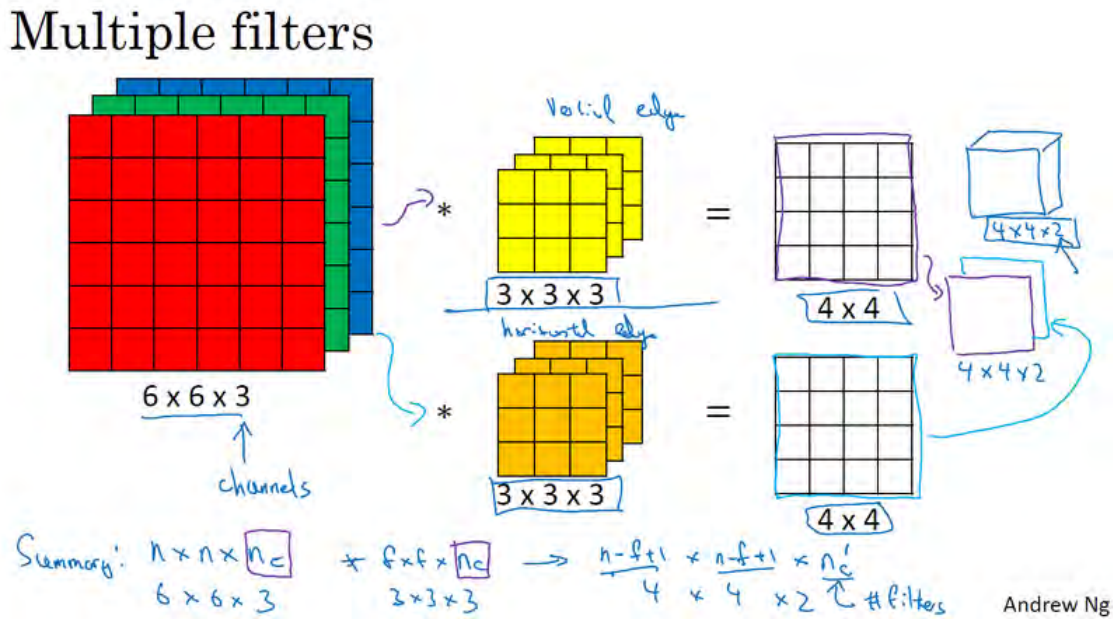
再注意一下这个卷积立方体，一个 6×6×6 的输入图像卷积上一个 3×3×3 的过滤器，得到一个 4×4 的二维输出。

现在你已经了解了如何对立方体卷积，还有最后一个概念，对建立卷积神经网络至关重要。就是，如果我们不仅仅想要检测垂直边缘怎么办？如果我们同时检测垂直边缘和水平边缘，还有 45°倾斜的边缘，还有 70°倾斜的边缘怎么做？换句话说，如果你想同时用多个过滤器怎么办？

这是我们上一张幻灯片的图片，我们让这个 6×6×3 的图像和这个 3×3×3 的过滤器卷积，得到 4×4 的输出。（第一个）这可能是一个垂直边界检测器或者是学习检测其他的特征。第二个过滤器可以用橘色来表示，它可以是一个水平边缘检测器。



所以和第一个过滤器卷积，可以得到第一个 4×4 的输出，然后卷积第二个过滤器，得到一个不同的 4×4 的输出。我们做完卷积，然后把这两个 4×4 的输出，取第一个把它放到前面，然后取第二个过滤器输出，我把它画在这，放到后面。所以把这两个输出堆叠在一起，这样你就都得到了一个 4×4×2 的输出立方体，你可以把这个立方体当成，重新画在这，就是一个这样的盒子，所以这就是一个 4×4×2 的输出立方体。它用 6×6×3 的图像，然后卷积上这两个不同的 3×3 的过滤器，得到两个 4×4 的输出，它们堆叠在一起，形成一个 4×4×2 的立方体，这里的 2 的来源于我们用了两个不同的过滤器。



我们总结一下维度，如果你有一个$n \times n \times n_c$（通道数）的输入图像，在这个例子中就是 6×6×3，这里的n_c就是通道数目，然后卷积上一个$f \times f \times n_c$，这个例子中是 3×3×3，按照惯例，这个（前一个n_c）和这个（后一个n_c）必须数值相同。然后你就得到了$(n-f+1)$

× （$n-f+1$）× $n_{c'}$，这里$n_{c'}$其实就是下一层的通道数，它就是你用的过滤器的个数，在我们的例子中，那就是 4×4×2。我写下这个假设时，用的步幅为 1，并且没有 **padding**。如果你用了不同的步幅或者 **padding**，那么这个$n-f+1$数值会变化，正如前面的视频演示的那样。

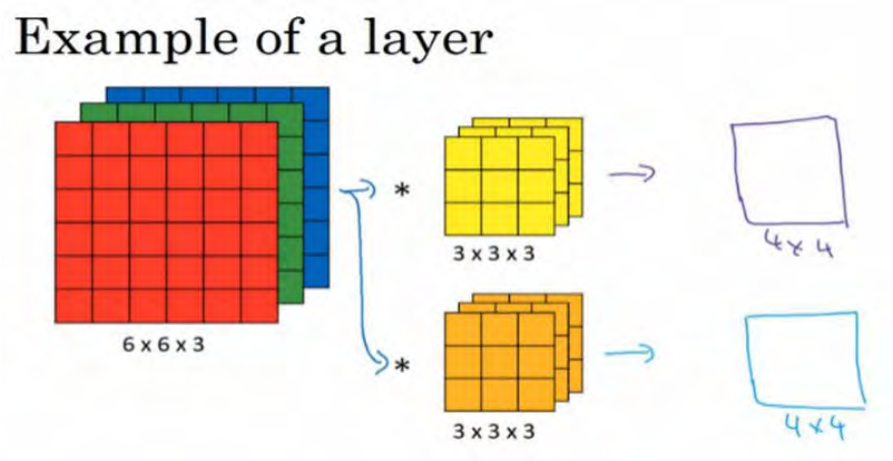
这个对立方体卷积的概念真的很有用，你现在可以用它的一小部分直接在三个通道的 RGB 图像上进行操作。更重要的是，你可以检测两个特征，比如垂直和水平边缘或者 10 个或者 128 个或者几百个不同的特征，并且输出的通道数会等于你要检测的特征数。

对于这里的符号，我一直用通道数（n_c）来表示最后一个维度，在文献里大家也把它叫做 3 维立方体的深度。这两个术语，即通道或者深度，经常被用在文献中。但我觉得深度容易让人混淆，因为你通常也会说神经网络的深度。所以，在这些视频里我会用通道这个术语来表示过滤器的第三个维度的大小。

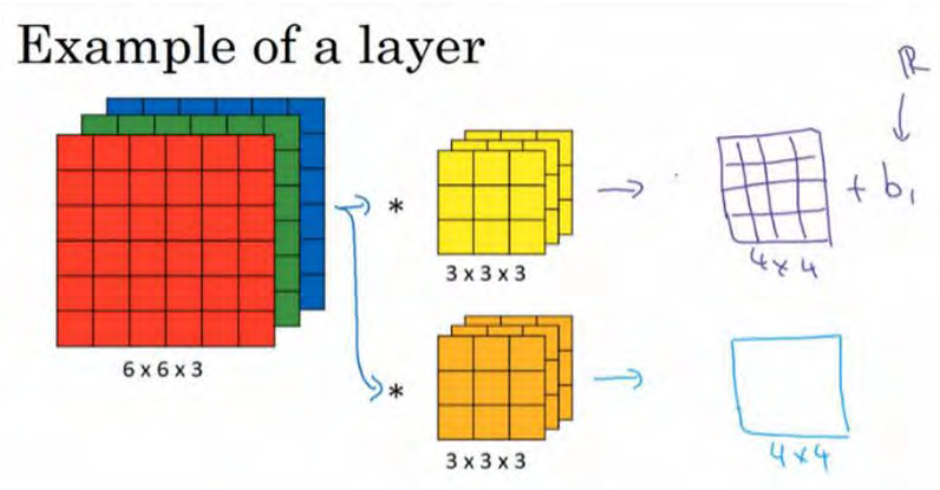
所以你已经知道怎么对立方体做卷积了，你已经准备好了实现卷积神经其中一层了，在下个视频里让我们看看是怎么做的。

1.7 单层卷积网络（One layer of a convolutional network）

今天我们要讲的是如何构建卷积神经网络的卷积层，下面来看个例子。

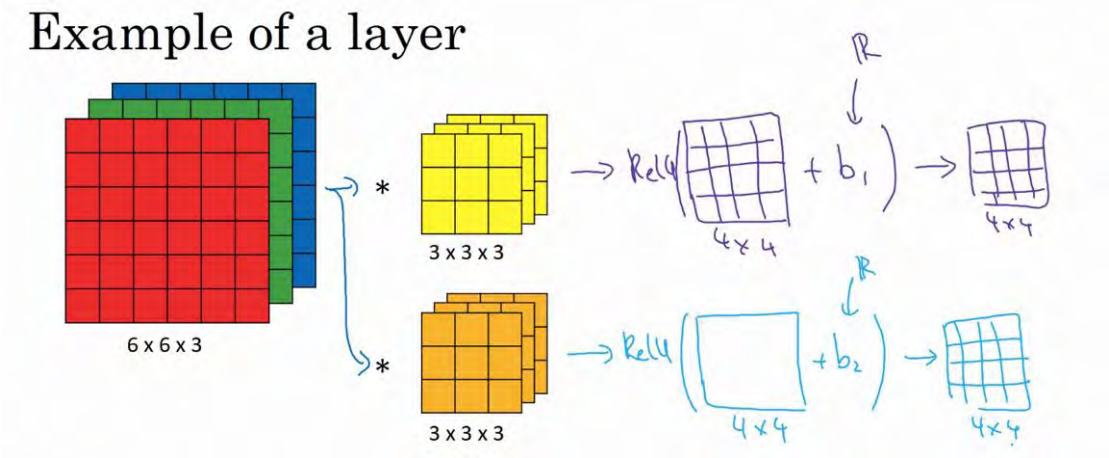


上节课，我们已经讲了如何通过两个过滤器卷积处理一个三维图像，并输出两个不同的 4×4 矩阵。假设使用第一个过滤器进行卷积，得到第一个 4×4 矩阵。使用第二个过滤器进行卷积得到另外一个 4×4 矩阵。

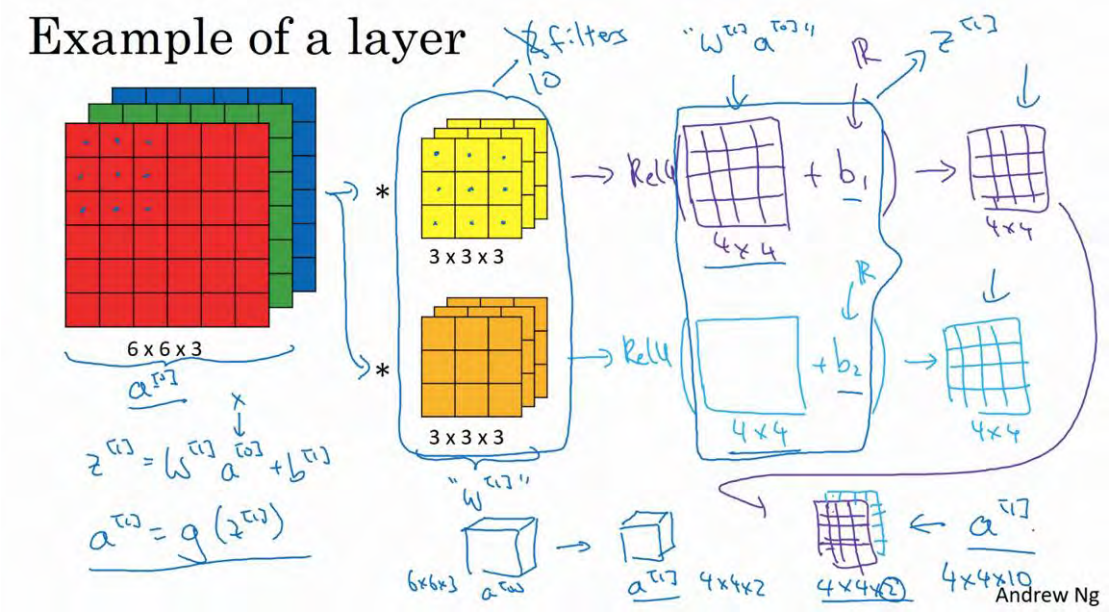


最终各自形成一个卷积神经网络层，然后增加偏差，它是一个实数，通过 **Python** 的广播机制给这 16 个元素都加上同一偏差。然后应用非线性函数，为了说明，它是一个非线性激活函数 **ReLU**，输出结果是一个 4×4 矩阵。

对于第二个 4×4 矩阵，我们加上不同的偏差，它也是一个实数，16 个数字都加上同一个实数，然后应用非线性函数，也就是一个非线性激活函数 **ReLU**，最终得到另一个 4×4 矩阵。然后重复我们之前的步骤，把这两个矩阵堆叠起来，最终得到一个 4×4×2 的矩阵。我们通过计算，从 6×6×3 的输入推导出一个 4×4×2 矩阵，它是卷积神经网络的一层，把它映射到标准神经网络中四个卷积层中的某一层或者一个非卷积神经网络中。



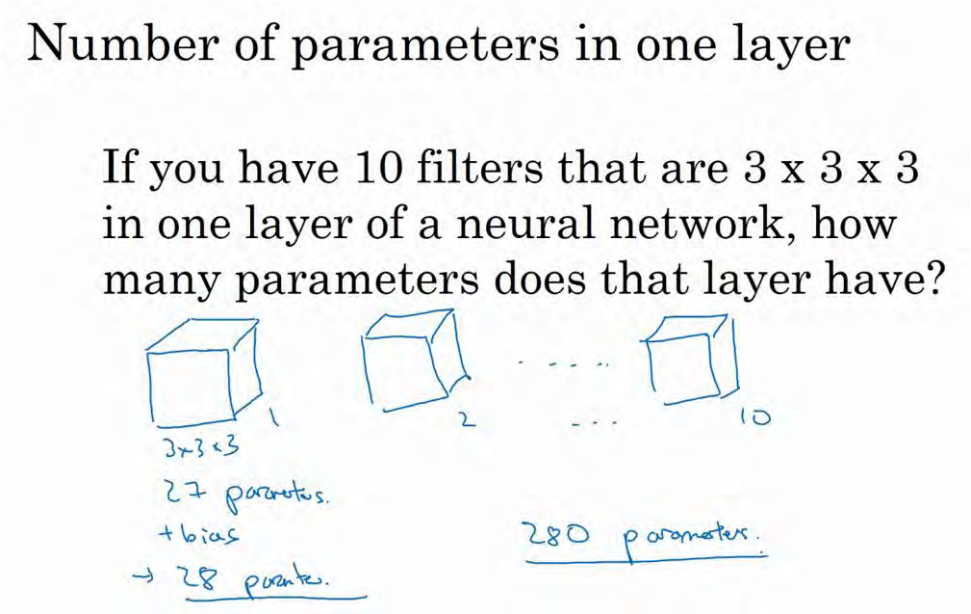
注意前向传播中一个操作就是$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$，其中$a^{[0]} = x$，执行非线性函数得到$a^{[1]}$，即$a^{[1]} = g(z^{[1]})$。这里的输入是$a^{[0]}$，也就是$x$，这些过滤器用变量$W^{[1]}$表示。在卷积过程中，我们对这 27 个数进行操作，其实是 27×2，因为我们用了两个过滤器，我们取这些数做乘法。实际执行了一个线性函数，得到一个 4×4 的矩阵。卷积操作的输出结果是一个 4×4 的矩阵，它的作用类似于$W^{[1]}a^{[0]}$，也就是这两个 4×4 矩阵的输出结果，然后加上偏差。



这一部分（图中蓝色边框标记的部分）就是应用激活函数 **ReLU** 之前的值，它的作用类似于$z^{[1]}$，最后应用非线性函数，得到的这个 4×4×2 矩阵，成为神经网络的下一层，也就是激活层。

这就是$a^{[0]}$到$a^{[1]}$的演变过程，首先执行线性函数，然后所有元素相乘做卷积，具体做法是运用线性函数再加上偏差，然后应用激活函数 **ReLU**。这样就通过神经网络的一层把一个 6×6×3 的维度$a^{[0]}$演化为一个 4×4×2 维度的$a^{[1]}$，这就是卷积神经网络的一层。

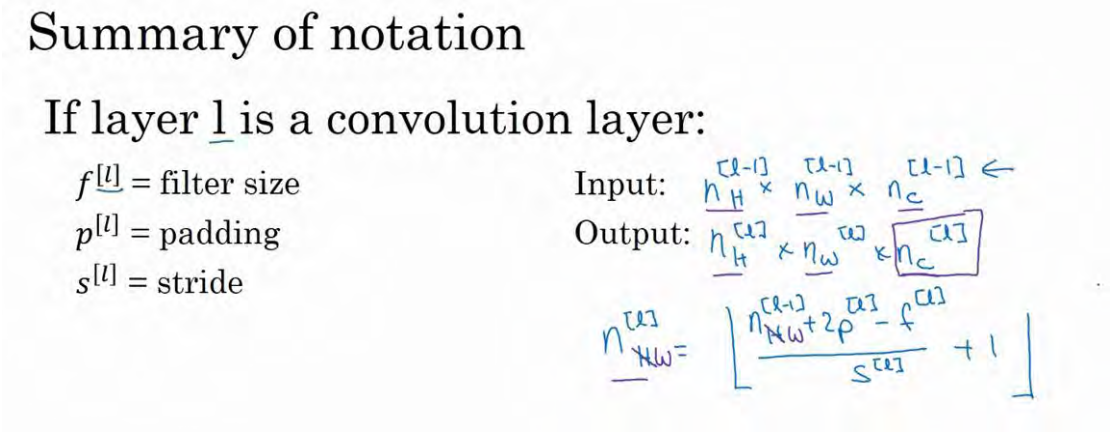
示例中我们有两个过滤器，也就是有两个特征，因此我们才最终得到一个 4×4×2 的输出。但如果我们用了 10 个过滤器，而不是 2 个，我们最后会得到一个 4×4×10 维度的输出图像，因为我们选取了其中 10 个特征映射，而不仅仅是 2 个，将它们堆叠在一起，形成一个 4×4×10 的输出图像，也就是$a^{[1]}$。



为了加深理解，我们来做一个练习。假设你有 10 个过滤器，而不是 2 个，神经网络的一层是 3×3×3，那么，这一层有多少个参数呢？我们来计算一下，每一层都是一个 3×3×3 的矩阵，因此每个过滤器有 27 个参数，也就是 27 个数。然后加上一个偏差，用参数b表示，现在参数增加到 28 个。上一页幻灯片里我画了 2 个过滤器，而现在我们有 10 个，加在一起是 28×10，也就是 280 个参数。

请注意一点，不论输入图片有多大，1000×1000 也好，5000×5000 也好，参数始终都是 280 个。用这 10 个过滤器来提取特征，如垂直边缘，水平边缘和其它特征。即使这些图片很大，参数却很少，这就是卷积神经网络的一个特征，叫作“**避免过拟合**”。你已经知道到如何提取 10 个特征，可以应用到大图片中，而参数数量固定不变，此例中只有 28 个，相对较少。

最后我们总结一下用于描述卷积神经网络中的一层（以l层为例），也就是卷积层的各种标记。



这一层是卷积层，用$f^{[l]}$表示过滤器大小，我们说过过滤器大小为$f \times f$，上标$[l]$表示l层中过滤器大小为$f \times f$。通常情况下，上标$[l]$用来标记l层。用$p^{[l]}$来标记 **padding** 的数量，**padding** 数量也可指定为一个 **valid** 卷积，即无 **padding**。或是 **same** 卷积，即选定 **padding**，如此一来，输出和输入图片的高度和宽度就相同了。用$s^{[l]}$标记步幅。

这一层的输入会是某个维度的数据，表示为$n \times n \times n_c$，n_c某层上的颜色通道数。

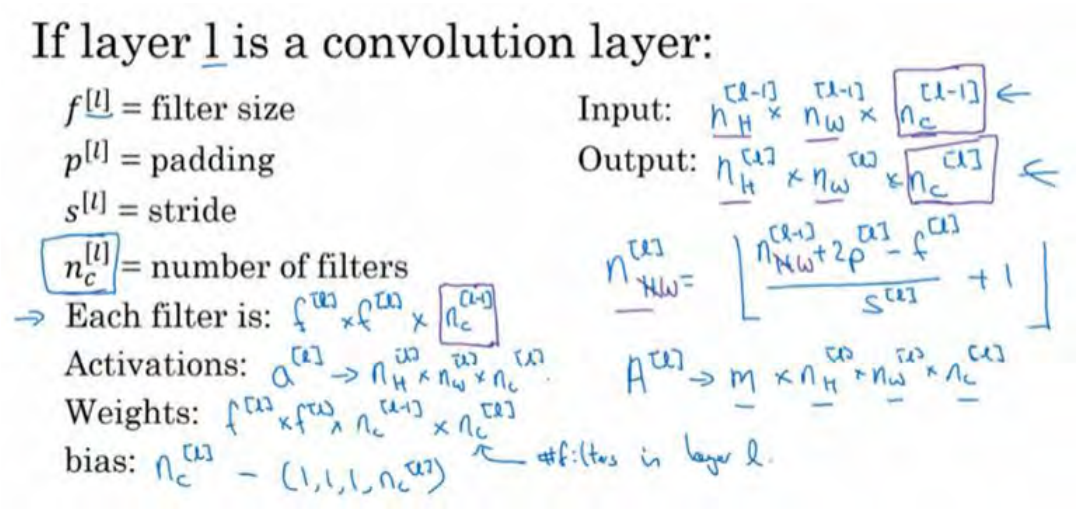
我们要稍作修改，增加上标$[l-1]$，即$n^{[l-1]} \times n^{[l-1]} \times n_c^{[l-1]}$，因为它是上一层的激活值。

此例中，所用图片的高度和宽度都一样，但它们也有可能不同，所以分别用上下标H和W来标记，即$n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$。那么在第$l$层，图片大小为$n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$，$l$层的输入就是上一层的输出，因此上标要用$[l-1]$。神经网络这一层中会有输出，它本身会输出图像。其大小为$n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$，这就是输出图像的大小。

前面我们提到过，这个公式给出了输出图片的大小，至少给出了高度和宽度，$\lfloor \frac{n+2p-f}{s} + 1 \rfloor$（$\frac{n+2p-f}{s} + 1$)直接用这个运算结果，也可以向下取整）。在这个新表达式中，l层输出图像的高度，即$n_H^{[l]} = \lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$，同样我们可以计算出图像的宽度，用$W$替换参数$H$，即$n_W^{[l]} = \lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$，公式一样，只要变化高度和宽度的参数我们便能计算输出图像的高度或宽度。这就是由$n_H^{[l-1]}$推导$n_H^{[l]}$以及$n_W^{[l-1]}$推导$n_W^{[l]}$的过程。

那么通道数量又是什么？这些数字从哪儿来的？我们来看一下。输出图像也具有深度，通过上一个示例，我们知道它等于该层中过滤器的数量，如果有 2 个过滤器，输出图像就是 4×4×2，它是二维的，如果有 10 个过滤器，输出图像就是 4×4×10。输出图像中的通道数量就是神经网络中这一层所使用的过滤器的数量。如何确定过滤器的大小呢？我们知道卷积一个 6×6×3 的图片需要一个 3×3×3 的过滤器，因此过滤器中通道的数量必须与输入中通道的

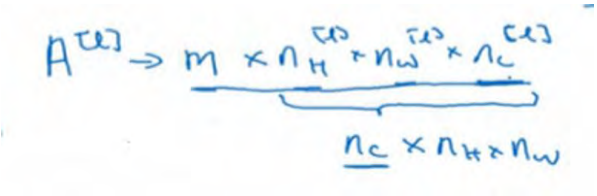
数量一致。因此，输出通道数量就是输入通道数量，所以过滤器维度等于$f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$。



应用偏差和非线性函数之后，这一层的输出等于它的激活值$a^{[l]}$，也就是这个维度（输出维度）。$a^{[l]}$是一个三维体，即$n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$。当你执行批量梯度下降或小批量梯度下降时，如果有$m$个例子，就是有$m$个激活值的集合，那么输出$A^{[l]} = m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$。如果采用批量梯度下降，变量的排列顺序如下，首先是索引和训练示例，然后是其它三个变量。

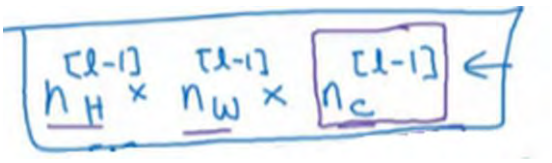
该如何确定权重参数，即参数 W 呢？过滤器的维度已知，为$f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$，这只是一个过滤器的维度，有多少个过滤器，这（$n_c^{[l]}$）是过滤器的数量，权重也就是所有过滤器的集合再乘以过滤器的总数量，即$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$，损失数量 L 就是$l$层中过滤器的个数。

最后我们看看偏差参数，每个过滤器都有一个偏差参数，它是一个实数。偏差包含了这些变量，它是该维度上的一个向量。后续课程中我们会看到，为了方便，偏差在代码中表示为一个 1×1×1×$n_c^{[l]}$的四维向量或四维张量。



卷积有很多种标记方法，这是我们最常用的卷积符号。大家在线搜索或查看开源代码时，关于高度，宽度和通道的顺序并没有完全统一的标准卷积，所以在查看 **GitHub** 上的源代码或阅读一些开源实现的时候，你会发现有些作者会采用把通道放在首位的编码标准，有时所有变量都采用这种标准。实际上在某些架构中，当检索这些图片时，会有一个变量或参数来标识计算通道数量和通道损失数量的先后顺序。只要保持一致，这两种卷积标准都可用。很

遗憾，这只是一部分标记法，因为深度学习文献并未对标记达成一致，但课上我会采用这种卷积标识法，按高度，宽度和通道损失数量的顺序依次计算。

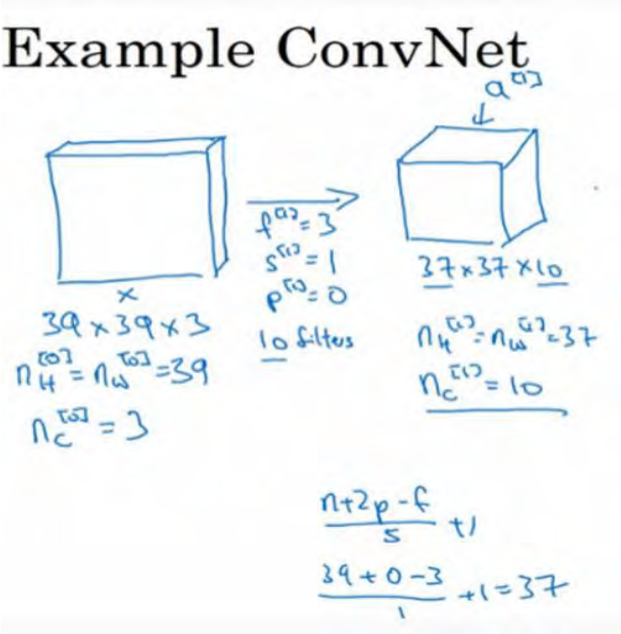


我知道，忽然间接触到这么多新的标记方法，你可能会说，这么多怎么记呢？别担心，不用全都记住，你可以通过本周的练习来熟悉它们。而这节课我想讲的重点是，卷积神经网络的某一卷积层的工作原理，以及如何计算某一卷积层的激活函数，并映射到下一层的激活值。了解了卷积神经网络中某一卷积层的工作原理，我们就可以把它们堆叠起来形成一个深度卷积神经网络，我们下节课再讲。

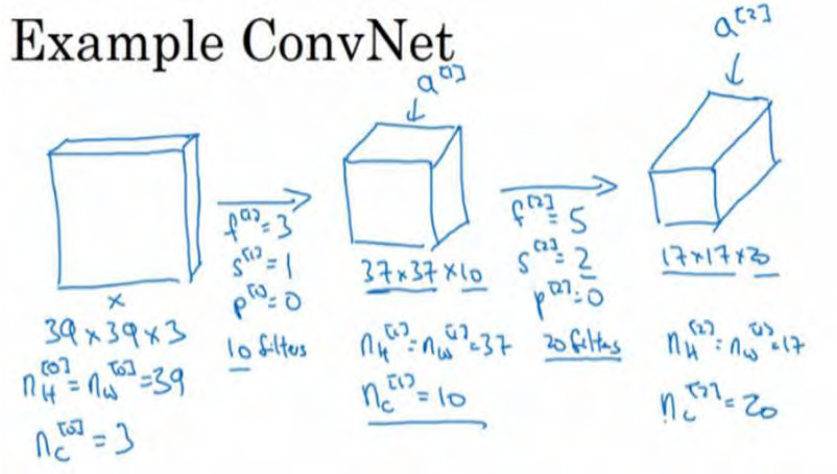
1.8 简单卷积网络示例（A simple convolution network example）

上节课，我们讲了如何为卷积网络构建一个卷积层。今天我们看一个深度卷积神经网络的具体示例，顺便练习一下我们上节课所学的标记法。

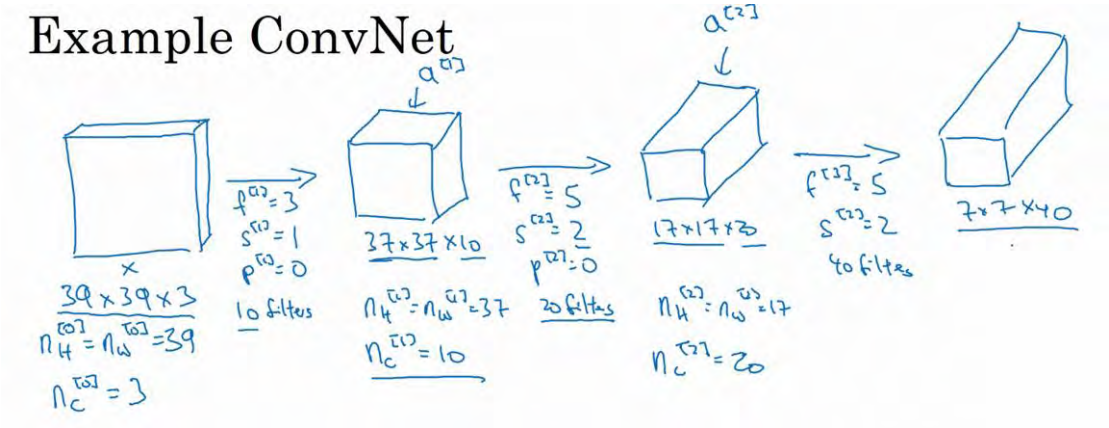
假设你有一张图片，你想做图片分类或图片识别，把这张图片输入定义为x，然后辨别图片中有没有猫，用 0 或 1 表示，这是一个分类问题，我们来构建适用于这项任务的卷积神经网络。针对这个示例，我用了一张比较小的图片，大小是 39×39×3，这样设定可以使其中一些数字效果更好。所以$n_H^{[0]} = n_W^{[0]}$，即高度和宽度都等于 39，$n_c^{[0]} = 3$，即 0 层的通道数为 3。



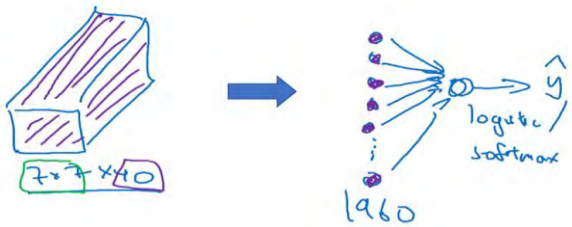
假设第一层我们用一个 3×3 的过滤器来提取特征，那么$f^{[1]} = 3$，因为过滤器时 3×3 的矩阵。$s^{[1]} = 1$，$p^{[1]} = 0$，所以高度和宽度使用 same 卷积。如果有 10 个过滤器，神经网络下一层的激活值为 37×37×10，写 10 是因为我们用了 10 个过滤器，37 是公式$\frac{n+2p-f}{s} + 1$的计算结果，也就是$\frac{39+0-3}{1} + 1 = 37$，所以输出是 37×37，它是一个 **vaild** 卷积，这是输出结果的大小。第一层标记为$n_H^{[1]} = n_W^{[1]} = 37$，$n_c^{[1]} = 10$，$n_c^{[1]}$等于第一层中过滤器的个数，这（37×37×10）是第一层激活值的维度。



假设还有另外一个卷积层，这次我们采用的过滤器是 5×5 的矩阵。在标记法中，神经网络下一层的$f = 5$，即$f^{[2]} = 5$步幅为 2，即$s^{[2]} = 2$。**padding** 为 0，即$p^{[2]} = 0$，且有 20 个过滤器。所以其输出结果会是一张新图像，这次的输出结果为 17×17×20，因为步幅是 2，维度缩小得很快，大小从 37×37 减小到 17×17，减小了一半还多，过滤器是 20 个，所以通道数也是 20，17×17×20 即激活值$a^{[2]}$的维度。因此$n_H^{[2]} = n_W^{[2]} = 17$，$n_c^{[2]} = 20$。



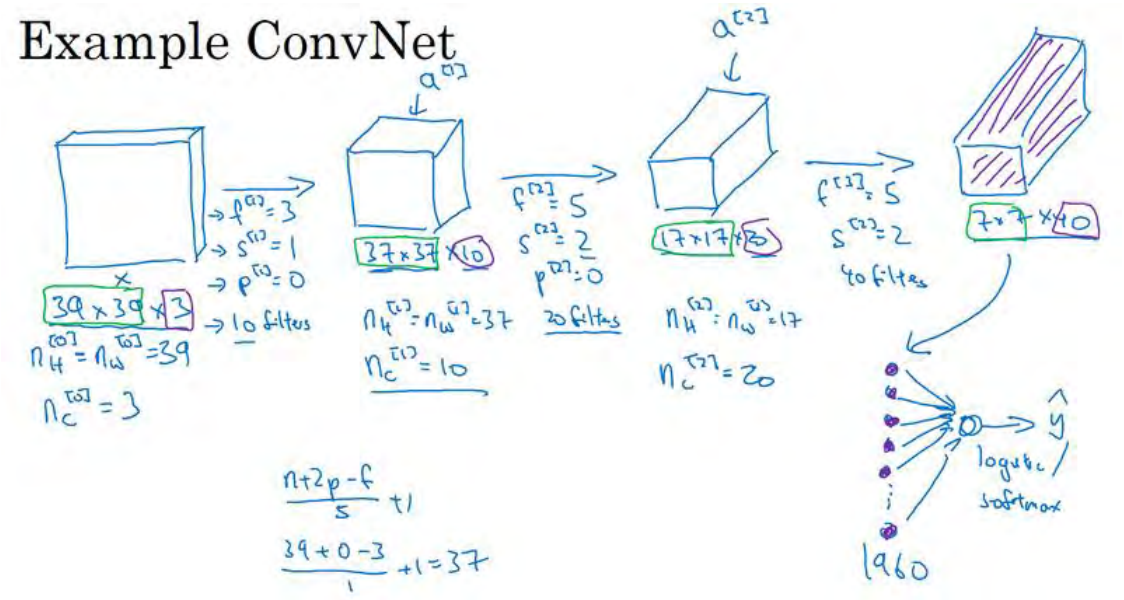
我们来构建最后一个卷积层，假设过滤器还是 5×5，步幅为 2，即$f^{[2]} = 5$，$s^{[3]} = 2$，计算过程我跳过了，最后输出为 7×7×40，假设使用了 40 个过滤器。**padding** 为 0，40 个过滤器，最后结果为 7×7×40。



到此，这张 39×39×3 的输入图像就处理完毕了，为图片提取了 7×7×40 个特征，计算出来就是 1960 个特征。然后对该卷积进行处理，可以将其平滑或展开成 1960 个单元。平滑处

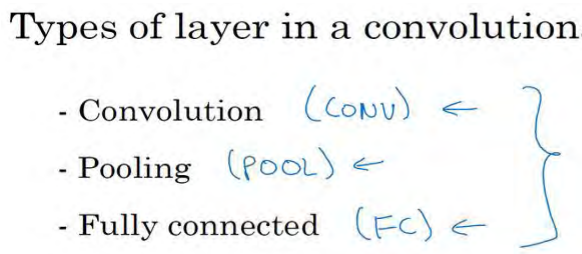
理后可以输出一个向量，其填充内容是 **logistic** 回归单元还是 **softmax** 回归单元，完全取决于我们是想识图片上有没有猫，还是想识别K种不同对象中的一种，用$\hat{y}$表示最终神经网络的预测输出。明确一点，最后这一步是处理所有数字，即全部的 1960 个数字，把它们展开成一个很长的向量。为了预测最终的输出结果，我们把这个长向量填充到 **softmax** 回归函数中。

这是卷积神经网络的一个典型范例，设计卷积神经网络时，确定这些超参数比奥费工夫。要决定过滤器的大小、步幅、**padding** 以及使用多少个过滤器。这周和下周，我会针对选择参数的问题提供一些建议和指导。



而这节课你要掌握的一点是，随着神经网络计算深度不断加深，通常开始时的图像也要更大一些，初始值为 39×39，高度和宽度会在一段时间内保持一致，然后随着网络深度的加深而逐渐减小，从 39 到 37，再到 17，最后到 7。而通道数量在增加，从 3 到 10，再到 20，最后到 40。在许多其它卷积神经网络中，你也可以看到这种趋势。关于如何确定这些参数，后面课上我会更详细讲解，这是我们讲的第一个卷积神经网络示例。

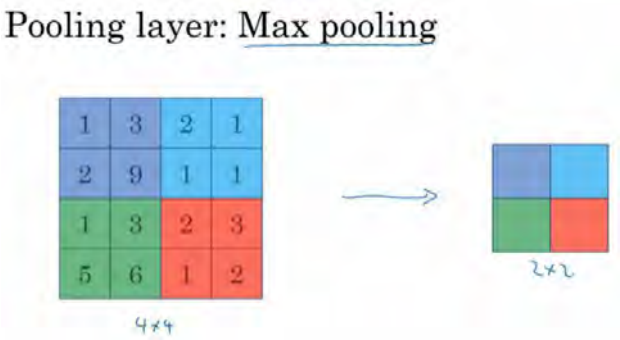
一个典型的卷积神经网络通常有三层，一个是卷积层，我们常常用 **Conv** 来标注。上一个例子，我用的就是 **CONV**。还有两种常见类型的层，我们留在后两节课讲。一个是池化层，我们称之为 **POOL**。最后一个是全连接层，用 **FC** 表示。虽然仅用卷积层也有可能构建出很好的神经网络，但大部分神经望楼架构师依然会添加池化层和全连接层。幸运的是，池化层和全连接层比卷积层更容易设计。后两节课我们会快速讲解这两个概念以便你更好的了解神经网络中最常用的这几种层，你就可以利用它们构建更强大的网络了。



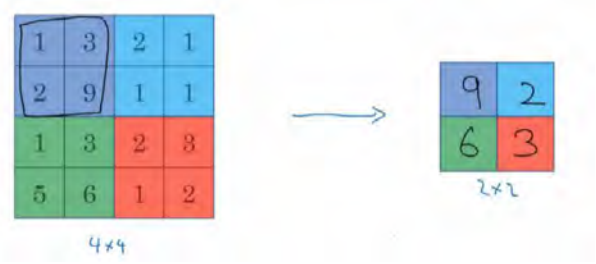
再次恭喜你已经掌握了第一个卷积神经网络，本周后几节课，我们会学习如何训练这些卷积神经网络。不过在这之前，我还要简单介绍一下池化层和全连接层。然后再训练这些网络，到时我会用到大家熟悉的反向传播训练方法。那么下节课，我们就先来了解如何构建神经网络的池化层。

1.9 池化层（Pooling layers）

除了卷积层，卷积网络也经常使用池化层来缩减模型的大小，提高计算速度，同时提高所提取特征的鲁棒性，我们来看一下。

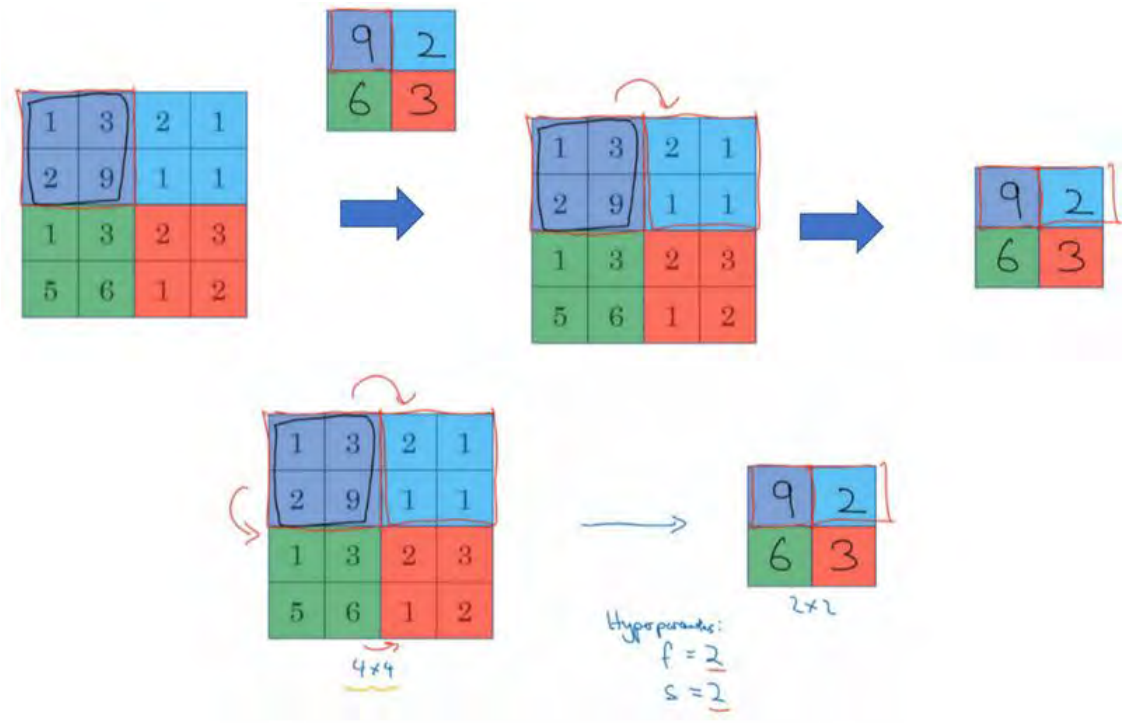


先举一个池化层的例子，然后我们再讨论池化层的必要性。假如输入是一个 4×4 矩阵，用到的池化类型是最大池化（**max pooling**）。执行最大池化的树池是一个 2×2 矩阵。执行过程非常简单，把 4×4 的输入拆分成不同的区域，我把这个区域用不同颜色来标记。对于 2×2 的输出，输出的每个元素都是其对应颜色区域中的最大元素值。



左上区域的最大值是 9，右上区域的最大元素值是 2，左下区域的最大值是 6，右下区域的最大值是 3。为了计算出右侧这 4 个元素值，我们需要对输入矩阵的 2×2 区域做最大值运算。这就像是应用了一个规模为 2 的过滤器，因为我们选用的是 2×2 区域，步幅是 2，这些就是最大池化的超参数。

因为我们使用的过滤器为 2×2，最后输出是 9。然后向右移动 2 个步幅，计算出最大值 2。然后是第二行，向下移动 2 步得到最大值 6。最后向右移动 3 步，得到最大值 3。这是一个 2×2 矩阵，即$f = 2$，步幅是 2，即$s = 2$。



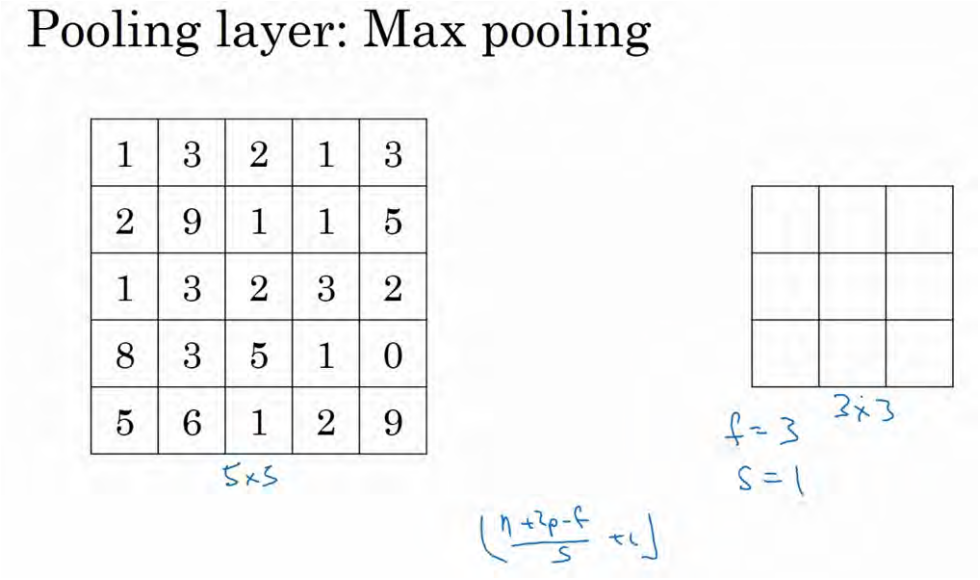
这是对最大池化功能的直观理解，你可以把这个 4×4 输入看作是某些特征的集合，也许不是。你可以把这个 4×4 区域看作是某些特征的集合，也就是神经网络中某一层的非激活值集合。数字大意味着可能探测到了某些特定的特征，左上象限具有的特征可能是一个垂直边缘，一只眼睛，或是大家害怕遇到的 **CAP** 特征。显然左上象限中存在这个特征，这个特征可能是一只猫眼探测器。然而，右上象限并不存在这个特征。最大化操作的功能就是只要在任何一个象限内提取到某个特征，它都会保留在最大化的池化输出里。所以最大化运算的实际作用就是，如果在过滤器中提取到某个特征，那么保留其最大值。如果没有提取到这个特征，可能在右上象限中不存在这个特征，那么其中的最大值也还是很小，这就是最大池化的直观理解。

必须承认，人们使用最大池化的主要原因是此方法在很多实验中效果都很好。尽管刚刚描述的直观理解经常被引用，不知大家是否完全理解它的真正原因，不知大家是否理解最大池化效率很高的真正原因。

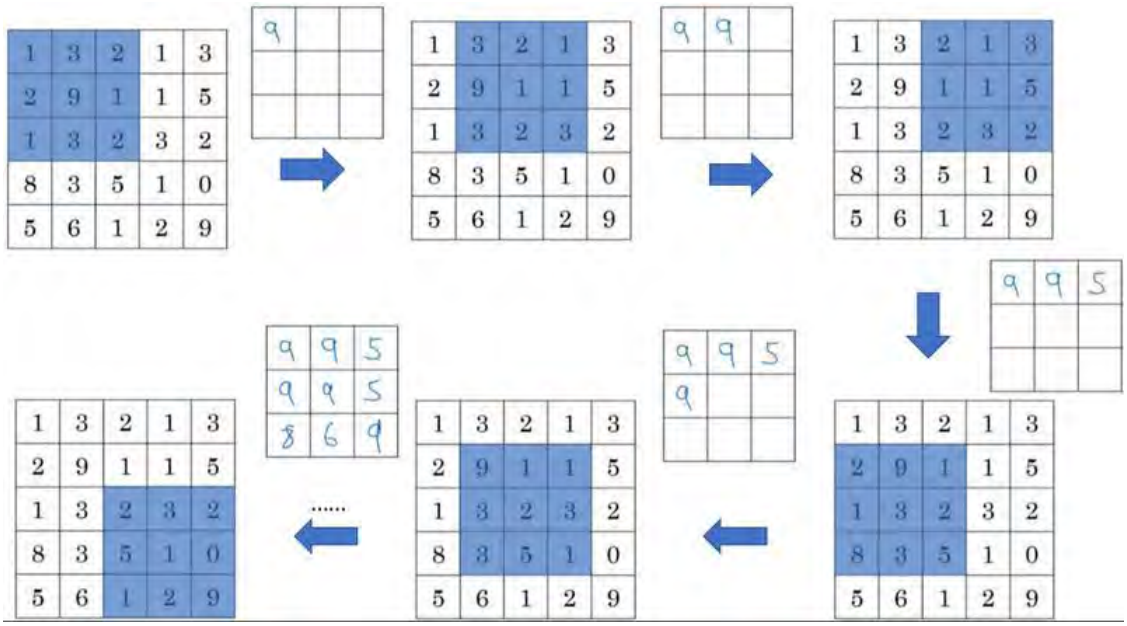
其中一个有意思的特点就是，它有一组超参数，但并没有参数需要学习。实际上，梯度下降没有什么可学的，一旦确定了f和s，它就是一个固定运算，梯度下降无需改变任何值。

我们来看一个有若干个超级参数的示例，输入是一个 5×5 的矩阵。我们采用最大池化法，它的过滤器参数为 3×3，即$f = 3$，步幅为 1，$s = 1$，输出矩阵是 3×3.之前讲的计算卷积层输出大小的公式同样适用于最大池化，即$\frac{n+2p-f}{s} + 1$，这个公式也可以计算最大池化的

输出大小。

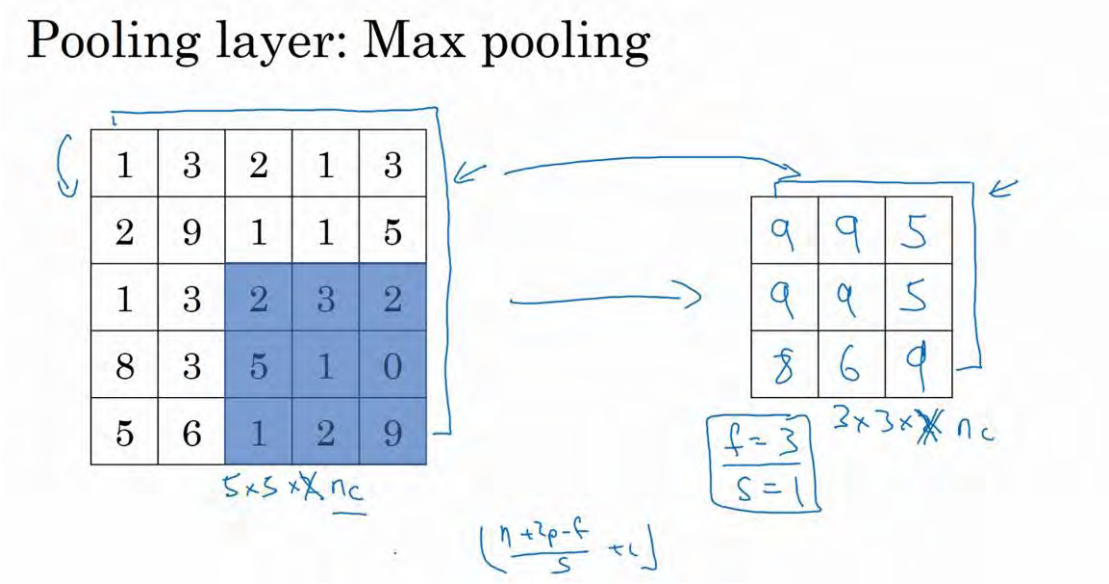


此例是计算 3×3 输出的每个元素，我们看左上角这些元素，注意这是一个 3×3 区域，因为有 3 个过滤器，取最大值 9。然后移动一个元素，因为步幅是 1，蓝色区域的最大值是 9. 继续向右移动，蓝色区域的最大值是 5。然后移到下一行，因为步幅是 1，我们只向下移动一个格，所以该区域的最大值是 9。这个区域也是 9。这两个区域的最大值都是 5。最后这三个区域的最大值分别为 8，6 和 9。超参数$f = 3$，$s = 1$，最终输出如图所示。

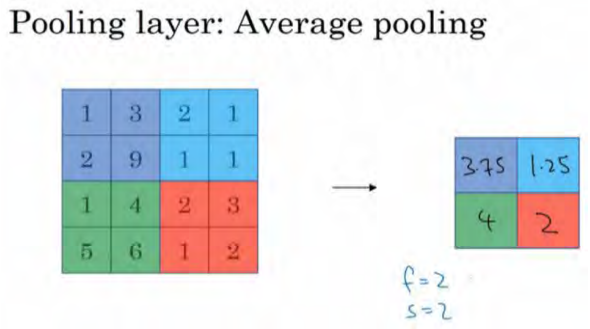


以上就是一个二维输入的最大池化的演示，如果输入是三维的，那么输出也是三维的。例如，输入是 5×5×2，那么输出是 3×3×2。计算最大池化的方法就是分别对每个通道执行刚刚的计算过程。如上图所示，第一个通道依然保持不变。对于第二个通道，我刚才画在下面的，在这个层做同样的计算，得到第二个通道的输出。一般来说，如果输入是 5×5×n_c，输出

就是 3×3×n_c，n_c个通道中每个通道都单独执行最大池化计算，以上就是最大池化算法。



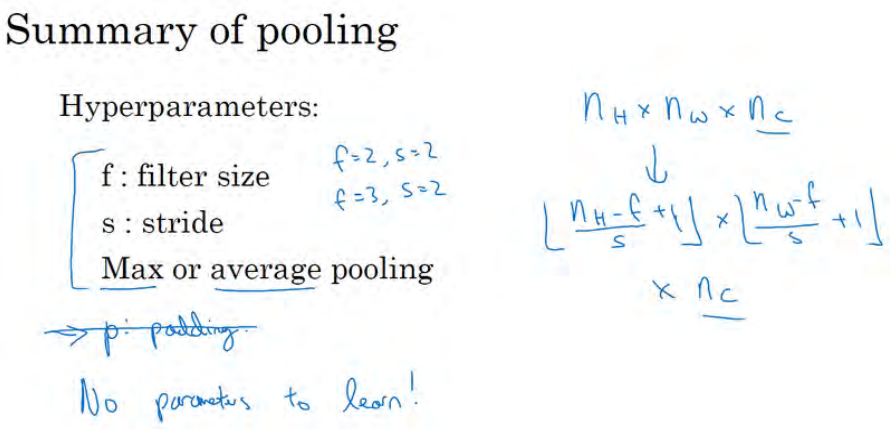
另外还有一种类型的池化，平均池化，它不太常用。我简单介绍一下，这种运算顾名思义，选取的不是每个过滤器的最大值，而是平均值。示例中，紫色区域的平均值是 3.75，后面依次是 1.25、4 和 2。这个平均池化的超级参数$f = 2$，$s = 2$，我们也可以选择其它超级参数。



目前来说，最大池化比平均池化更常用。但也有例外，就是深度很深的神经网络，你可以用平均池化来分解规模为 7×7×1000 的网络的表示层，在整个空间内求平均值，得到 1×1×1000，一会我们看个例子。但在神经网络中，最大池化要比平均池化用得更多。

总结一下，池化的超级参数包括过滤器大小f和步幅s，常用的参数值为$f = 2$，$s = 2$，应用频率非常高，其效果相当于高度和宽度缩减一半。也有使用$f = 3$，$s = 2$的情况。至于其它超级参数就要看你用的是最大池化还是平均池化了。你也可以根据自己意愿增加表示 **padding** 的其他超级参数，虽然很少这么用。最大池化时，往往很少用到超参数 padding，当然也有例外的情况，我们下周会讲。大部分情况下，最大池化很少用 **padding**。目前p最常用的值是 0，即 p=0。最大池化的输入就是$n_H \times n_W \times n_c$，假设没有 **padding**，则输出$\lfloor \frac{n_H - f}{s} +$

$1] \times \lfloor \frac{n_w - f}{s} + 1] \times n_c$。输入通道与输出通道个数相同，因为我们对每个通道都做了池化。需要注意的一点是，池化过程中没有需要学习的参数。执行反向传播时，反向传播没有参数适用于最大池化。只有这些设置过的超参数，可能是手动设置的，也可能是通过交叉验证设置的。



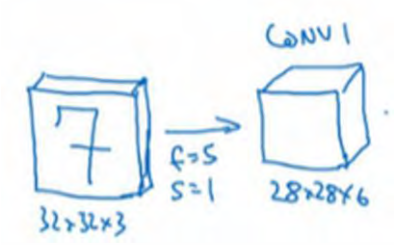
除了这些，池化的内容就全部讲完了。最大池化只是计算神经网络某一层的静态属性，没有什么需要学习的，它只是一个静态属性。

关于池化我们就讲到这儿，现在我们已经知道如何构建卷积层和池化层了。下节课，我们会分析一个更复杂的可以引进全连接层的卷积网络示例。

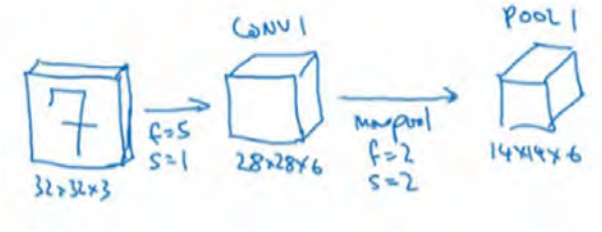
1.10 卷积神经网络示例（Convolutional neural network example）

构建全卷积神经网络的构造模块我们已经掌握得差不多了，下面来看个例子。

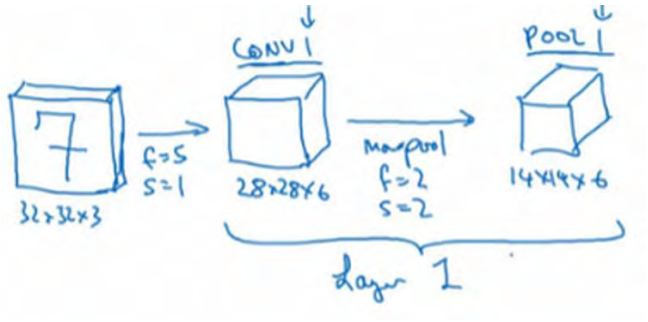
假设，有一张大小为 32×32×3 的输入图片，这是一张 **RGB** 模式的图片，你想做手写体数字识别。32×32×3 的 **RGB** 图片中含有某个数字，比如 7，你想识别它是从 0-9 这 10 个数字中的哪一个，我们构建一个神经网络来实现这个功能。



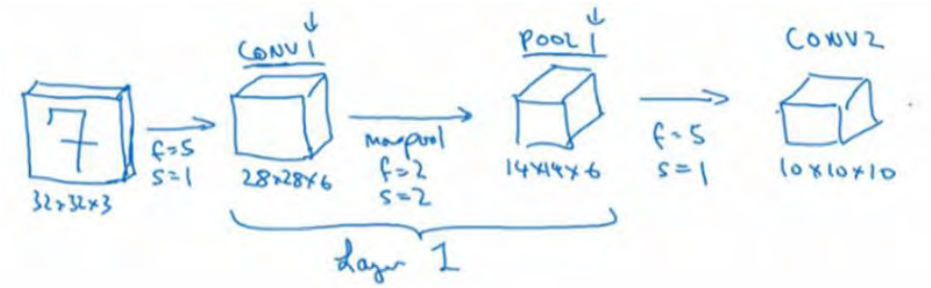
我用的这个网络模型和经典网络 **LeNet-5** 非常相似，灵感也来源于此。**LeNet-5** 是多年前 **Yann LeCun** 创建的，我所采用的模型并不是 **LeNet-5**，但是受它启发，许多参数选择都与 **LeNet-5** 相似。输入是 32×32×3 的矩阵，假设第一层使用过滤器大小为 5×5，步幅是 1，**padding** 是 0，过滤器个数为 6，那么输出为 28×28×6。将这层标记为 **CONV1**，它用了 6 个过滤器，增加了偏差，应用了非线性函数，可能是 **ReLU** 非线性函数，最后输出 **CONV1** 的结果。



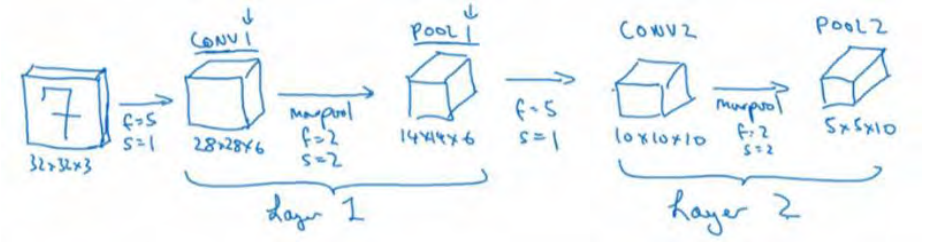
然后构建一个池化层，这里我选择用最大池化，参数$f=2$，$s=2$，因为 **padding** 为 0，我就不写出来了。现在开始构建池化层，最大池化使用的过滤器为 2×2，步幅为 2，表示层的高度和宽度会减少一半。因此，28×28 变成了 14×14，通道数量保持不变，所以最终输出为 14×14×6，将该输出标记为 **POOL1**。



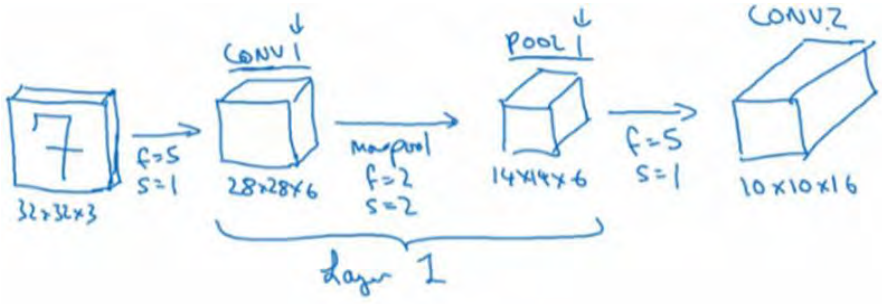
人们发现在卷积神经网络文献中，卷积有两种分类，这与所谓层的划分存在一致性。一类卷积是一个卷积层和一个池化层一起作为一层，这就是神经网络的 **Layer1**。另一类卷积是把卷积层作为一层，而池化层单独作为一层。人们在计算神经网络有多少层时，通常只统计具有权重和参数的层。因为池化层没有权重和参数，只有一些超参数。这里，我们把 **CONV1** 和 **POOL1** 共同作为一个卷积，并标记为 **Layer1**。虽然你在阅读网络文章或研究报告时，你可能会看到卷积层和池化层各为一层的情况，这只是两种不同的标记术语。一般我在统计网络层数时，只计算具有权重的层，也就是把 **CONV1** 和 **POOL1** 作为 **Layer1**。这里我们用 **CONV1** 和 **POOL1** 来标记，两者都是神经网络 **Layer1** 的一部分，**POOL1** 也被划分在 **Layer1** 中，因为它没有权重，得到的输出是 14×14×6。



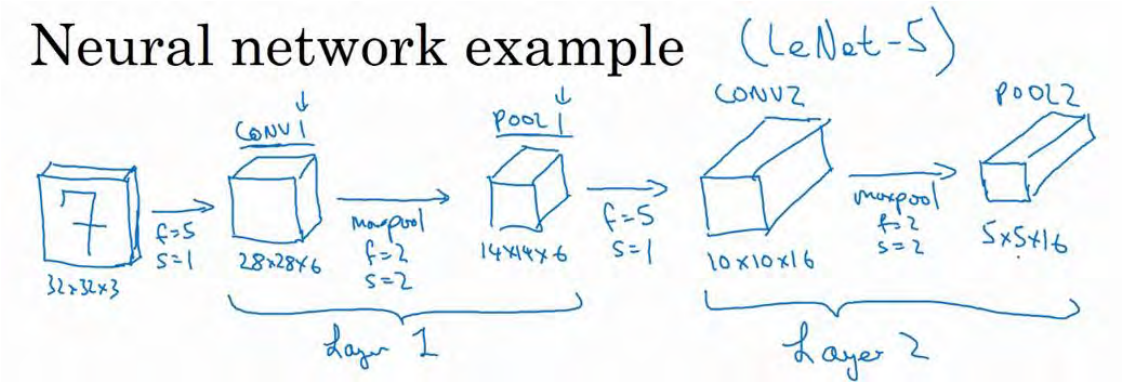
我们再为它构建一个卷积层，过滤器大小为 5×5，步幅为 1，这次我们用 10 个过滤器，最后输出一个 10×10×10 的矩阵，标记为 **CONV2**。



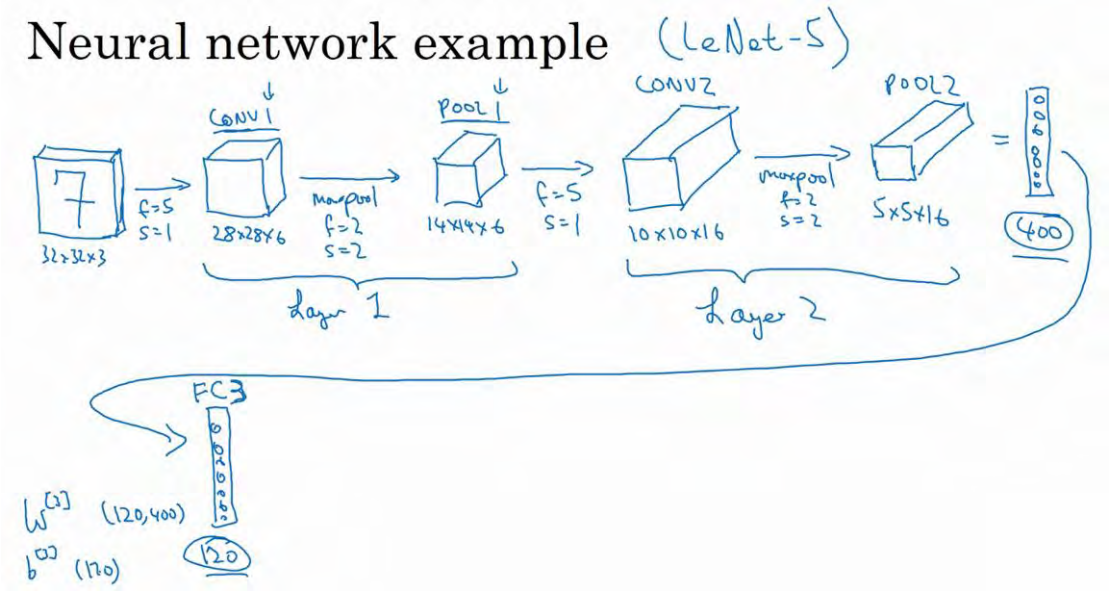
然后做最大池化，超参数$f=2$，$s=2$。你大概可以猜出结果，$f=2$，$s=2$，高度和宽度会减半，最后输出为 5×5×10，标记为 **POOL2**，这就是神经网络的第二个卷积层，即 **Layer2**。



如果对 **Layer1** 应用另一个卷积层，过滤器为 5×5，即$f=5$，步幅是 1，**padding** 为 0，所以这里省略了，过滤器 16 个，所以 **CONV2** 输出为 10×10×16。我们看看 **CONV2**，这是 **CONV2** 层。



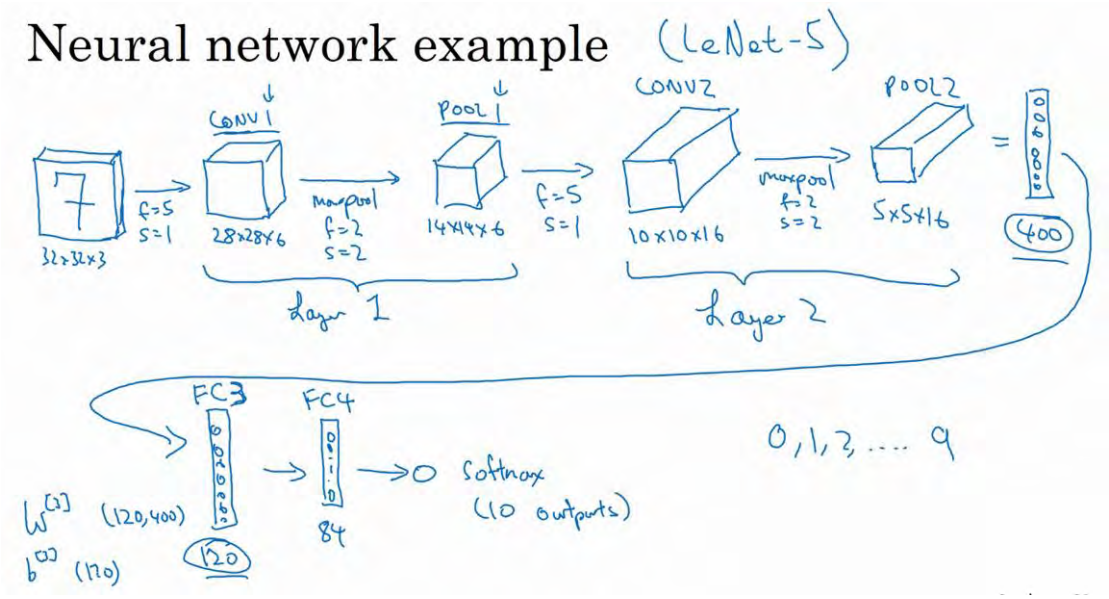
继续执行做大池化计算，参数$f=2$，$s=2$，你能猜到结果么？对 10×10×16 输入执行最大池化计算，参数$f=2$，$s=2$，高度和宽度减半，计算结果猜到了吧。最大池化的参数$f=2, s=2$，输入的高度和宽度会减半，结果为 5×5×16，通道数和之前一样，标记为 **POOL2**。这是一个卷积，即 **Layer2**，因为它只有一个权重集和一个卷积层 **CONV2**。



5×5×16 矩阵包含 400 个元素，现在将 **POOL2** 平整化为一个大小为 400 的一维向量。我

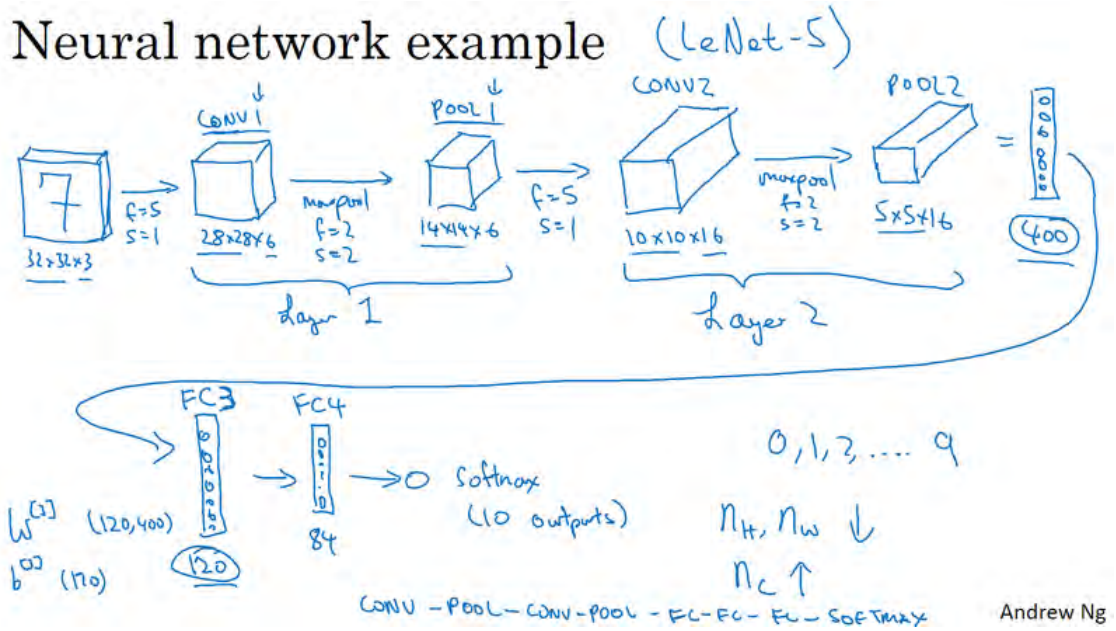
们可以把平整化结果想象成这样的一个神经元集合，然后利用这 400 个单元构建下一层。下一层含有 120 个单元，这就是我们第一个全连接层，标记为 **FC3**。这 400 个单元与 120 个单元紧密相连，这就是全连接层。它很像我们在第一和第二门课中讲过的单神经网络层，这是一个标准的神经网络。它的权重矩阵为$W^{[3]}$，维度为 120×40。这就是所谓的“全连接”，因为这 400 个单元与这 120 个单元的每一项连接，还有一个偏差参数。最后输出 120 个维度，因为有 120 个输出。

然后我们对这个 120 个单元再添加一个全连接层，这层更小，假设它含有 84 个单元，标记为 **FC4**。



最后，用这 84 个单元填充一个 **softmax** 单元。如果我们想通过手写数字识别来识别手写 0-9 这 10 个数字，这个 **softmax** 就会有 10 个输出。

此例中的卷积神经网络很典型，看上去它有很多超参数，关于如何选定这些参数，后面我提供更多建议。常规做法是，尽量不要自己设置超参数，而是查看文献中别人采用了哪些超参数，选一个在别人任务中效果很好的架构，那么它也有可能适用于你自己的应用程序，这块下周我会细讲。



现在，我想指出的是，随着神经网络深度的加深，高度n_H和宽度n_W通常都会减少，前面我就提到过，从 32×32 到 28×28，到 14×14，到 10×10，再到 5×5。所以随着层数增加，高度和宽度都会减小，而通道数量会增加，从 3 到 6 到 16 不断增加，然后得到一个全连接层。

在神经网络中，另一种常见模式就是一个或多个卷积后面跟随一个池化层，然后一个或多个卷积层后面再跟一个池化层，然后是几个全连接层，最后是一个 **softmax**。这是神经网络的另一种常见模式。

接下来我们讲讲神经网络的激活值形状，激活值大小和参数数量。输入为 32×32×3，这些数做乘法，结果为 3072，所以激活值$a^{[0]}$有 3072 维，激活值矩阵为 32×32×3，输入层没有参数。计算其他层的时候，试着自己计算出激活值，这些都是网络中不同层的激活值形状和激活值大小。

Neural network example

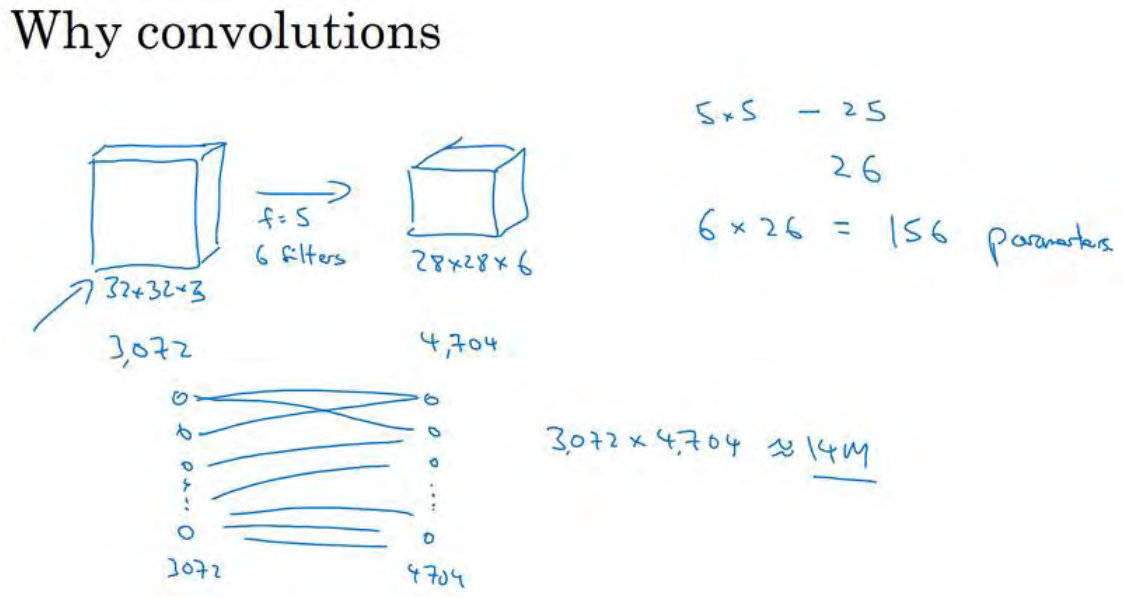
	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

有几点要注意，第一，池化层和最大池化层没有参数；第二卷积层的参数相对较少，前面课上我们提到过，其实许多参数都存在于神经网络的全连接层。观察可发现，随着神经网络的加深，激活值尺寸会逐渐变小，如果激活值尺寸下降太快，也会影响神经网络性能。示例中，激活值尺寸在第一层为 6000，然后减少到 1600，慢慢减少到 84，最后输出 **softmax** 结果。我们发现，许多卷积网络都具有这些属性，模式上也相似。

神经网络的基本构造模块我们已经讲完了，一个卷积神经网络包括卷积层、池化层和全连接层。许多计算机视觉研究正在探索如何把这些基本模块整合起来，构建高效的神经网络，整合这些基本模块确实需要深入的理解。根据我的经验，找到整合基本构造模块最好方法就是大量阅读别人的案例。下周我会演示一些整合基本模块，成功构建高效神经网络的具体案例。我希望下周的课程可以帮助你找到构建有效神经网络的感觉，或许你也可以将别人开发的框架应用于自己的应用程序，这是下周的内容。下节课，也是本周最后一节课，我想花点时间讨论下，为什么大家愿意使用卷积，使用卷积的好处和优势是什么，以及如何整合多个卷积，如何检验神经网络，如何在训练集上训练神经网络来识别图片或执行其他任务，我们下节课继续。

1.11 为什么使用卷积？（Why convolutions?）

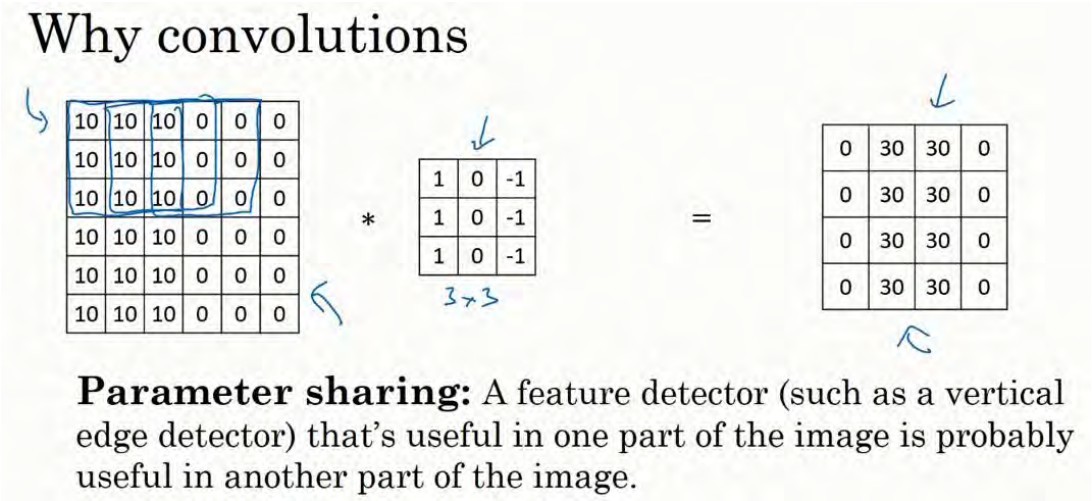
这是本周最后一节课，我们来分析一下卷积在神经网络中如此受用的原因，然后对如何整合这些卷积，如何通过一个标注过的训练集训练卷积神经网络做个简单概括。和只用全连接层相比，卷积层的两个主要优势在于参数共享和稀疏连接，举例说明一下。



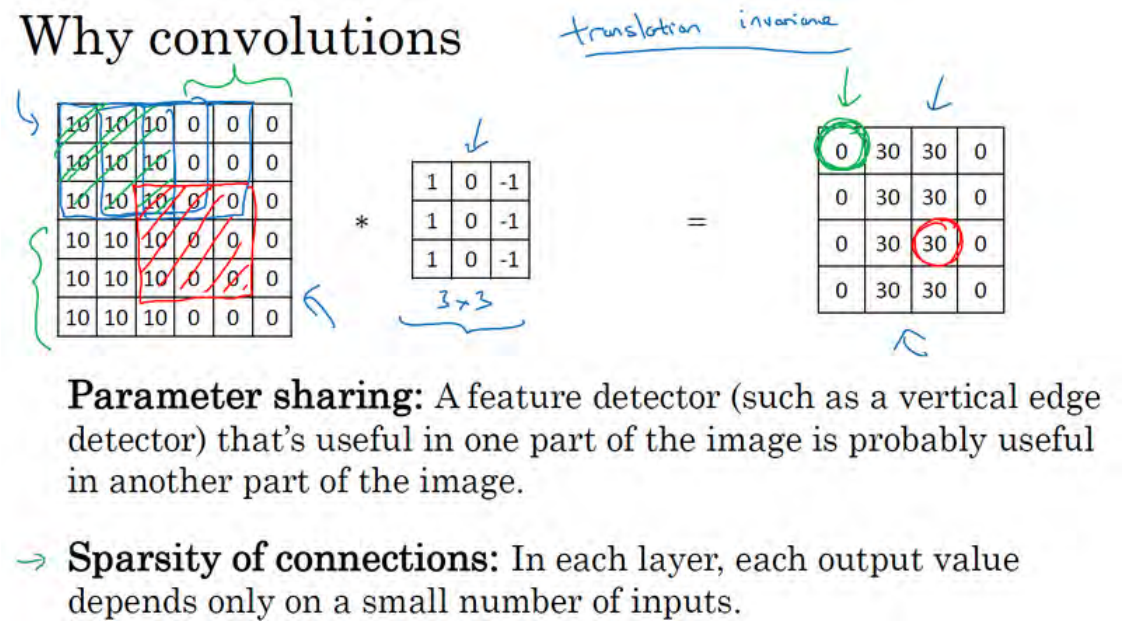
假设有一张 32×32×3 维度的图片，这是上节课的示例，假设用了 6 个大小为 5×5 的过滤器，输出维度为 28×28×6。32×32×3=3072，28×28×6=4704。我们构建一个神经网络，其中一层含有 3072 个单元，下一层含有 4074 个单元，两层中的每个神经元彼此相连，然后计算权重矩阵，它等于 4074×3072≈1400 万，所以要训练的参数很多。虽然以现在的技术，我们可以用 1400 多万个参数来训练网络，因为这张 32×32×3 的图片非常小，训练这么多参数没有问题。如果这是一张 1000×1000 的图片，权重矩阵会变得非常大。我们看看这个卷积层的参数数量，每个过滤器都是 5×5，一个过滤器有 25 个参数，再加上偏差参数，那么每个过滤器就有 26 个参数，一共有 6 个过滤器，所以参数共计 156 个，参数数量还是很少。

卷积网络映射这么少参数有两个原因，一是参数共享。观察发现，特征检测如垂直边缘检测如果适用于图片的某个区域，那么它也可能适用于图片的其他区域。也就是说，如果你用一个 3×3 的过滤器检测垂直边缘，那么图片的左上角区域，以及旁边的各个区域（左边矩阵中蓝色方框标记的部分）都可以使用这个 3×3 的过滤器。每个特征检测器以及输出都可以在输入图片的不同区域中使用同样的参数，以便提取垂直边缘或其它特征。它不仅适用于边缘特征这样的低阶特征，同样适用于高阶特征，例如提取脸上的眼睛，猫或者其他特征对象。即使减少参数个数，这 9 个参数同样能计算出 16 个输出。直观感觉是，一个特征检测器，

如垂直边缘检测器用于检测图片左上角区域的特征，这个特征很可能也适用于图片的右下角区域。因此在计算图片左上角和右下角区域时，你不需要添加其它特征检测器。假如有一个这样的数据集，其左上角和右下角可能有不同分布，也有可能稍有不同，但很相似，整张图片共享特征检测器，提取效果也很好。



使得卷积网络参数相对较少的第二个方法是使用稀疏连接，我来解释下。这个 0 是通过 3×3 的卷积计算得到的，它只依赖于这个 3×3 的输入的单元格，右边这个输出单元（元素 0）仅与 36 个输入特征中 9 个相连接。而且其它像素值都不会对输出产生任影响，这就是稀疏连接的概念。

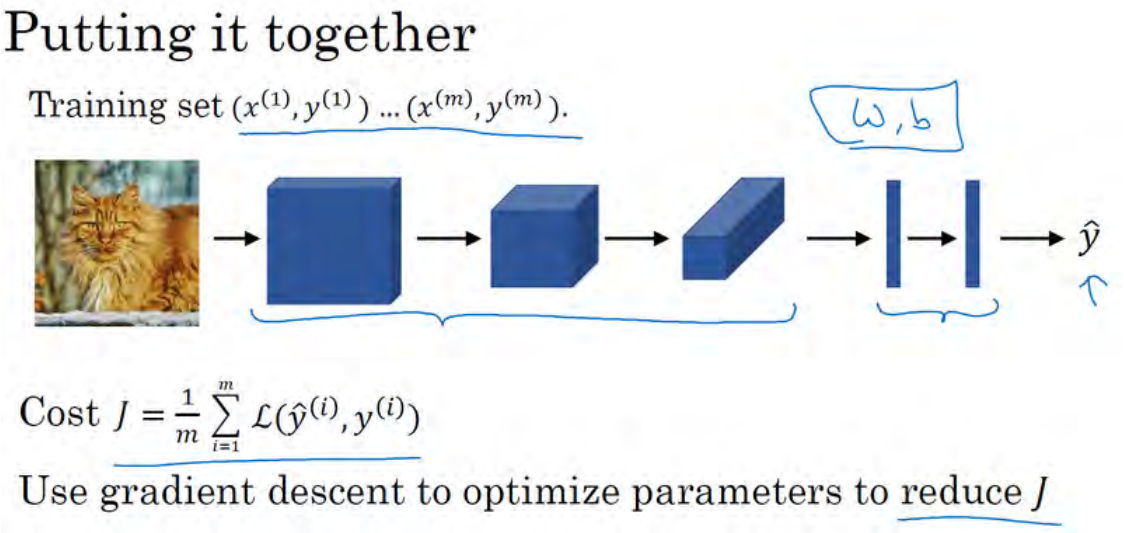


再举一个例子，这个输出（右边矩阵中红色标记的元素 30）仅仅依赖于这 9 个特征（左边矩阵红色方框标记的区域），看上去只有这 9 个输入特征与输出相连接，其它像素对输出没有任何影响。

神经网络可以通过这两种机制减少参数，以便我们用更小的训练集来训练它，从而预防

过度拟合。你们也可能听过，卷积神经网络善于捕捉平移不变。通过观察可以发现，向右移动两个像素，图片中的猫依然清晰可见，因为神经网络的卷积结构使得即使移动几个像素，这张图片依然具有非常相似的特征，应该属于同样的输出标记。实际上，我们用同一个过滤器生成各层中，图片的所有像素值，希望网络通过自动学习变得更加健壮，以便更好地取得所期望的平移不变属性。

这就是卷积或卷积网络在计算机视觉任务中表现良好的原因。



最后，我们把这些层整合起来，看看如何训练这些网络。比如我们要构建一个猫咪检测器，我们有下面这个标记训练集，x表示一张图片，$\hat{y}$是二进制标记或某个重要标记。我们选定了一个卷积神经网络，输入图片，增加卷积层和池化层，然后添加全连接层，最后输出一个 **softmax**，即$\hat{y}$。卷积层和全连接层有不同的参数w和偏差b，我们可以用任何参数集合来定义代价函数。一个类似于我们之前讲过的那种代价函数，并随机初始化其参数w和b，代价函数 J 等于神经网络对整个训练集的预测的损失总和再除以m（即$\text{Cost}\,J = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$）。所以训练神经网络，你要做的就是使用梯度下降法，或其它算法，例如 **Momentum** 梯度下降法，含 **RMSProp** 或其它因子的梯度下降来优化神经网络中所有参数，以减少代价函数J的值。通过上述操作你可以构建一个高效的猫咪检测器或其它检测器。

恭喜你完成了这一周的课程，你已经学习了卷积神经网络的所有基本构造模块，以及如何在高效图片识别系统中整合这些模块。透过本周编程练习，你可以更加具体了解这些概念，试着整合这些构造模块，并用它们解决自己的问题。

下周，我们将继续深入学习卷积神经网络。我曾提到卷积神经网络中有很多超参数，下周，我打算具体展示一些最有效的卷积神经网络示例，你也可以尝试去判断哪些网络架构类型效率更高。人们通常的做法是将别人发现和发表在研究报告上的架构应用于自己的应用程

序。下周看过更多具体的示例后，相信你会做的更好。此外，下星期我们也会深入分析卷积神经网络如此高效的原因，同时讲解一些新的计算机视觉应用程序，例如，对象检测和神经风格迁移以及如何利用这些算法创造新的艺术品形式。

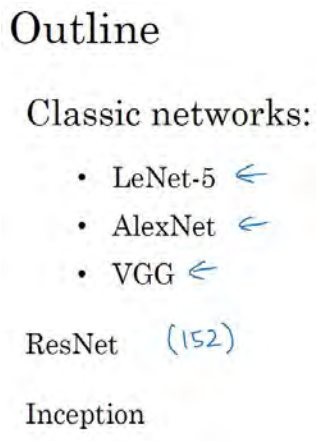
第二周 深度卷积网络：实例探究（Deep convolutional models: case studies）

2.1 为什么要进行实例探究？（Why look at case studies?）

大家好，欢迎回来。这周我们首先来看看一些卷积神经网络的实例分析，为什么要看这些实例分析呢？上周我们讲了基本构建，比如卷积层、池化层以及全连接层这些组件。事实上，过去几年计算机视觉研究中的大量研究都集中在如何把这些基本构件组合起来，形成有效的卷积神经网络。最直观的方式之一就是去看一些案例，就像很多人通过看别人的代码来学习编程一样，通过研究别人构建有效组件的案例是个不错的办法。实际上在计算机视觉任务中表现良好的神经网络框架往往也适用于其它任务，也许你的任务也不例外。也就是说，如果有人已经训练或者计算出擅长识别猫、狗、人的神经网络或者神经网络框架，而你的计算机视觉识别任务是构建一个自动驾驶汽车，你完全可以借鉴别人的神经网络框架来解决自己的问题。

最后，学完这几节课，你应该可以读一些计算机视觉方面的研究论文了，我希望这也是你学习本课程的收获。当然，读论文并不是必须的，但是我希望当你发现你可以读懂一些计算机视觉方面的研究论文或研讨会内容时会有一种满足感。言归正传，我们进入主题。

这是后面几节课的提纲，首先我们来看几个经典的网络。



LeNet-5 网络，我记得应该是 1980 年代的，经常被引用的 **AlexNet**，还有 **VGG** 网络。这些都是非常有效的神经网络范例，当中的一些思路为现代计算机视觉技术的发展奠定了基础。论文中的这些想法可能对你大有裨益，对你的工作也可能有所帮助。

然后是 **ResNet**，又称残差网络。神经网络正在不断加深，对此你可能有所了解。**ResNet**

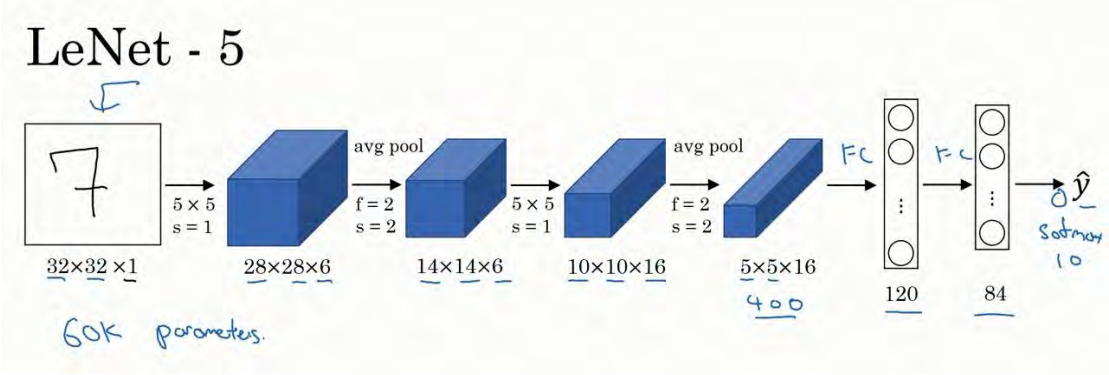
神经网络训练了一个深达 152 层的神经网络，并且在如何有效训练方面，总结出了一些有趣的想法和窍门。课程最后，我们还会讲一个 **Inception** 神经网络的实例分析。

了解了这些神经网络，我相信你会对如何构建有效的卷积神经网络更有感觉。即使计算机视觉并不是你的主要方向，但我相信你会从 **ResNet** 和 **Inception** 网络这样的实例中找到一些不错的想法。这里面有很多思路都是多学科融合的产物。总之，即便你不打算构建计算机视觉应用程序，试着从中发现一些有趣的思路，对你的工作也会有所帮助。

2.2 经典网络（Classic networks）

这节课，我们来学习几个经典的神经网络结构，分别是 **LeNet-5**、**AlexNet** 和 **VGGNet**，开始吧。

首先看看 **LeNet-5** 的网络结构，假设你有一张 32×32×1 的图片，**LeNet-5** 可以识别图中的手写数字，比如像这样手写数字 7。**LeNet-5** 是针对灰度图片训练的，所以图片的大小只有 32×32×1。实际上 **LeNet-5** 的结构和我们上周讲的最后一个范例非常相似，使用 6 个 5×5 的过滤器，步幅为 1。由于使用了 6 个过滤器，步幅为 1，padding 为 0，输出结果为 28×28×6，图像尺寸从 32×32 缩小到 28×28。然后进行池化操作，在这篇论文写成的那个年代，人们更喜欢使用平均池化，而现在我们可能用最大池化更多一些。在这个例子中，我们进行平均池化，过滤器的宽度为 2，步幅为 2，图像的尺寸，高度和宽度都缩小了 2 倍，输出结果是一个 14×14×6 的图像。我觉得这张图片应该不是完全按照比例绘制的，如果严格按照比例绘制，新图像的尺寸应该刚好是原图像的一半。



接下来是卷积层，我们用一组 16 个 5×5 的过滤器，新的输出结果有 16 个通道。**LeNet-5** 的论文是在 1998 年撰写的，当时人们并不使用 padding，或者总是使用 **valid** 卷积，这就是为什么每进行一次卷积，图像的高度和宽度都会缩小，所以这个图像从 14 到 14 缩小到了 10×10。然后又是池化层，高度和宽度再缩小一半，输出一个 5×5×16 的图像。将所有数字相乘，乘积是 400。

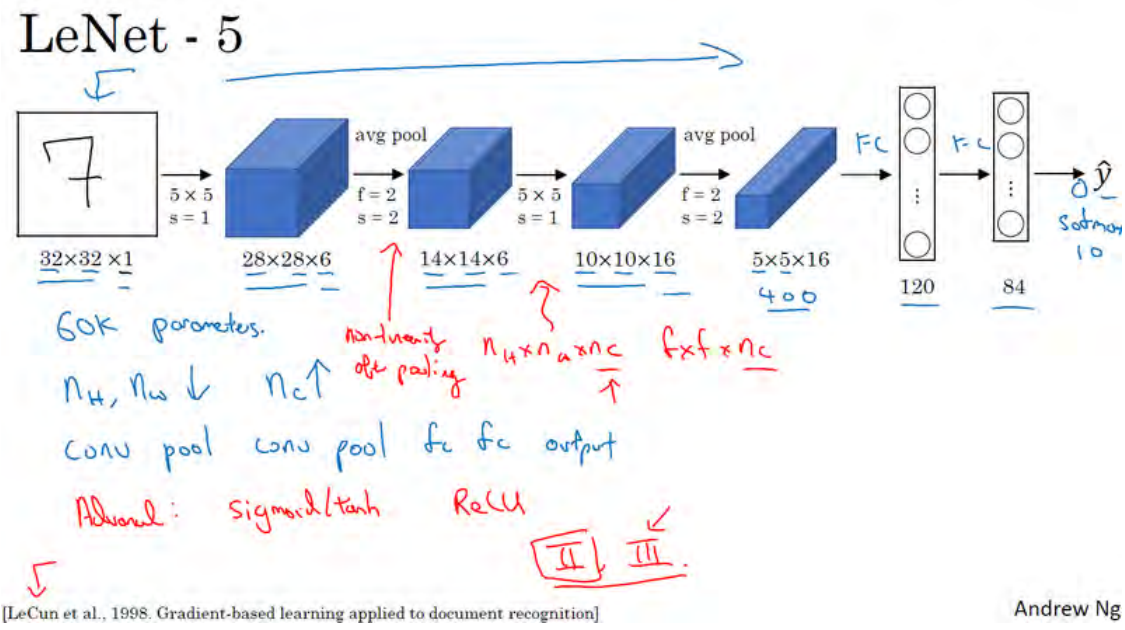
下一层是全连接层，在全连接层中，有 400 个节点，每个节点有 120 个神经元，这里已经有了一个全连接层。但有时还会从这 400 个节点中抽取一部分节点构建另一个全连接层，就像这样，有 2 个全连接层。

最后一步就是利用这 84 个特征得到最后的输出，我们还可以在这里再加一个节点用来预测$\hat{y}$的值，$\hat{y}$有 10 个可能的值，对应识别 0-9 这 10 个数字。在现在的版本中则使用 **softmax** 函数输出十种分类结果，而在当时，**LeNet-5** 网络在输出层使用了另外一种，现在已经很少

用到的分类器。

相比现代版本，这里得到的神经网络会小一些，只有约 6 万个参数。而现在，我们经常看到含有一千万到一亿个参数的神经网络，比这大 1000 倍的神经网络也不在少数。

不管怎样，如果我们从左往右看，随着网络越来越深，图像的高度和宽度在缩小，从最初的 32×32 缩小到 28×28，再到 14×14、10×10，最后只有 5×5。与此同时，随着网络层次的加深，通道数量一直在增加，从 1 增加到 6 个，再到 16 个。



这个神经网络中还有一种模式至今仍然经常用到，就是一个或多个卷积层后面跟着一个池化层，然后又是若干个卷积层再接一个池化层，然后是全连接层，最后是输出，这种排列方式很常用。

对于那些想尝试阅读论文的同学，我再补充几点。接下来的部分主要针对那些打算阅读经典论文的同学，所以会更加深入。这些内容你完全可以跳过，算是对神经网络历史的一种回顾吧，听不懂也不要紧。

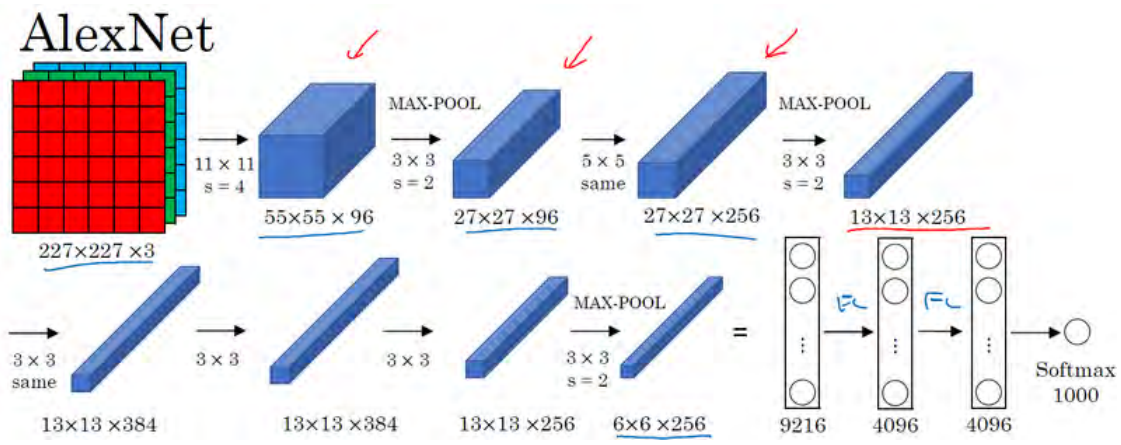
读到这篇经典论文时，你会发现，过去，人们使用 **sigmod** 函数和 **tanh** 函数，而不是 **ReLu** 函数，这篇论文中使用的正是 **sigmod** 函数和 **tanh** 函数。这种网络结构的特别之处还在于，各网络层之间是有关联的，这在今天看来显得很有趣。

比如说，你有一个$n_H \times n_W \times n_C$的网络，有$n_C$个通道，使用尺寸为 f×f×$n_C$的过滤器，每个过滤器的通道数和它上一层的通道数相同。这是由于在当时，计算机的运行速度非常慢，为了减少计算量和参数，经典的 **LeNet-5** 网络使用了非常复杂的计算方式，每个过滤器都采用和输入模块一样的通道数量。论文中提到的这些复杂细节，现在一般都不用了。

我认为当时所进行的最后一步其实到现在也还没有真正完成，就是经典的 **LeNet-5** 网络在池化后进行了非线性函数处理，在这个例子中，池化层之后使用了 **sigmod** 函数。如果你真的去读这篇论文，这会是最难理解的部分之一，我们会在后面的课程中讲到。

下面要讲的网络结构简单一些，幻灯片的大部分类容来自于原文的第二段和第三段，原文的后几段介绍了另外一种思路。文中提到的这种图形变形网络如今并没有得到广泛应用，所以在读这篇论文的时候，我建议精读第二段，这段重点介绍了这种网络结构。泛读第三段，这里面主要是一些有趣的实验结果。

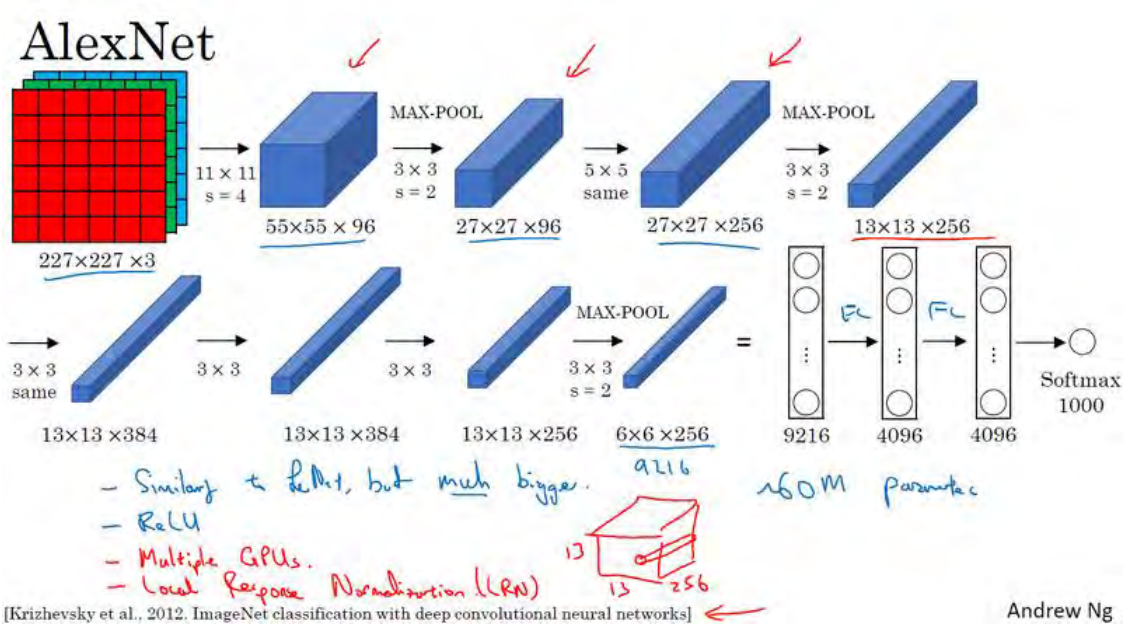
我要举例说明的第二种神经网络是 **AlexNet**，是以论文的第一作者 **Alex Krizhevsky** 的名字命名的，另外两位合著者是 **ilya Sutskever** 和 **Geoffery Hinton**。



AlexNet 首先用一张 227×227×3 的图片作为输入，实际上原文中使用的图像是 224×224×3，但是如果你尝试去推导一下，你会发现 227×227 这个尺寸更好一些。第一层我们使用 96 个 11×11 的过滤器，步幅为 4，由于步幅是 4，因此尺寸缩小到 55×55，缩小了 4 倍左右。然后用一个 3×3 的过滤器构建最大池化层，f=3，步幅*s*为 2，卷积层尺寸缩小为 27×27×96。接着再执行一个 5×5 的卷积，**padding** 之后，输出是 27×27×276。然后再次进行最大池化，尺寸缩小到 13×13。再执行一次 **same** 卷积，相同的 **padding**，得到的结果是 13×13×384，384 个过滤器。再做一次 same 卷积，就像这样。再做一次同样的操作，最后再进行一次最大池化，尺寸缩小到 6×6×256。6×6×256 等于 9216，将其展开为 9216 个单元，然后是一些全连接层。最后使用 **softmax** 函数输出识别的结果，看它究竟是 1000 个可能的对象中的哪一个。

实际上，这种神经网络与 **LeNet** 有很多相似之处，不过 **AlexNet** 要大得多。正如前面讲到的 **LeNet** 或 **LeNet-5** 大约有 6 万个参数，而 **AlexNet** 包含约 6000 万个参数。当用于训练图像和数据集时，**AlexNet** 能够处理非常相似的基本构造模块，这些模块往往包含着大量的隐藏单元或数据，这一点 **AlexNet** 表现出色。**AlexNet** 比 **LeNet** 表现更为出色的另一个原因是

它使用了 **ReLu** 激活函数。



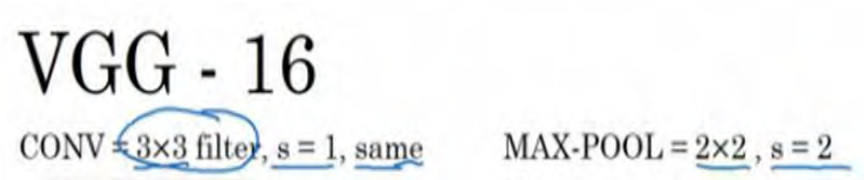
同样的，我还会讲一些比较深奥的内容，如果你并不打算阅读论文，不听也没有关系。第一点，在写这篇论文的时候，**GPU** 的处理速度还比较慢，所以 **AlexNet** 采用了非常复杂的方法在两个 **GPU** 上进行训练。大致原理是，这些层分别拆分到两个不同的 **GPU** 上，同时还专门有一个方法用于两个 **GPU** 进行交流。

论文还提到，经典的 **AlexNet** 结构还有另一种类型的层，叫作“局部响应归一化层”（**Local Response Normalization**），即 **LRN** 层，这类层应用得并不多，所以我并没有专门讲。局部响应归一层的基本思路是，假如这是网络的一块，比如是 13×13×256，**LRN** 要做的就是选取一个位置，比如说这样一个位置，从这个位置穿过整个通道，能得到 256 个数字，并进行归一化。进行局部响应归一化的动机是，对于这张 13×13 的图像中的每个位置来说，我们可能并不需要太多的高激活神经元。但是后来，很多研究者发现 **LRN** 起不到太大作用，这应该是被我划掉的内容之一，因为并不重要，而且我们现在并不用 **LRN** 来训练网络。

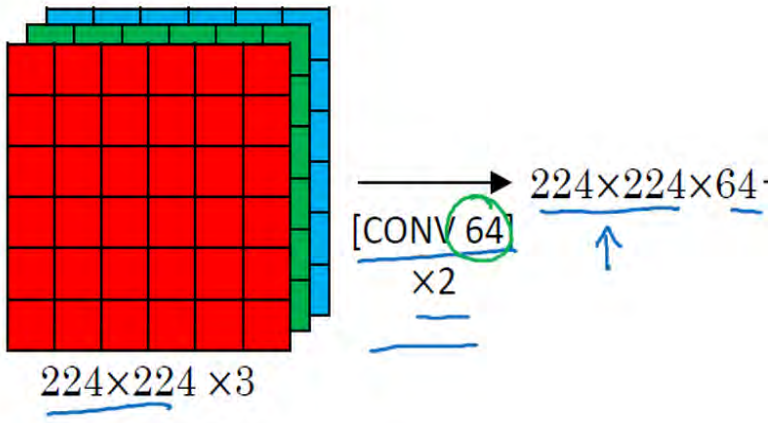
如果你对深度学习的历史感兴趣的话，我认为在 **AlexNet** 之前，深度学习已经在语音识别和其它几个领域获得了一些关注，但正是通过这篇论文，计算机视觉群体开始重视深度学习，并确信深度学习可以应用于计算机视觉领域。此后，深度学习在计算机视觉及其它领域

的影响力与日俱增。如果你并不打算阅读这方面的论文，其实可以不用学习这节课。但如果你想读懂一些相关的论文，这是比较好理解的一篇，学起来会容易一些。

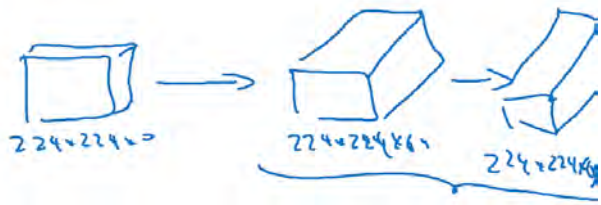
AlexNet 网络结构看起来相对复杂，包含大量超参数，这些数字（55×55×96、27×27×96、27×27×256……）都是 **Alex Krizhevsky** 及其合著者不得不给出的。



这节课要讲的第三个，也是最后一个范例是 **VGG**，也叫作 **VGG-16** 网络。值得注意的一点是，**VGG-16** 网络没有那么多超参数，这是一种只需要专注于构建卷积层的简单网络。首先用 3×3，步幅为 1 的过滤器构建卷积层，**padding** 参数为 **same** 卷积中的参数。然后用一个 2×2，步幅为 2 的过滤器构建最大池化层。因此 **VGG** 网络的一大优点是它确实简化了神经网络结构，下面我们具体讲讲这种网络结构。



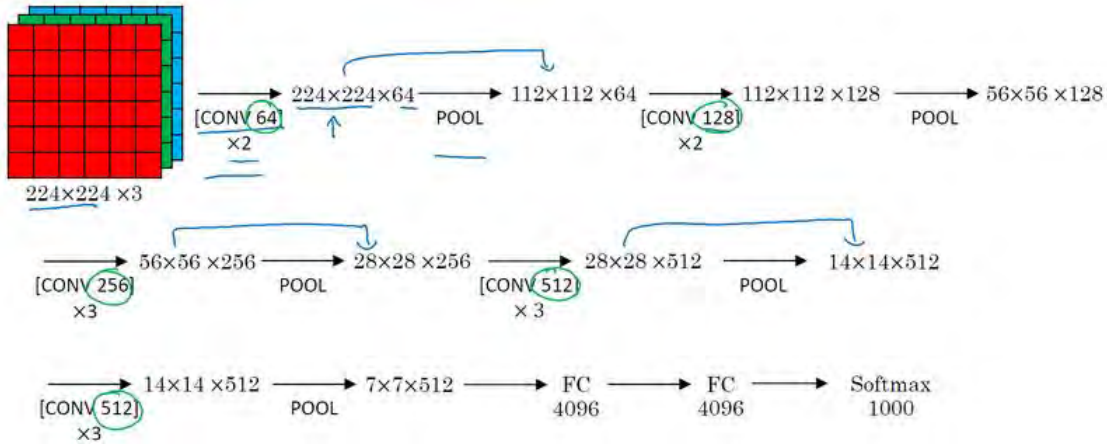
假设要识别这个图像，在最开始的两层用 64 个 3×3 的过滤器对输入图像进行卷积，输出结果是 224×224×64，因为使用了 same 卷积，通道数量也一样。**VGG-16** 其实是一个很深的网络，这里并没有把所有卷积层都画出来。



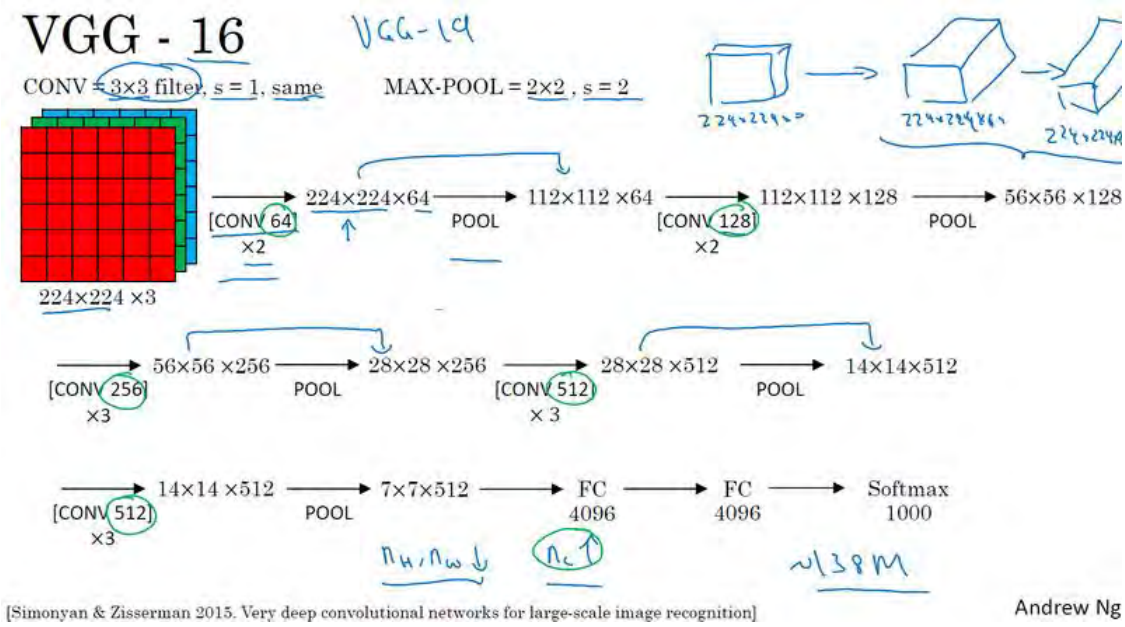
假设这个小图是我们的输入图像，尺寸是 224×224×3，进行第一个卷积之后得到 224×224×64 的特征图，接着还有一层 224×224×64，得到这样 2 个厚度为 64 的卷积层，意味着我们用 64 个过滤器进行了两次卷积。正如我在前面提到的，这里采用的都是大小为 3×3，

步幅为 1 的过滤器，并且都是采用 same 卷积，所以我就不再把所有的层都画出来了，只用一串数字代表这些网络。

接下来创建一个池化层，池化层将输入图像进行压缩，从 224×224×64 缩小到多少呢？没错，减少到 112×112×64。然后又是若干个卷积层，使用 129 个过滤器，以及一些 **same** 卷积，我们看看输出什么结果，112×112×128.然后进行池化，可以推导出池化后的结果是这样（56×56×128）。接着再用 256 个相同的过滤器进行三次卷积操作，然后再池化，然后再卷积三次，再池化。如此进行几轮操作后，将最后得到的 7×7×512 的特征图进行全连接操作，得到 4096 个单元，然后进行 **softmax** 激活，输出从 1000 个对象中识别的结果。



顺便说一下，**VGG-16** 的这个数字 16，就是指在这个网络中包含 16 个卷积层和全连接层。确实是个很大的网络，总共包含约 1.38 亿个参数，即便以现在的标准来看都算是非常大的网络。但 **VGG-16** 的结构并不复杂，这点非常吸引人，而且这种网络结构很规整，都是几个卷积层后面跟着可以压缩图像大小的池化层，池化层缩小图像的高度和宽度。同时，卷积层的过滤器数量变化存在一定的规律，由 64 翻倍变成 128，再到 256 和 512。作者可能认为 512 已经足够大了，所以后面的层就不再翻倍了。无论如何，每一步都进行翻倍，或者说在每一组卷积层进行过滤器翻倍操作，正是设计此种网络结构的另一个简单原则。这种相对一致的网络结构对研究者很有吸引力，而它的主要缺点是需要训练的特征数量非常巨大。



有些文章还介绍了 **VGG-19** 网络，它甚至比 **VGG-16** 还要大，如果你想了解更多细节，请参考幻灯片下方的注文，阅读由 **Karen Simonyan** 和 **Andrew Zisserman** 撰写的论文。由于 **VGG-16** 的表现几乎和 **VGG-19** 不分高下，所以很多人还是会使用 **VGG-16**。我最喜欢它的一点是，文中揭示了，随着网络的加深，图像的高度和宽度都在以一定的规律不断缩小，每次池化后刚好缩小一半，而通道数量在不断增加，而且刚好也是在每组卷积操作后增加一倍。也就是说，图像缩小的比例和通道数增加的比例是有规律的。从这个角度来看，这篇论文很吸引人。

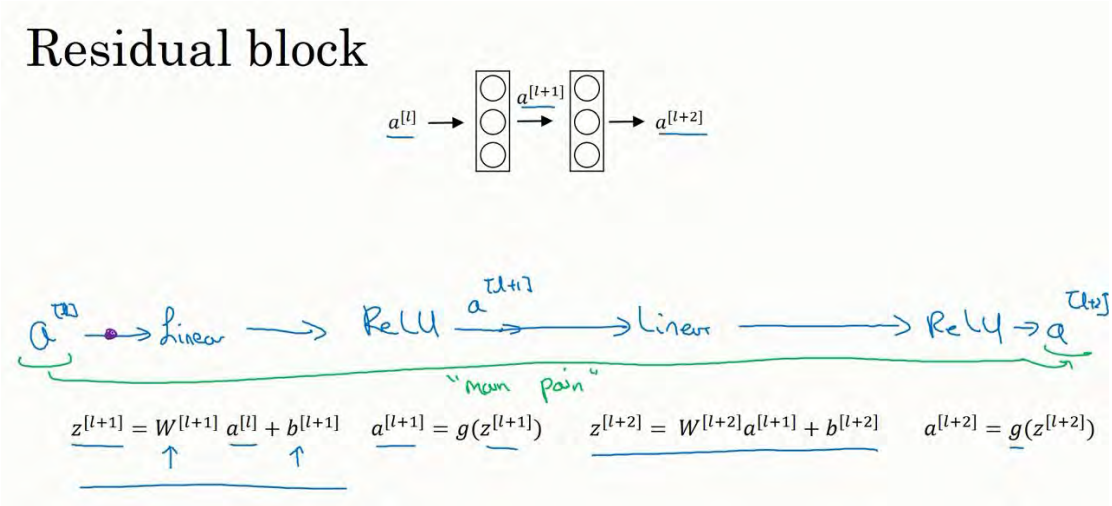
以上就是三种经典的网络结构，如果你对这些论文感兴趣，我建议从介绍 **AlexNet** 的论文开始，然后就是 **VGG** 的论文，最后是 **LeNet** 的论文。虽然有些晦涩难懂，但对于了解这些网络结构很有帮助。

学过这些经典的网络之后，下节课我们会学习一些更先高级更强大的神经网络结构，下节课见。

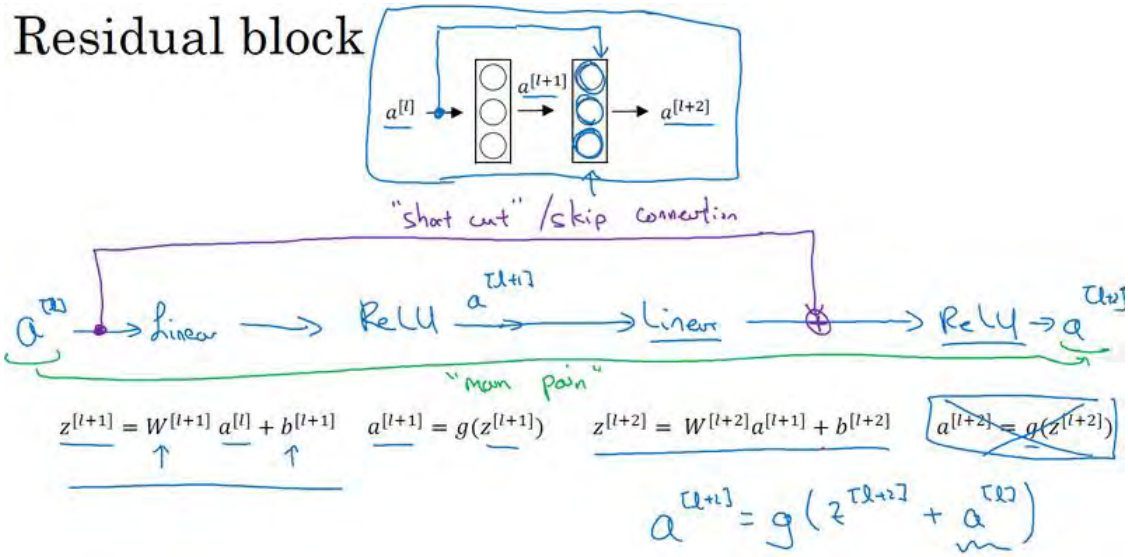
2.3 残差网络（Residual Networks (ResNets)）

非常非常深的神经网络是很难训练的，因为存在梯度消失和梯度爆炸问题。这节课我们学习跳跃连接（**Skip connection**），它可以从某一层网络层获取激活，然后迅速反馈给另外一层，甚至是神经网络的更深层。我们可以利用跳跃连接构建能够训练深度网络的 **ResNets**，有时深度能够超过 100 层，让我们开始吧。

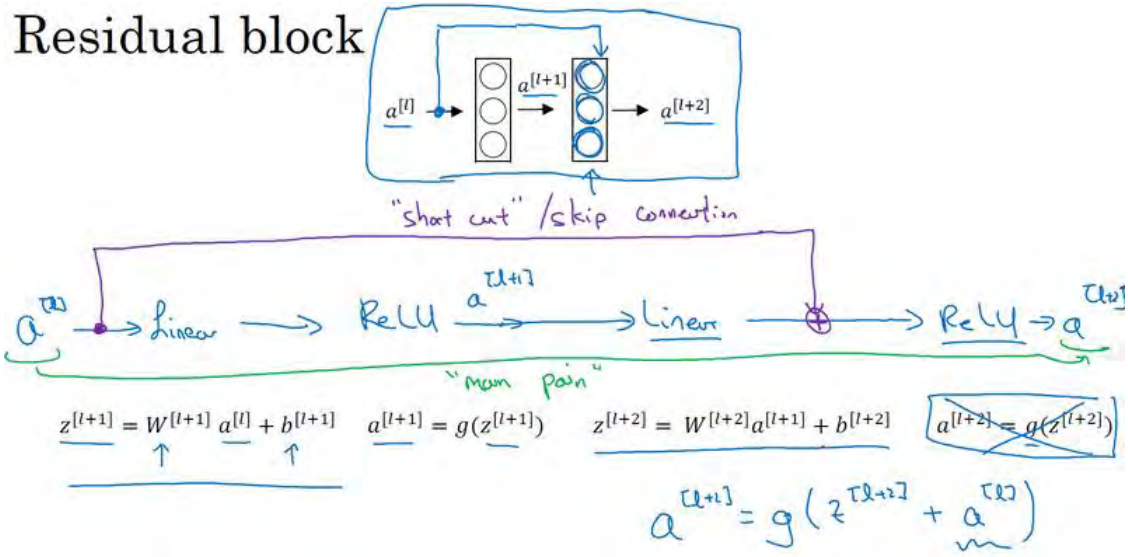
ResNets 是由残差块（**Residual block**）构建的，首先我解释一下什么是残差块。



这是一个两层神经网络，在L层进行激活，得到$a^{[l+1]}$，再次进行激活，两层之后得到$a^{[l+2]}$。计算过程是从$a^{[l]}$开始，首先进行线性激活，根据这个公式：$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]}$，通过$a^{[l]}$算出$z^{[l+1]}$，即$a^{[l]}$乘以权重矩阵，再加上偏差因子。然后通过 **ReLU** 非线性激活函数得到$a^{[l+1]}$，$a^{[l+1]} = g(z^{[l+1]})$计算得出。接着再次进行线性激活，依据等式$z^{[l+2]} = W^{[2+1]}a^{[l+1]} + b^{[l+2]}$，最后根据这个等式再次进行 ReLu 非线性激活，即$a^{[l+2]} = g(z^{[l+2]})$，这里的g是指 **ReLU** 非线性函数，得到的结果就是$a^{[l+2]}$。换句话说，信息流从$a^{[l]}$到$a^{[l+2]}$需要经过以上所有步骤，即这组网络层的主路径。



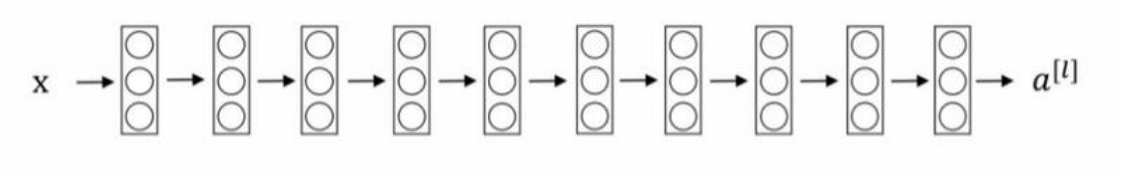
在残差网络中有一点变化，我们将$a^{[l]}$直接向后，拷贝到神经网络的深层，在 **ReLU** 非线性激活函数前加上$a^{[l]}$，这是一条捷径。$a^{[l]}$的信息直接到达神经网络的深层，不再沿着主路径传递，这就意味着最后这个等式($a^{[l+2]} = g(z^{[l+2]})$)去掉了，取而代之的是另一个 **ReLU** 非线性函数，仍然对$z^{[l+2]}$进行 g 函数处理，但这次要加上$a^{[l]}$，即： $a^{[l+2]} = g\left(z^{[l+2]} + a^{[l]}\right)$，也就是加上的这个$a^{[l]}$产生了一个残差块。



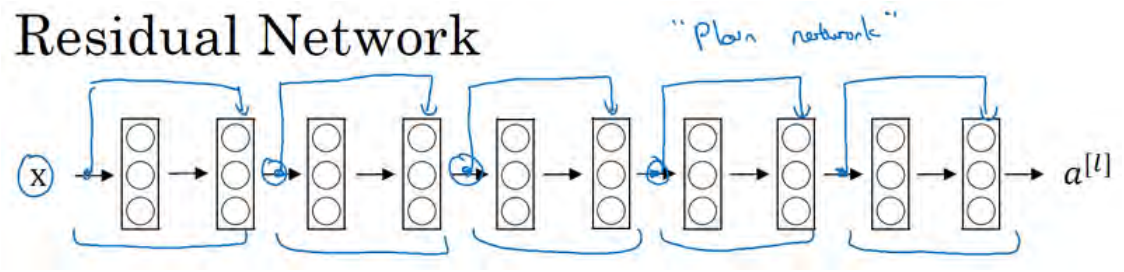
在上面这个图中，我们也可以画一条捷径，直达第二层。实际上这条捷径是在进行 **ReLU** 非线性激活函数之前加上的，而这里的每一个节点都执行了线性函数和 ReLU 激活函数。所以$a^{[l]}$插入的时机是在线性激活之后，**ReLU** 激活之前。除了捷径，你还会听到另一个术语“跳跃连接”，就是指$a^{[l]}$跳过一层或者好几层，从而将信息传递到神·经网络的更深层。

ResNet 的发明者是**何凯明（Kaiming He）**、**张翔宇（Xiangyu Zhang）**、**任少卿（Shaoqing Ren）**和**孙剑（Jiangxi Sun）**，他们发现使用残差块能够训练更深的神经网络。所以构建一个

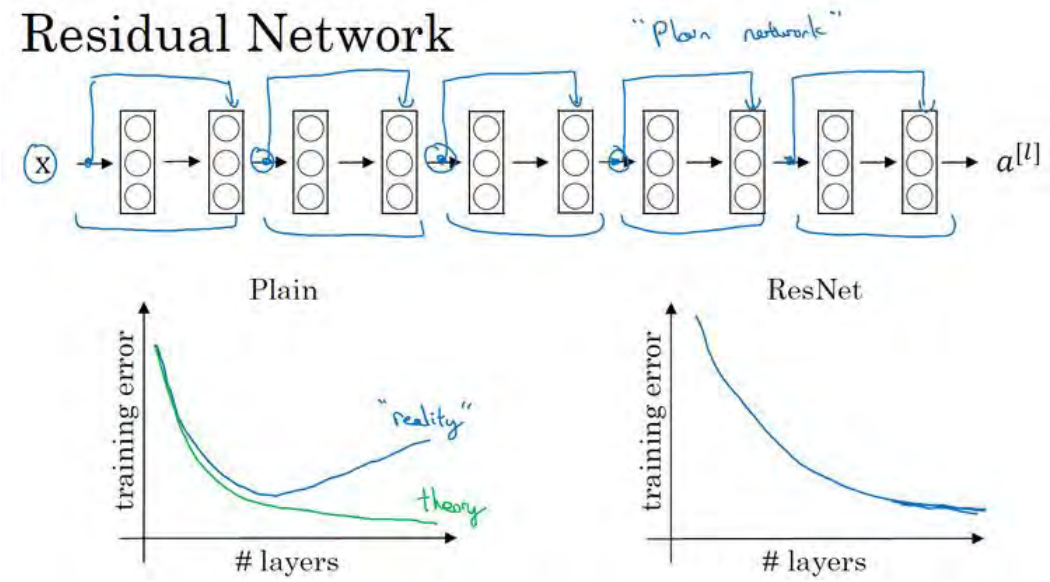
ResNet 网络就是通过将很多这样的残差块堆积在一起，形成一个很深神经网络，我们来看看这个网络。



这并不是一个残差网络，而是一个普通网络（**Plain network**），这个术语来自 **ResNet** 论文。



把它变成 **ResNet** 的方法是加上所有跳跃连接，正如前一张幻灯片中看到的，每两层增加一个捷径，构成一个残差块。如图所示，5 个残差块连接在一起构成一个残差网络。



如果我们使用标准优化算法训练一个普通网络，比如说梯度下降法，或者其它热门的优化算法。如果没有残差，没有这些捷径或者跳跃连接，凭经验你会发现随着网络深度的加深，训练错误会先减少，然后增多。而理论上，随着网络深度的加深，应该训练得越来越好才对。也就是说，理论上网络深度越深越好。但实际上，如果没有残差网络，对于一个普通网络来说，深度越深意味着用优化算法越难训练。实际上，随着网络深度的加深，训练错误会越来越多。

但有了 **ResNets** 就不一样了，即使网络再深，训练的表现却不错，比如说训练误差减少，就算是训练深达 100 层的网络也不例外。有人甚至在 1000 多层的神经网络中做过实验，尽管目前我还没有看到太多实际应用。但是对x的激活，或者这些中间的激活能够到达网络的更深层。这种方式确实有助于解决梯度消失和梯度爆炸问题，让我们在训练更深网络的同时，又能保证良好的性能。也许从另外一个角度来看，随着网络越来深，网络连接会变得臃肿，但是 **ResNet** 确实在训练深度网络方面非常有效。

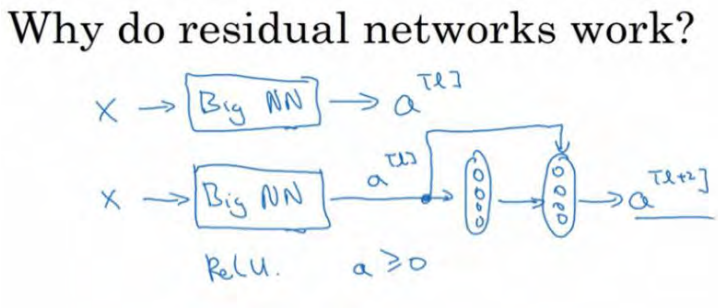
现在大家对 **ResNet** 已经有了一个大致的了解，通过本周的编程练习，你可以尝试亲自实现一下这些想法。至于为什么 **ResNets** 能有如此好的表现，接下来我会有更多更棒的内容分享给大家，我们下个视频见。

- 参考文献：Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun - Deep Residual Learning for Image Recognition (2015)

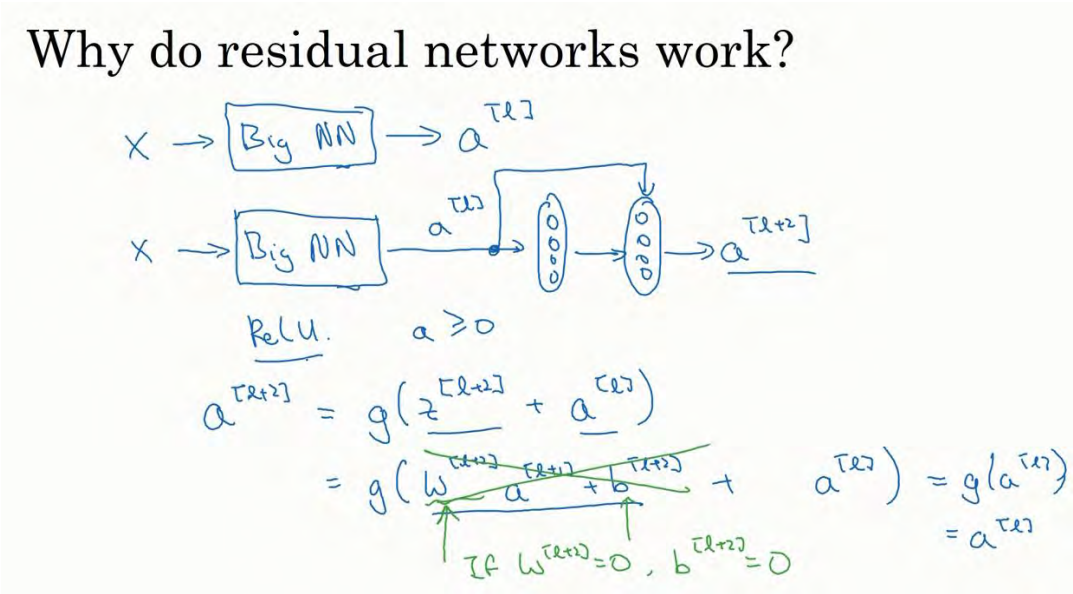
2.4 残差网络为什么有用？（Why ResNets work?）

为什么 **ResNets** 能有如此好的表现，我们来看个例子，它解释了其中的原因，至少可以说明，如何构建更深层次的 **ResNets** 网络的同时还不降低它们在训练集上的效率。希望你已经通过第三门课了解到，通常来讲，网络在训练集上表现好，才能在 **Hold-Out** 交叉验证集或 **dev** 集和测试集上有好的表现，所以至少在训练集上训练好 **ResNets** 是第一步。

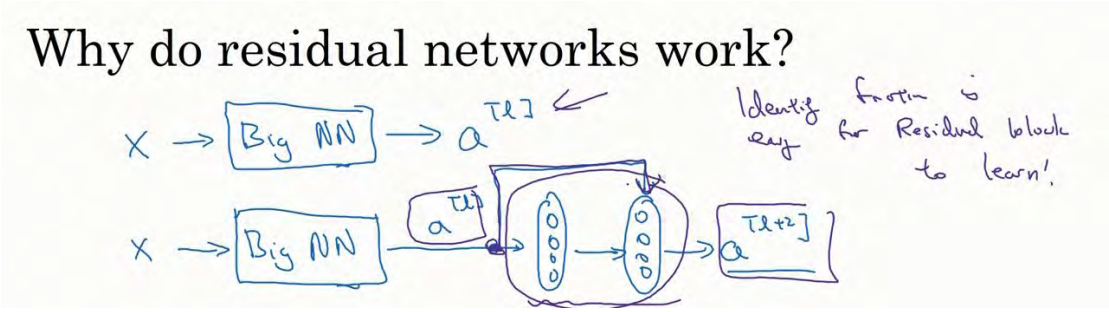
先来看个例子，上节课我们了解到，一个网络深度越深，它在训练集上训练的效率就会有所减弱，这也是有时候我们不希望加深网络的原因。而事实并非如此，至少在训练 **ResNets** 网络时，并非完全如此，举个例子。



假设有一个大型神经网络，其输入为X，输出激活值$a^{[l]}$。假如你想增加这个神经网络的深度，那么用 **Big NN** 表示，输出为$a^{[l]}$。再给这个网络额外添加两层，依次添加两层，最后输出为$a^{[l+2]}$，可以把这两层看作一个 **ResNets** 块，即具有捷径连接的残差块。为了方便说明，假设我们在整个网络中使用 **ReLU** 激活函数，所以激活值都大于等于 0，包括输入X的非零异常值。因为 **ReLU** 激活函数输出的数字要么是 0，要么是正数。



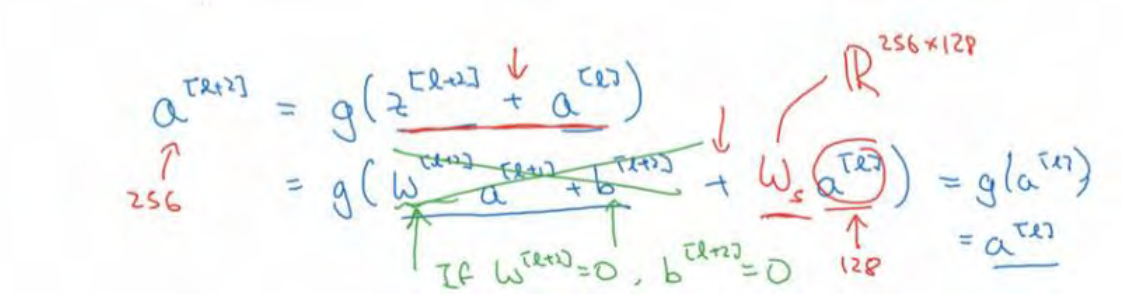
我们看一下$a^{[l+2]}$的值，也就是上节课讲过的表达式，即$a^{[l+2]}=g(z^{[l+2]}+a^{[l]})$，添加项$a^{[l]}$是刚添加的跳跃连接的输入。展开这个表达式$a^{[l+2]}=g(W^{[l+2]}a^{[l+1]}+b^{[l+2]}+a^{[l]})$，其中$z^{[l+2]}=W^{[l+2]}a^{[l+1]}+b^{[l+2]}$。注意一点，如果使用$L2$正则化或权重衰减，它会压缩$W^{[l+2]}$的值。如果对$b$应用权重衰减也可达到同样的效果，尽管实际应用中，你有时会对b应用权重衰减，有时不会。这里的W是关键项，如果$W^{[l+2]}=0$，为方便起见，假设$b^{[l+2]}=0$，这几项就没有了，因为它们（$W^{[l+2]}a^{[l+1]}+b^{[l+2]}$）的值为0。最后$a^{[l+2]}=g(a^{[l]})=a^{[l]}$，因为我们假定使用 **ReLU** 激活函数，并且所有激活值都是非负的，$g(a^{[l]})$是应用于非负数的 **ReLU** 函数，所以$a^{[l+2]}=a^{[l]}$。



结果表明，残差块学习这个恒等式函数并不难，跳跃连接使我们很容易得出$a^{[l+2]}=a^{[l]}$。这意味着，即使给神经网络增加了这两层，它的效率也并不逊色于更简单的神经网络，因为学习恒等函数对它来说很简单。尽管它多了两层，也只把$a^{[l]}$的值赋值给$a^{[l+2]}$。所以给大型神经网络增加两层，不论是把残差块添加到神经网络的中间还是末端位置，都不会影响网络的表现。

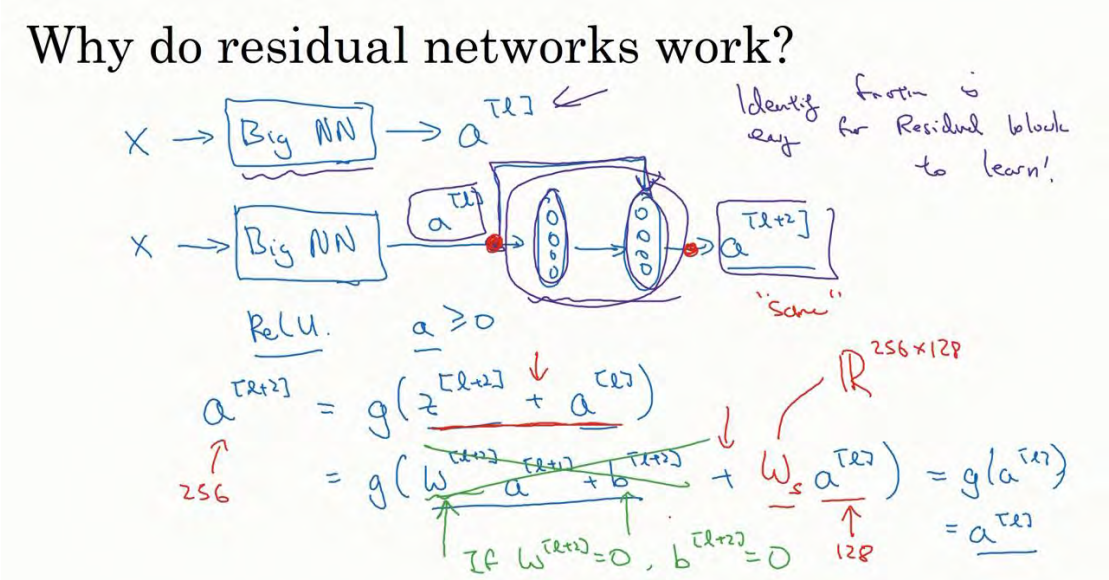
当然，我们的目标不仅仅是保持网络的效率，还要提升它的效率。想象一下，如果这些隐藏层单元学到一些有用信息，那么它可能比学习恒等函数表现得更好。而这些不含有残差块或跳跃连接的深度普通网络情况就不一样了，当网络不断加深时，就算是选用学习恒等函数的参数都很困难，所以很多层最后的表现不但没有更好，反而更糟。

我认为残差网络起作用的主要原因就是这些残差块学习恒等函数非常容易，你能确定网络性能不会受到影响，很多时候甚至可以提高效率，或者说至少不会降低网络的效率，因此创建类似残差网络可以提升网络性能。

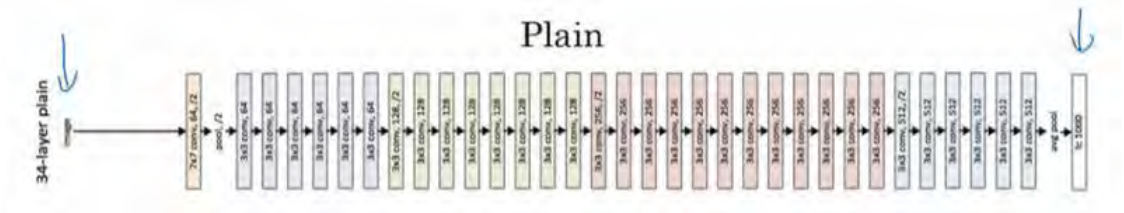


除此之外，关于残差网络，另一个值得探讨的细节是，假设$z^{[l+2]}$与$a^{[l]}$具有相同维度，所以 **ResNets** 使用了许多 **same** 卷积，所以这个$a^{[l]}$的维度等于这个输出层的维度。之所以能实现跳跃连接是因为 **same** 卷积保留了维度，所以很容易得出这个捷径连接，并输出这两个相同维度的向量。

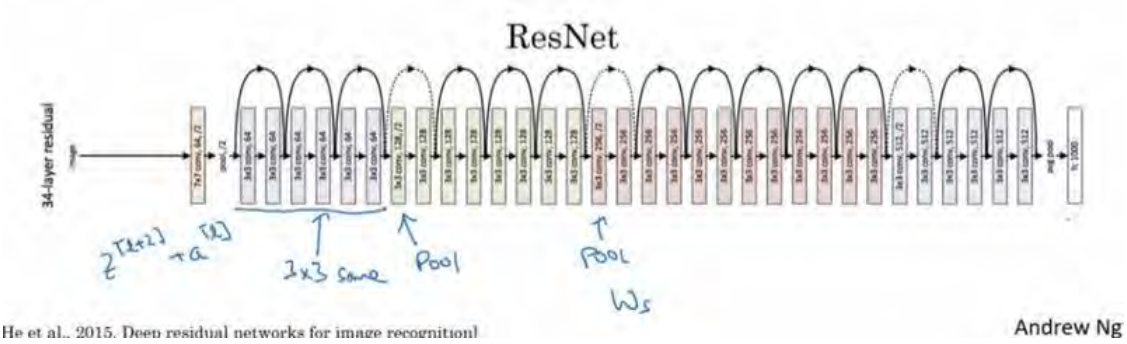
如果输入和输出有不同维度，比如输入的维度是 128，$a^{[l+2]}$的维度是 256，再增加一个矩阵，这里标记为W_s，W_s是一个 256×128 维度的矩阵，所以$W_s a^{[l]}$的维度是 256，这个新增项是 256 维度的向量。你不需要对W_s做任何操作，它是网络通过学习得到的矩阵或参数，它是一个固定矩阵，padding 值为 0，用 0 填充$a^{[l]}$，其维度为 256，所以者几个表达式都可以。



最后，我们来看看 **ResNets** 的图片识别。这些图片是我从何凯明等人论文中截取的，这是一个普通网络，我们给它输入一张图片，它有多个卷积层，最后输出了一个 **Softmax**。



如何把它转化为 **ResNets** 呢？只需要添加跳跃连接。这里我们只讨论几个细节，这个网络有很多层 3×3 卷积，而且它们大多都是 **same** 卷积，这就是添加等维特征向量的原因。所以这些都是卷积层，而不是全连接层，因为它们是 **same** 卷积，维度得以保留，这也解释了添加项$z^{[l+2]}+a^{[l]}$（维度相同所以能够相加）。



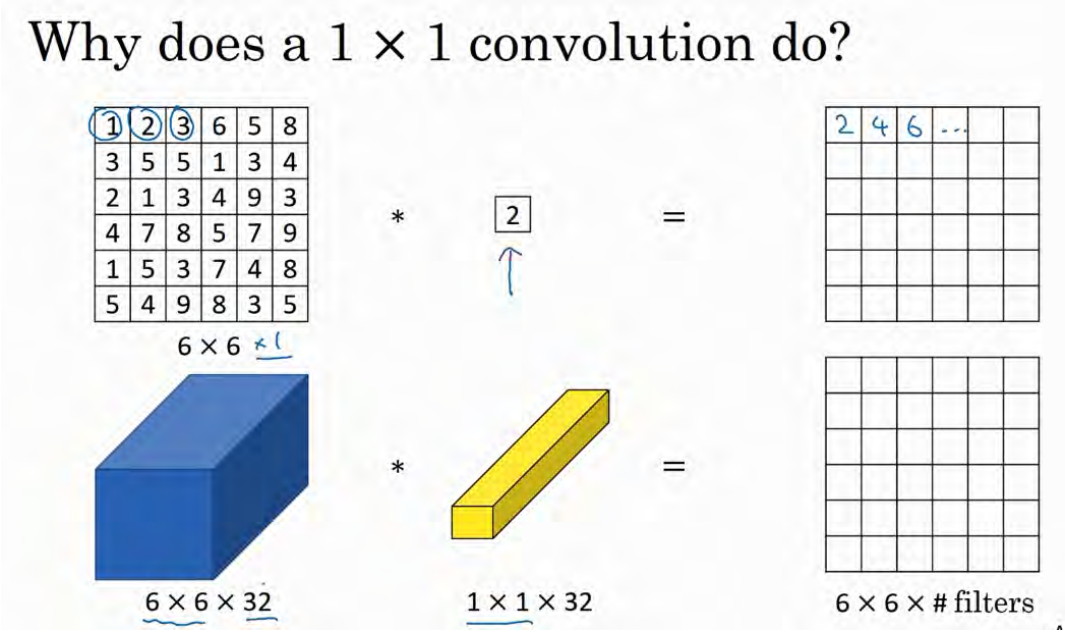
ResNets 类似于其它很多网络，也会有很多卷积层，其中偶尔会有池化层或类池化层的层。不论这些层是什么类型，正如我们在上一张幻灯片看到的，你都需要调整矩阵W_s的维度。普通网络和 **ResNets** 网络常用的结构是：卷积层-卷积层-卷积层-池化层-卷积层-卷积层-卷积层-池化层……依此重复。直到最后，有一个通过 **softmax** 进行预测的全连接层。

以上就是 **ResNets** 的内容。

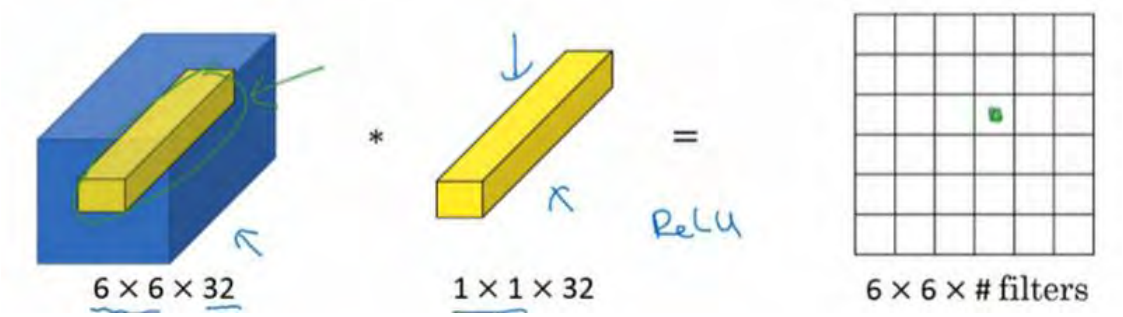
2.5 网络中的网络以及 1×1 卷积（Network in Network and 1×1 convolutions）

在架构内容设计方面，其中一个比较有帮助的想法是使用 1×1 卷积。也许你会好奇，1×1 的卷积能做什么呢？不就是乘以数字么？听上去挺好笑的，结果并非如此，我们来具体看看。

过滤器为 1×1，这里是数字 2，输入一张 6×6×1 的图片，然后对它做卷积，起过滤器大小为 1×1×1，结果相当于把这个图片乘以数字 2，所以前三个单元格分别是 2、4、6 等等。用 1×1 的过滤器进行卷积，似乎用处不大，只是对输入矩阵乘以某个数字。但这仅仅是对于 6×6×1 的一个通道图片来说，1×1 卷积效果不佳。

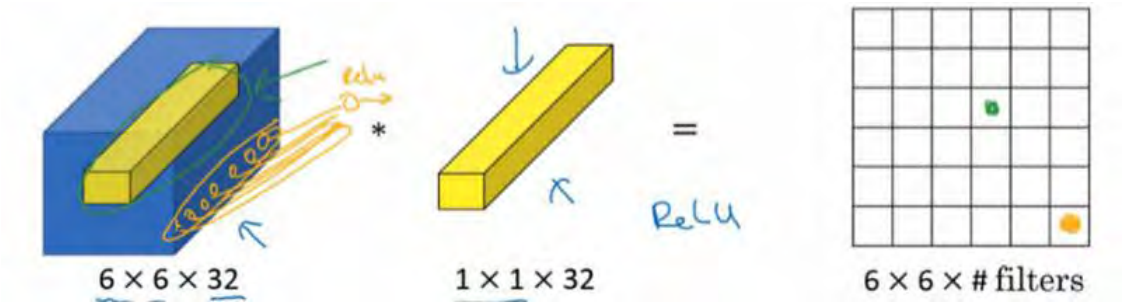


如果是一张 6×6×32 的图片，那么使用 1×1 过滤器进行卷积效果更好。具体来说，1×1 卷积所实现的功能是遍历这 36 个单元格，计算左图中 32 个数字和过滤器中 32 个数字的元素积之和，然后应用 ReLU 非线性函数。

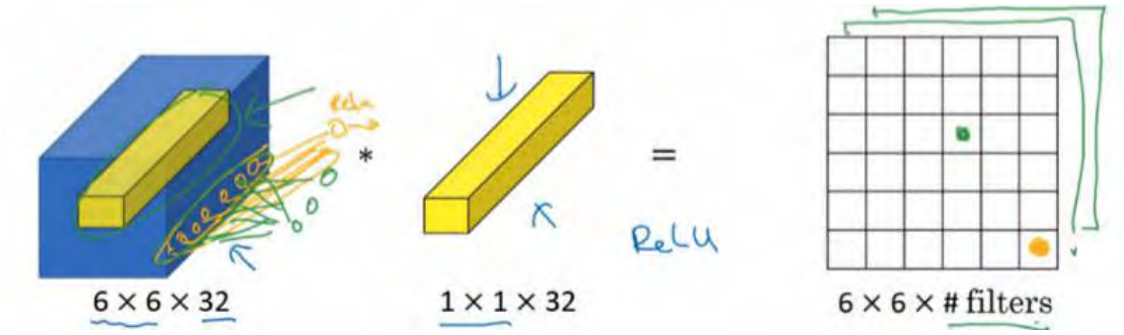


我们以其中一个单元为例，它是这个输入层上的某个切片，用这 36 个数字乘以这个输入层上 1×1 切片，得到一个实数，像这样把它画在输出中。

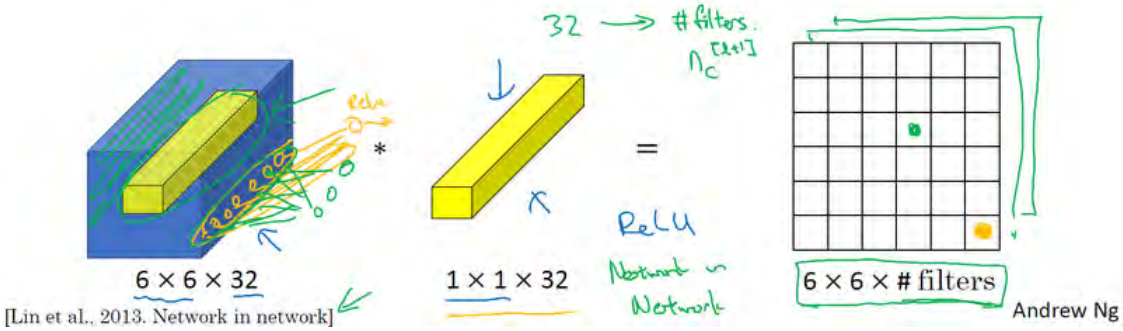
这个 1×1×32 过滤器中的 32 个数字可以这样理解，一个神经元的输入是 32 个数字（输入图片中左下角位置 32 个通道中的数字），即相同高度和宽度上某一切片上的 32 个数字，这 32 个数字具有不同通道，乘以 32 个权重（将过滤器中的 32 个数理解为权重），然后应用 ReLU 非线性函数，在这里输出相应的结果。



一般来说，如果过滤器不止一个，而是多个，就好像有多个输入单元，其输入内容为一个切片上所有数字，输出结果是 6×6 过滤器数量。



所以 1×1 卷积可以从根本上理解为对这 32 个不同的位置都应用一个全连接层，全连接层的作用是输入 32 个数字（过滤器数量标记为$n_c^{[l+1]}$，在这 36 个单元上重复此过程）,输出结果是 6×6×#filters（过滤器数量），以便在输入层上实施一个非平凡（**non-trivial**）计算。

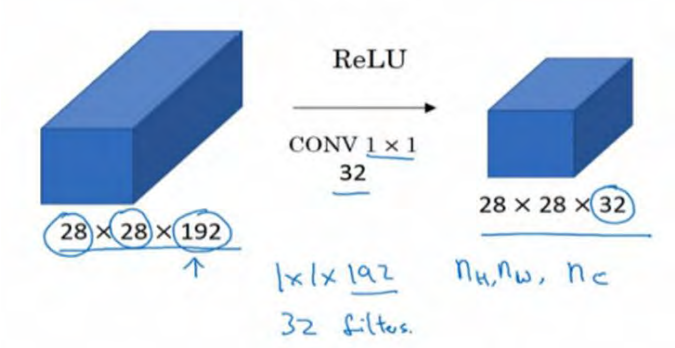


这种方法通常称为 1×1 卷积，有时也被称为 **Network in Network**，在林敏、陈强和杨学成的论文中有详细描述。虽然论文中关于架构的详细内容并没有得到广泛应用，但是 1×1 卷积或 **Network in Network** 这种理念却很有影响力，很多神经网络架构都受到它的影响，包括

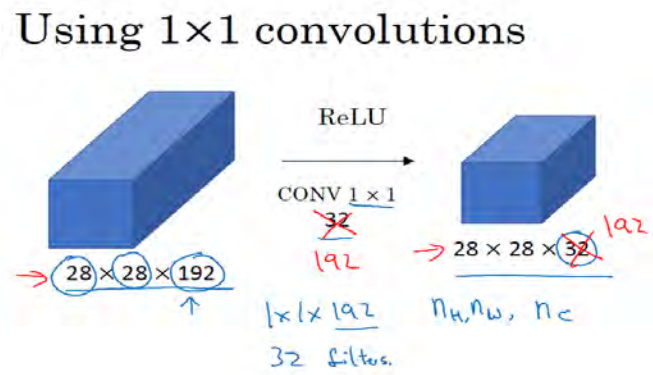
下节课要讲的 Inception 网络。

举个 1×1 卷积的例子，相信对大家有所帮助，这是它的一个应用。

假设这是一个 28×28×192 的输入层，你可以使用池化层压缩它的高度和宽度，这个过程我们很清楚。但如果通道数量很大，该如何把它压缩为 28×28×32 维度的层呢？你可以用 32 个大小为 1×1 的过滤器，严格来讲每个过滤器大小都是 1×1×192 维，因为过滤器中通道数量必须与输入层中通道的数量保持一致。但是你使用了 32 个过滤器，输出层为 28×28×32，这就是压缩通道数（n_c）的方法，对于池化层我只是压缩了这些层的高度和宽度。



在之后我们看到在某些网络中 1×1 卷积是如何压缩通道数量并减少计算的。当然如果你想保持通道数 192 不变，这也是可行的，1×1 卷积只是添加了非线性函数，当然也可以让网络学习更复杂的函数，比如，我们再添加一层，其输入为 28×28×192，输出为 28×28×192。



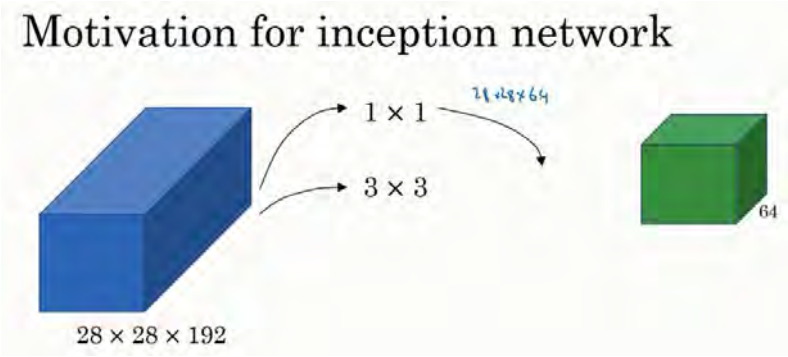
1×1 卷积层就是这样实现了一些重要功能的（**doing something pretty non-trivial**），它给神经网络添加了一个非线性函数，从而减少或保持输入层中的通道数量不变，当然如果你愿意，也可以增加通道数量。后面你会发现这对构建 **Inception** 网络很有帮助，我们放在下节课讲。

这节课我们演示了如何根据自己的意愿通过 1×1 卷积的简单操作来压缩或保持输入层中的通道数量，甚至是增加通道数量。下节课，我们将要讲解 1×1 卷积是如何帮助我们构建 **Inception** 网络。

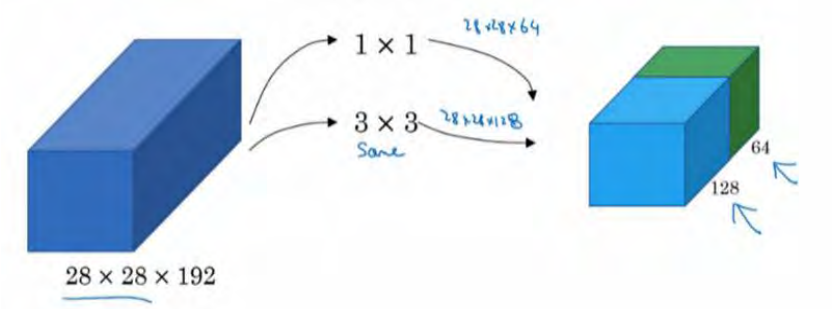
2.6 谷歌 Inception 网络简介（Inception network motivation）

构建卷积层时，你要决定过滤器的大小究竟是 1×1（原来是 1×3，猜测为口误），3×3 还是 5×5，或者要不要添加池化层。而 Inception 网络的作用就是代替你来决定，虽然网络架构因此变得更加复杂，但网络表现却非常好，我们来了解一下其中的原理。

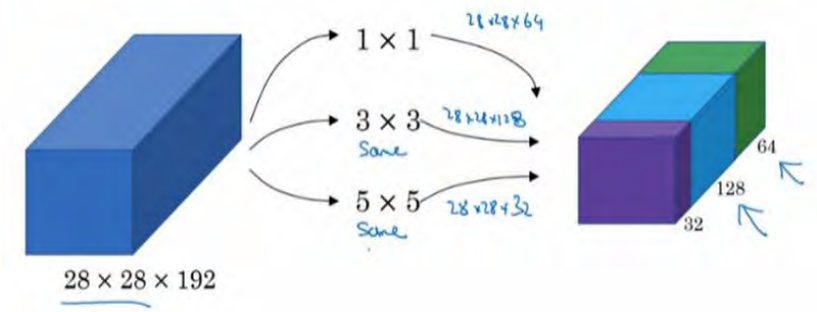
例如，这是你 28×28×192 维度的输入层，Inception 网络或 Inception 层的作用就是代替人工来确定卷积层中的过滤器类型，或者确定是否需要创建卷积层或池化层，我们演示一下。



如果使用 1×1 卷积，输出结果会是 28×28×#（某个值），假设输出为 28×28×64，并且这里只有一个层。

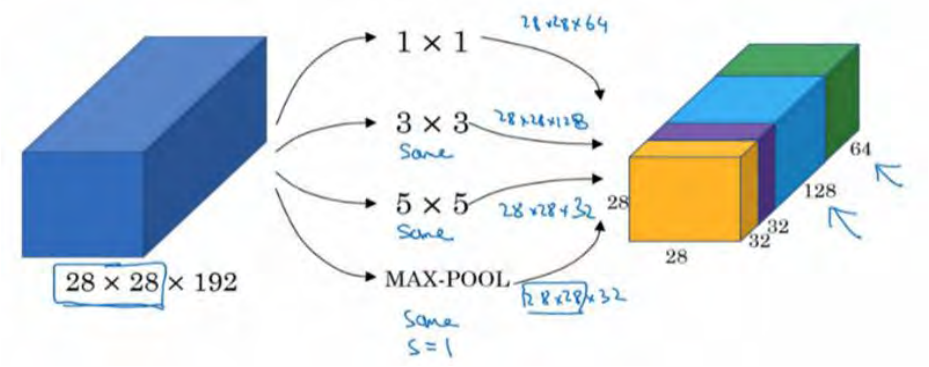


如果使用 3×3 的过滤器，那么输出是 28×28×128。然后我们把第二个值堆积到第一个值上，为了匹配维度，我们应用 **same** 卷积，输出维度依然是 28×28，和输入维度相同，即高度和宽度相同。



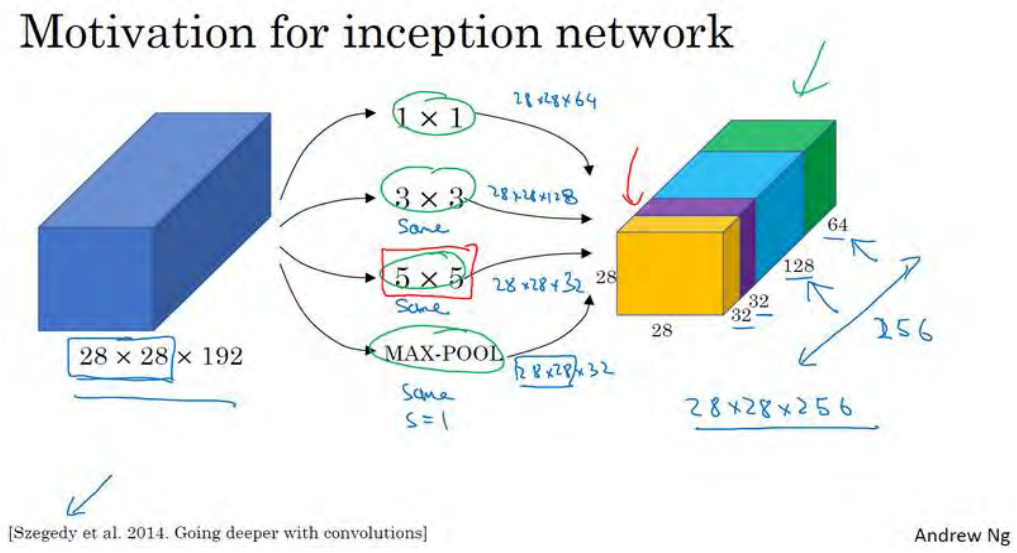
或许你会说，我希望提升网络的表现，用 5×5 过滤器或许会更好，我们不妨试一下，输

出变成 28×28×32，我们再次使用 same 卷积，保持维度不变。



或许你不想要卷积层，那就用池化操作，得到一些不同的输出结果，我们把它也堆积起来，这里的池化输出是 28×28×32。为了匹配所有维度，我们需要对最大池化使用 **padding**，它是一种特殊的池化形式，因为如果输入的高度和宽度为 28×28，则输出的相应维度也是 28×28。然后再进行池化，**padding** 不变，步幅为 1。

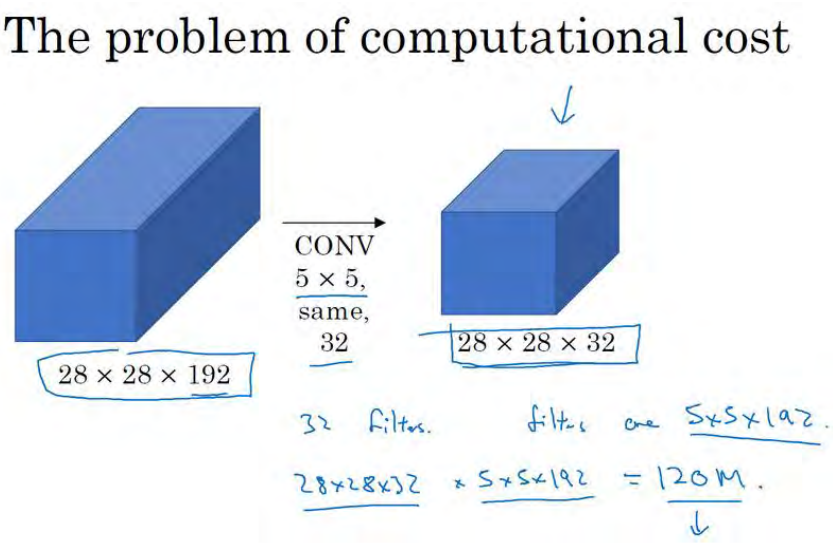
这个操作非常有意思，但我们要继续学习后面的内容，一会再实现这个池化过程。



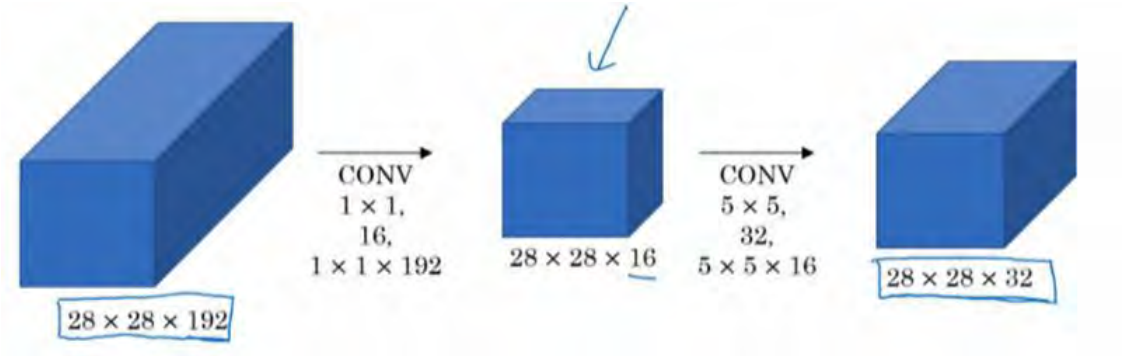
有了这样的 **Inception** 模块，你就可以输入某个量，因为它累加了所有数字，这里的最终输出为 32+32+128+64=256。**Inception** 模块的输入为 28×28×129，输出为 28×28×256。这就是 **Inception** 网络的核心内容，提出者包括 **Christian Szegedy、刘伟、贾阳青、Pierre Sermanet、Scott Reed、Dragomir Anguelov、Dumitru Erhan、Vincent Vanhoucke** 和 **Andrew Rabinovich**。基本思想是 Inception 网络不需要人为决定使用哪个过滤器或者是否需要池化，而是由网络自行确定这些参数，你可以给网络添加这些参数的所有可能值，然后把这些输出连接起来，让网络自己学习它需要什么样的参数，采用哪些过滤器组合。

不难发现，我所描述的 **Inception** 层有一个问题，就是计算成本，下一张幻灯片，我们

就来计算这个 5×5 过滤器在该模块中的计算成本。

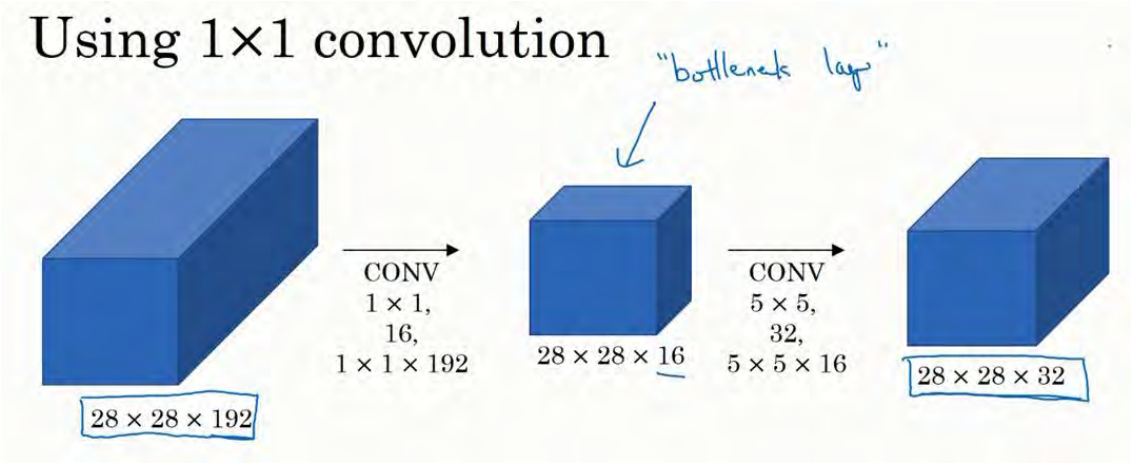


我们把重点集中在前一张幻灯片中的 5×5 的过滤器，这是一个 28×28×192 的输入块，执行一个 5×5 卷积，它有 32 个过滤器，输出为 28×28×32。前一张幻灯片中，我用一个紫色的细长块表示，这里我用一个看起来更普通的蓝色块表示。我们来计算这个 28×28×32 输出的计算成本，它有 32 个过滤器，因为输出有 32 个通道，每个过滤器大小为 5×5×192，输出大小为 28×28×32，所以你要计算 28×28×32 个数字。对于输出中的每个数字来说，你都需要执行 5×5×192 次乘法运算，所以乘法运算的总次数为每个输出值所需要执行的乘法运算次数（5×5×192）乘以输出值个数（28×28×32），把这些数相乘结果等于 1.2 亿。即使在现在，用计算机执行 1.2 亿次乘法运算，成本也是相当高的。下一张幻灯片会介绍 1×1 卷积的应用，也就是我们上节课所学的。为了降低计算成本，我们用计算成本除以因子 10，结果它从 1.2 亿减小到原来的十分之一。请记住 120 这个数字，一会还要和下一页看到的数字做对比。



这里还有另外一种架构，其输入为 28×28×192，输出为 28×28×32。其结果是这样的，对于输入层，使用 1×1 卷积把输入值从 192 个通道减少到 16 个通道。然后对这个较小层运行 5×5 卷积，得到最终输出。请注意，输入和输出的维度依然相同，输入是 28×28×192，输出

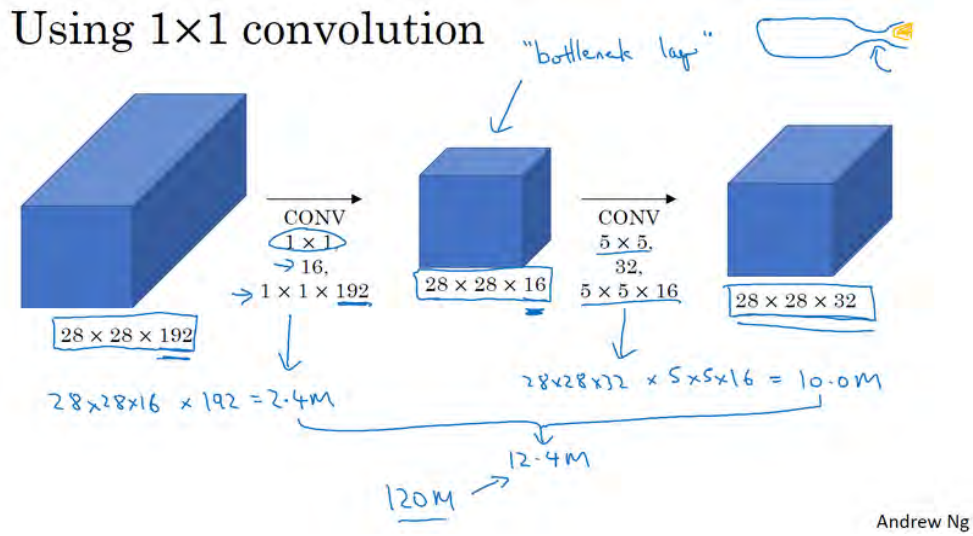
是 28×28×32，和上一页的相同。但我们要做的就是把左边这个大的输入层压缩成这个较小的的中间层，它只有 16 个通道，而不是 192 个。



有时候这被称为瓶颈层，瓶颈通常是某个对象最小的部分，假如你有这样一个玻璃瓶，这是瓶塞位置，瓶颈就是这个瓶子最小的部分。

同理，瓶颈层也是网络中最小的部分，我们先缩小网络表示，然后再扩大它。

接下来我们看看这个计算成本，应用 1×1 卷积，过滤器个数为 16，每个过滤器大小为 1×1×192，这两个维度相匹配（输入通道数与过滤器通道数），28×28×16 这个层的计算成本是，输出 28×28×192 中每个元素都做 192 次乘法，用 1×1×192 来表示，相乘结果约等于 240 万。



那第二个卷积层呢？240 万只是第一个卷积层的计算成本，第二个卷积层的计算成本又是多少呢？这是它的输出，28×28×32，对每个输出值应用一个 5×5×16 维度的过滤器，计算

结果为 1000 万。

所以所需要乘法运算的总次数是这两层的计算成本之和，也就是 1240 万，与上一张幻灯片中的值做比较，计算成本从 1.2 亿下降到了原来的十分之一，即 1240 万。所需要的加法运算与乘法运算的次数近似相等，所以我只统计了乘法运算的次数。

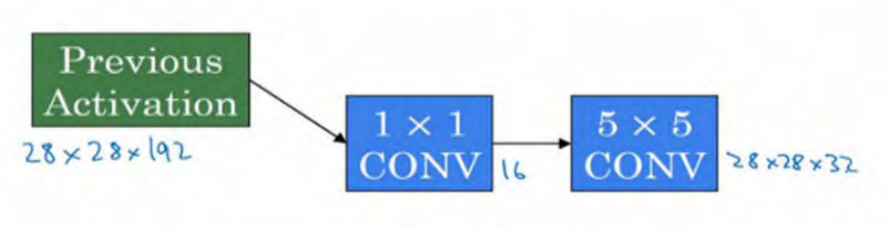
总结一下，如果你在构建神经网络层的时候，不想决定池化层是使用 1×1，3×3 还是 5×5 的过滤器，那么 **Inception** 模块就是最好的选择。我们可以应用各种类型的过滤器，只需要把输出连接起来。之后我们讲到计算成本问题，我们学习了如何通过使用 1×1 卷积来构建瓶颈层，从而大大降低计算成本。

你可能会问，仅仅大幅缩小表示层规模会不会影响神经网络的性能？事实证明，只要合理构建瓶颈层，你既可以显著缩小表示层规模，又不会降低网络性能，从而节省了计算。

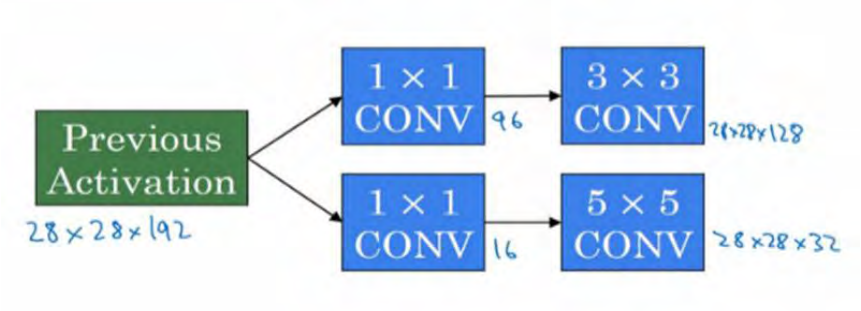
这就是 **Inception** 模块的主要思想，我们在这总结一下。下节课，我们将演示一个完整的 **Inception** 网络。

2.7 Inception 网络（Inception network）

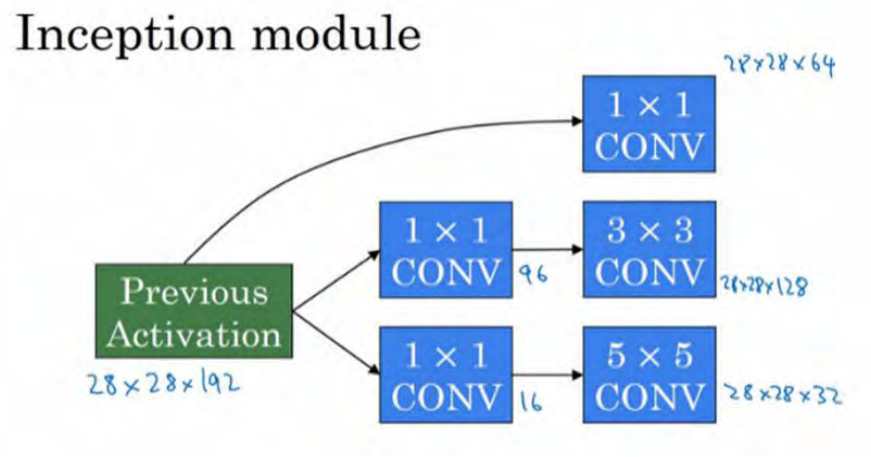
在上节视频中，你已经见到了所有的 Inception 网络基础模块。在本视频中，我们将学习如何将这些模块组合起来，构筑你自己的 Inception 网络。



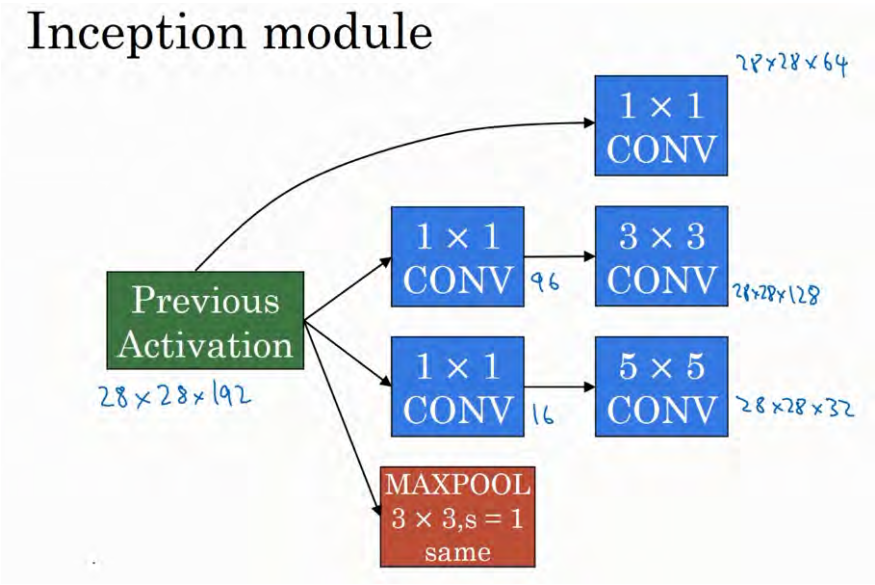
Inception 模块会将之前层的激活或者输出作为它的输入，作为前提，这是一个 28×28×192 的输入，和我们之前视频中的一样。我们详细分析过的例子是，先通过一个 1×1 的层，再通过一个 5×5 的层，1×1 的层可能有 16 个通道，而 5×5 的层输出为 28×28×32，共 32 个通道，这就是上个视频最后讲到的我们处理的例子。



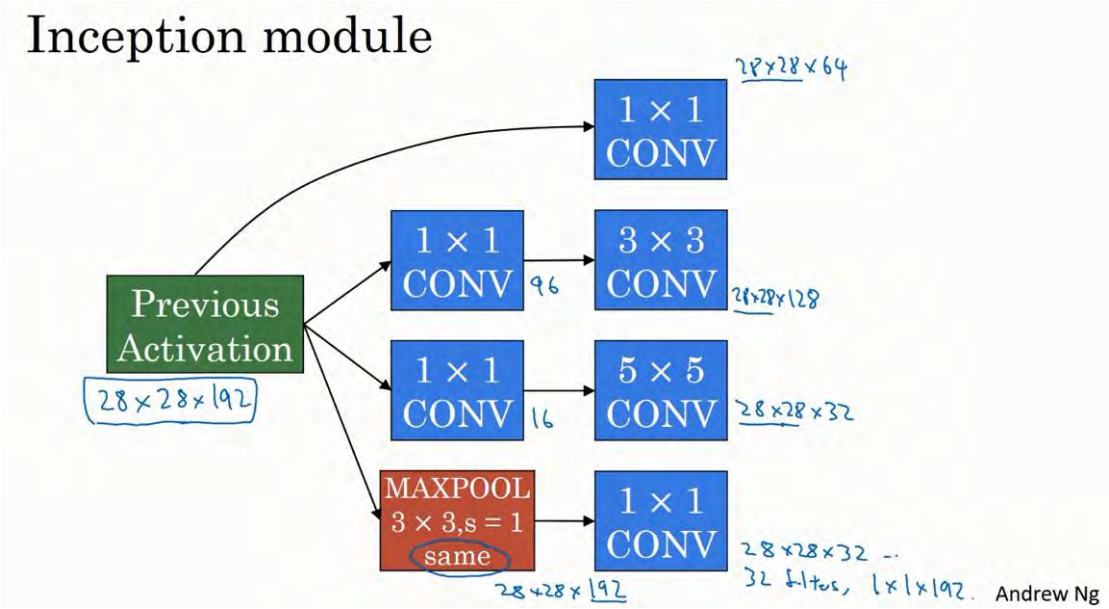
为了在这个 3×3 的卷积层中节省运算量，你也可以做相同的操作，这样的话 3×3 的层将会输出 28×28×128。



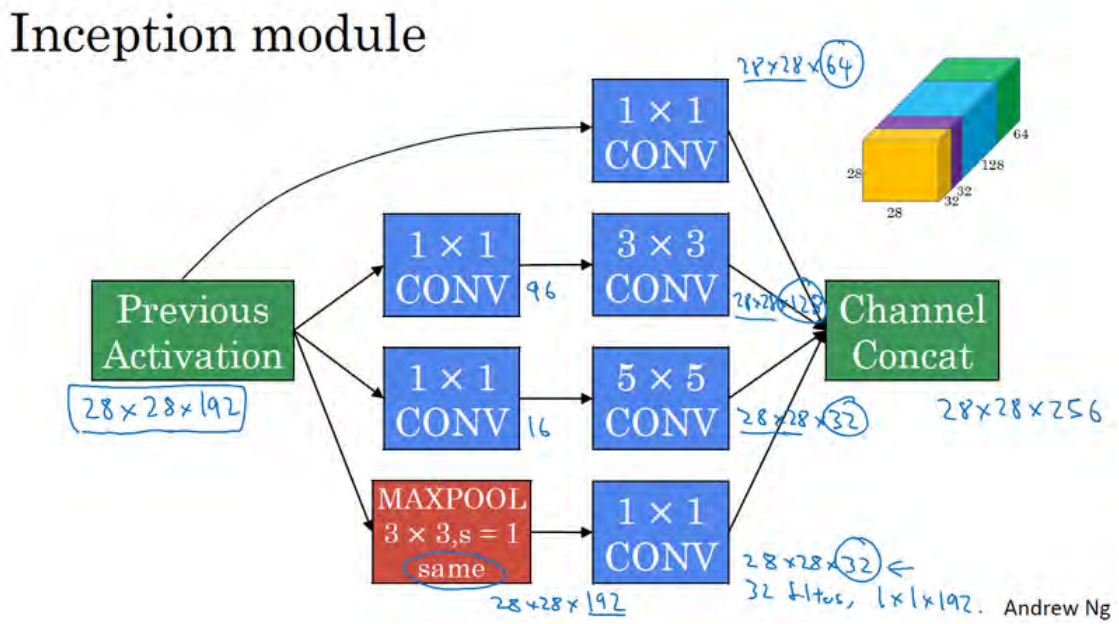
或许你还想将其直接通过一个 1×1 的卷积层，这时就不必在后面再跟一个 1×1 的层了，这样的话过程就只有一步，假设这个层的输出是 28×28×64。



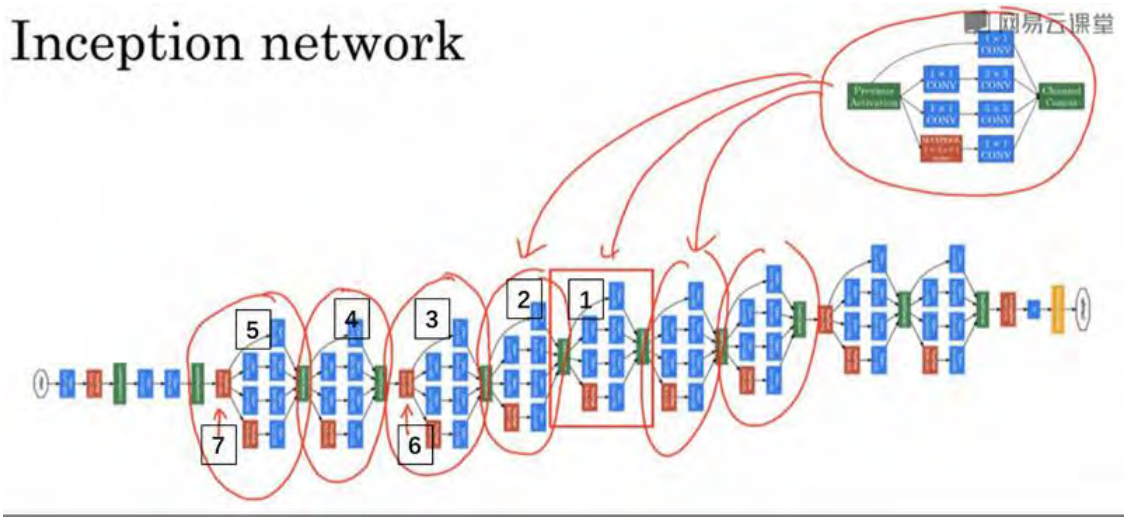
最后是池化层。



这里我们要做些有趣的事情，为了能在最后将这些输出都连接起来，我们会使用 **same** 类型的 **padding** 来池化，使得输出的高和宽依然是 28×28，这样才能将它与其他输出连接起来。但注意，如果你进行了最大池化，即便用了 **same padding**，3×3 的过滤器，**stride** 为 1，其输出将会是 28×28×192，其通道数或者说深度与这里的输入（通道数）相同。所以看起来它会有很多通道，我们实际要做的就是再加上一个 1×1 的卷积层，去进行我们在 1×1 卷积层的视频里所介绍的操作，将通道的数量缩小，缩小到 28×28×32。也就是使用 32 个维度为 1×1×192 的过滤器，所以输出的维度其通道数缩小为 32。这样就避免了最后输出时，池化层占据所有的通道。



最后，将这些方块全都连接起来。在这过程中，把得到的各个层的通道都加起来，最后得到一个 28×28×256 的输出。通道连接实际就是之前视频中看到过的，把所有方块连接在一起的操作。这就是一个 **Inception** 模块，而 **Inception** 网络所做的就是将这些模块都组合到一起。

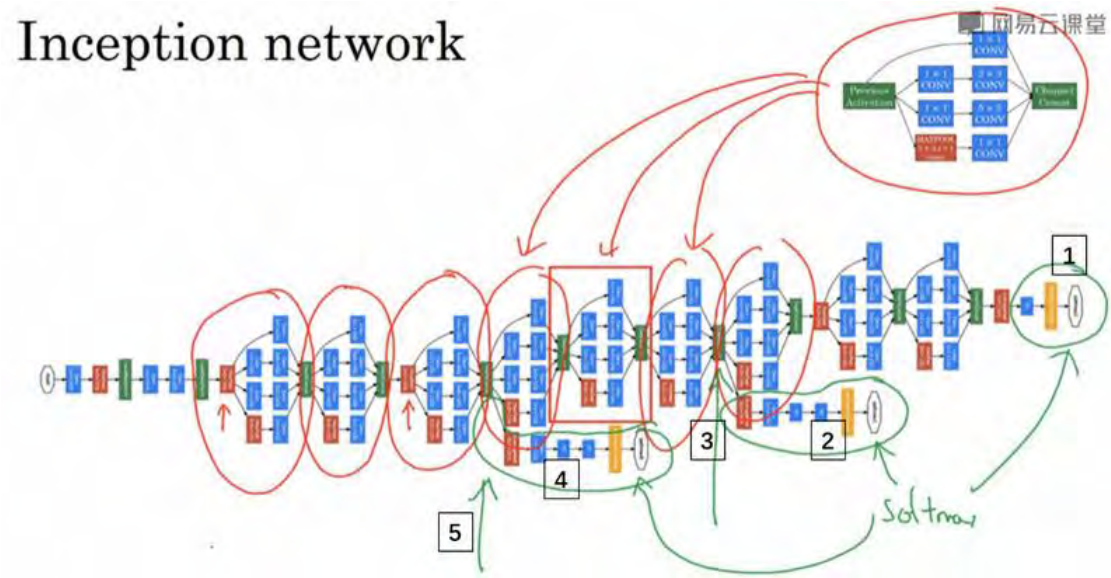


这是一张取自 **Szegety et al** 的论文中关于 **Inception** 网络的图片，你会发现图中有许多重复的模块，可能整张图看上去很复杂，但如果你只截取其中一个环节（编号 1），就会发现这是在前一页 ppt 中所见的 **Inception** 模块。

我们深入看看里边的一些细节，这是另一个 **Inception** 模块（编号 2），这也是一个 **Inception** 模块（编号 3）。这里有一些额外的最大池化层（编号 6）来修改高和宽的维度。这是另外一个 **Inception** 模块（编号 4），这是另外一个最大池化层（编号 7），它改变了高

和宽。而这里又是另一个 **Inception** 模块（编号 5）。

所以 **Inception** 网络只是很多这些你学过的模块在不同的位置重复组成的网络，所以如果你理解了之前所学的 **Inception** 模块，你就也能理解 **Inception** 网络。



事实上，如果你读过论文的原文，你就会发现，这里其实还有一些分支，我现在把它们加上去。所以这些分支有什么用呢？在网络的最后几层，通常称为全连接层，在它之后是一个 **softmax** 层（编号 1）来做出预测，这些分支（编号 2）所做的就是通过隐藏层（编号 3）来做出预测，所以这其实是一个 **softmax** 输出（编号 2），这（编号 1）也是。这是另一条分支（编号 4），它也包含了一个隐藏层，通过一些全连接层，然后有一个 **softmax** 来预测，输出结果的标签。

你应该把它看做 **Inception** 网络的一个细节，它确保了即便是隐藏单元和中间层（编号 5）也参与了特征计算，它们也能预测图片的分类。它在 **Inception** 网络中，起到一种调整的效果，并且能防止网络发生过拟合。

还有这个特别的 **Inception** 网络是由 **Google** 公司的作者所研发的，它被叫做 **GoogleLeNet**，这个名字是为了向 LeNet 网络致敬。在之前的视频中你应该了解了 **LeNet** 网络。我觉得这样非常好，因为深度学习研究人员是如此重视协作，深度学习工作者对彼此的工作成果有一种强烈的敬意。

最后，有个有趣的事实，**Inception** 网络这个名字又是缘何而来呢？**Inception** 的论文特地提到了这个模因（**meme**，网络用语即“梗”），就是“我们需要走的更深”（**We need to go deeper**），论文还引用了这个网址（http://knowyourmeme.com/memes/we-need-to-go-deeper），连接到这幅图片上，如果你看过 **Inception**（盗梦空间）这个电影，你应该能看懂这个由来。

作者其实是通过它来表明了建立更深的神经网络的决心，他们正是这样构建了 **Inception**。我想一般研究论文，通常不会引用网络流行模因（梗），但这里显然很合适。

http://knowyourmeme.com/memes/we-need-to-go-deeper Andrew Ng

最后总结一下，如果你理解了 **Inception** 模块，你就能理解 **Inception** 网络，无非是很多个 **Inception** 模块一环接一环，最后组成了网络。自从 **Inception** 模块诞生以来，经过研究者们的不断发展，衍生了许多新的版本。所以在你们看一些比较新的 **Inception** 算法的论文时，会发现人们使用这些新版本的算法效果也一样很好，比如 **Inception V2**、**V3** 以及 **V4**，还有一个版本引入了跳跃连接的方法，有时也会有特别好的效果。但所有的这些变体都建立在同一种基础的思想上，在之前的视频中你就已经学到过，就是把许多 **Inception** 模块通过某种方式连接到一起。通过这个视频，我想你应该能去阅读和理解这些 **Inception** 的论文，甚至是一些新版本的论文。

直到现在，你已经了解了许多专用的神经网络结构。在下节视频中，我将会告诉你们如何真正去使用这些算法来构建自己的计算机视觉系统，我们下节视频再见。

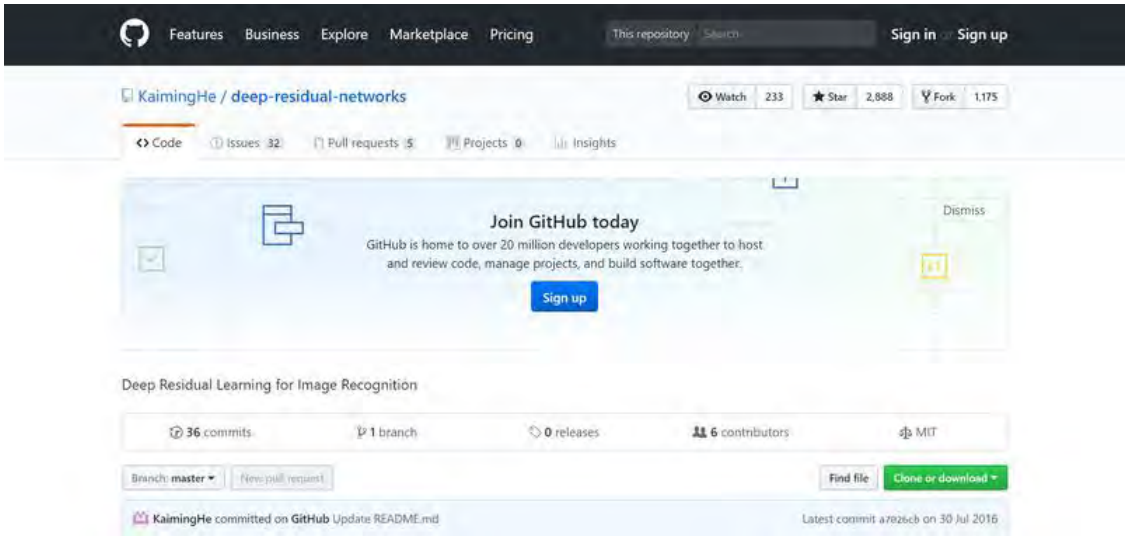
2.8 使用开源的实现方案（Using open-source implementations）

你现在已经学过几个非常有效的神经网络和 **ConvNet** 架构，在接下来的几段视频中我想与你分享几条如何使用它们的实用性建议，首先从使用开放源码的实现开始。

事实证明很多神经网络复杂细致，因而难以复制，因为一些参数调整的细节问题，例如学习率衰减等等，会影响性能。所以我发现有些时候，甚至在顶尖大学学习 **AI** 或者深度学习的博士生也很难通过阅读别人的研究论文来复制他人的成果。幸运的是有很多深度学习的研究者都习惯把自己的成果作为开发资源，放在像 **GitHub** 之类的网站上。当你自己编写代码时，我鼓励你考虑一下将你的代码贡献给开源社区。如果你看到一篇研究论文想应用它的成果，你应该考虑做一件事，我经常做的就是在网络上寻找一个开源的实现。因为你如果能得到作者的实现，通常要比你从头开始实现要快得多，虽然从零开始实现肯定可以是一个很好的锻炼。

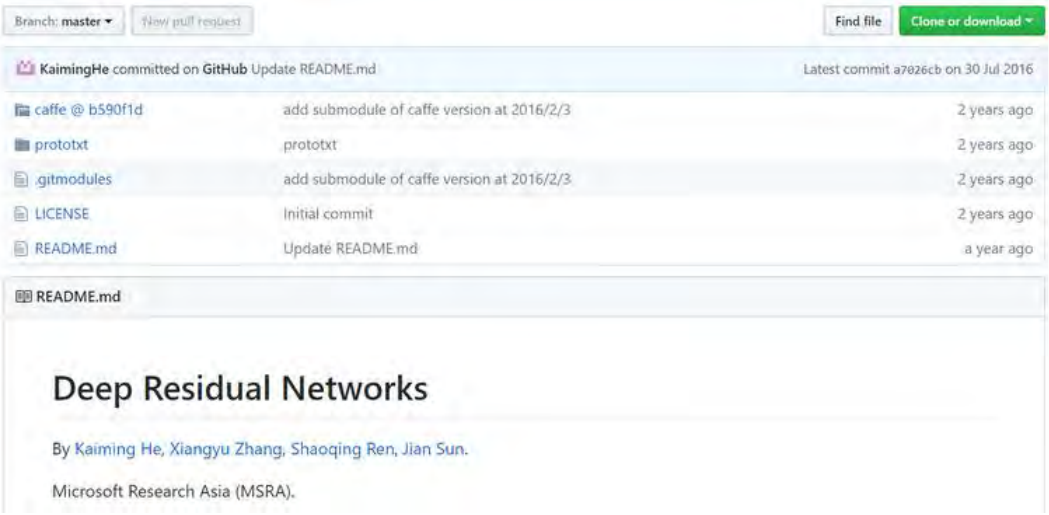
如果你已经熟悉如何使用 **GitHub**，这段视频对你来说可能没什么必要或者没那么重要。但是如果你不习惯从 **GitHub** 下载开源代码，让我来演示一下。

（整理者注：**ResNets** 实现的 **GitHub** 地址 https://github.com/KaimingHe/deep-residual-networks）

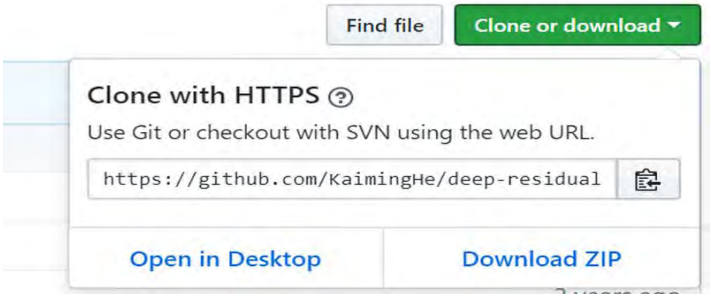


假设你对残差网络感兴趣，那就让我们搜索 **GitHub** 上的 **ResNets**，那么你可以在 **GitHub** 看到很多不同的 **ResNet** 的实现。我就打开这里的第一个网址，这是一个 **ResNets** 实现的 **GitHub** 资源库。在很多 **GitHub** 的网页上往下翻，你会看到一些描述，这个实现的文字说明。

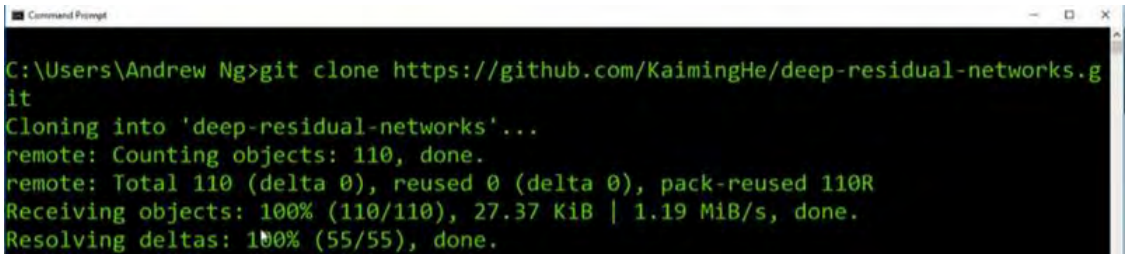
这个 **GitHub** 资源库，实际上是由 **ResNet** 论文原作者上传的。这些代码，这里有麻省理工学院的许可，你可以点击查看此许可的含义，**MIT** 许可是比较开放的开源许可之一。我将下载代码，点击这里的链接，它会给你一个 **URL**，通过这个你可以下载这个代码。



我点击这里的按钮（**Clone or download**），将这个 **URL** 复制到我的剪切板里。



（整理者注：**NG** 此处使用的是 **linux** 系统的 **bash** 命令行，对于 **win10** 系统，可以开启 **linux** 子系统功能，然后在 **win10** 应用商店下载 **ubuntu** 安装，运行 **CMD**，输入命令 **bash** 即可进入 **linux** 的 **bash** 命令行）



接着到这里，接下来你要做的就是输入 **git clone**，接着粘贴 **URL**，按下 **ENTER**，几秒之内就将这个资源库的副本下载到我的本地硬盘里。

让我们进入目录，让我们看一下，比起 **Windows**，我更习惯用 **Mac**，不过没关系，让我们试一下，让我们进入 **prototxt**，我认为这就是存放这些网络文件的地方。让我们看一下这个文件。因为这个文件很长，包含了 **ResNet** 里 101 层的详细配置。我记得，从这个网页上

看到这个特殊实现使用了 **Caffe** 框架。但如果你想通过其它编程框架来实现这一代码，你也可以尝试寻找一下。

```
C:\Users\Andrew Ng>cd deep-residual-networks

C:\Users\Andrew Ng\deep-residual-networks>dir
 Volume in drive C is Windows
 Volume Serial Number is 4E6C-6C59

 Directory of C:\Users\Andrew Ng\deep-residual-networks

10/15/2017  06:30 PM    <DIR>          .
10/15/2017  06:30 PM    <DIR>          ..
10/15/2017  06:30 PM                96 .gitmodules
10/15/2017  06:30 PM    <DIR>          caffe
10/15/2017  06:30 PM             1,100 LICENSE
10/15/2017  06:30 PM    <DIR>          prototxt
10/15/2017  06:30 PM             7,160 README.md
               3 File(s)          8,356 bytes
               4 Dir(s)  515,663,204,352 bytes free
```

```
C:\Users\Andrew Ng\deep-residual-networks>cd prototxt

C:\Users\Andrew Ng\deep-residual-networks\prototxt>dir
 Volume in drive C is Windows
 Volume Serial Number is 4E6C-6C59

 Directory of C:\Users\Andrew Ng\deep-residual-networks\prototxt

10/15/2017  06:30 PM    <DIR>          .
10/15/2017  06:30 PM    <DIR>          ..
10/15/2017  06:30 PM            69,987 ResNet-101-deploy.prototxt
10/15/2017  06:30 PM           104,809 ResNet-152-deploy.prototxt
10/15/2017  06:30 PM            34,820 ResNet-50-deploy.prototxt
               3 File(s)        209,616 bytes
               2 Dir(s)  515,663,151,104 bytes free
```

```
C:\Users\Andrew Ng\deep-residual-networks\prototxt>more ResNet-101-deploy.prototxt
```

```
name: "ResNet-101"
input: "data"
input_dim: 1
input_dim: 3
input_dim: 224
input_dim: 224

layer {
        bottom: "data"
        top: "conv1"
        name: "conv1"
        type: "Convolution"
        convolution_param {
                num_output: 64
                kernel_size: 7
                pad: 3
                stride: 2
                bias_term: false
-- More (0%) --
```

如果你在开发一个计算机视觉应用，一个常见的工作流程是，先选择一个你喜欢的架构，

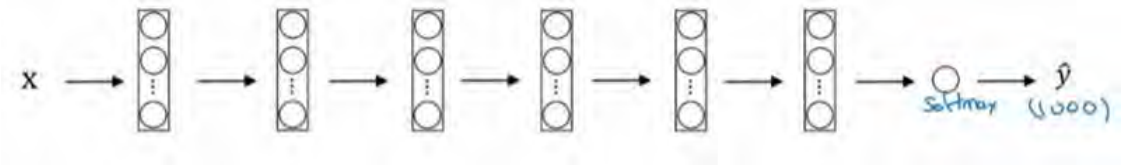
或许是你在这门课中学习到的，或者是你从朋友那听说的，或者是从文献中看到的，接着寻找一个开源实现，从 **GitHub** 下载下来，以此基础开始构建。这样做的优点在于，这些网络通常都需要很长的时间来训练，而或许有人已经使用多个 **GPU**，通过庞大的数据集预先训练了这些网络，这样一来你就可以使用这些网络进行迁移学习，我们将在下一节课讨论这些内容。

当然，如果你是一名计算机视觉研究员，从零来实现这些，那么你的工作流程将会不同，如果你自己构建，那么希望你将工作成果贡献出来，放到开源社区。因为已经有如此多计算机视觉研究者为了实现这些架构做了如此之多的工作，我发现从开源项目上开始是一个更好的方法，它也确实是一个更快开展新项目的方法。

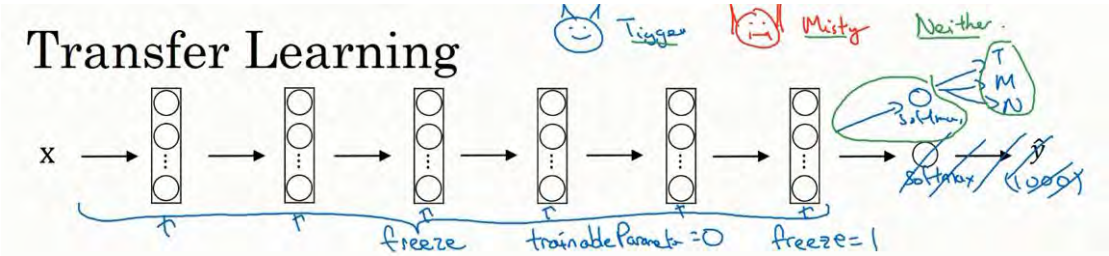
2.9 迁移学习（Transfer Learning）

如果你要做一个计算机视觉的应用，相比于从头训练权重，或者说从随机初始化权重开始，如果你下载别人已经训练好网络结构的权重，你通常能够进展的相当快，用这个作为预训练，然后转换到你感兴趣的任务上。计算机视觉的研究社区非常喜欢把许多数据集上传到网上，如果你听说过，比如 **ImageNet**，或者 **MS COCO**，或者 **Pascal** 类型的数据集，这些都是不同数据集的名字，它们都是由大家上传到网络的，并且有大量的计算机视觉研究者已经用这些数据集训练过他们的算法了。有时候这些训练过程需要花费好几周，并且需要很多的 **GPU**，其它人已经做过了，并且经历了非常痛苦的寻最优过程，这就意味着你可以下载花费了别人好几周甚至几个月而做出来的开源的权重参数，把它当作一个很好的初始化用在你自己的神经网络上。用迁移学习把公共的数据集的知识迁移到你自己的问题上，让我们看一下怎么做。

举个例子，假如说你要建立一个猫咪检测器，用来检测你自己的宠物猫。比如网络上的 **Tigger**，是一个常见的猫的名字，**Misty** 也是比较常见的猫名字。假如你的两只猫叫 **Tigger** 和 **Misty**，还有一种情况是，两者都不是。所以你现在有一个三分类问题，图片里是 **Tigger** 还是 **Misty**，或者都不是，我们忽略两只猫同时出现在一张图片里的情况。现在你可能没有 **Tigger** 或者 **Misty** 的大量的图片，所以你的训练集会很小，你该怎么办呢？

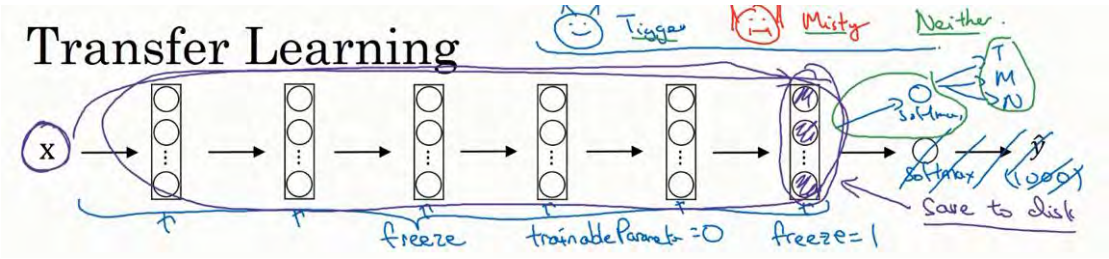


我建议你从网上下载一些神经网络开源的实现，不仅把代码下载下来，也把权重下载下来。有许多训练好的网络，你都可以下载。举个例子，**ImageNet** 数据集，它有 1000 个不同的类别，因此这个网络会有一个 **Softmax** 单元，它可以输出 1000 个可能类别之一。

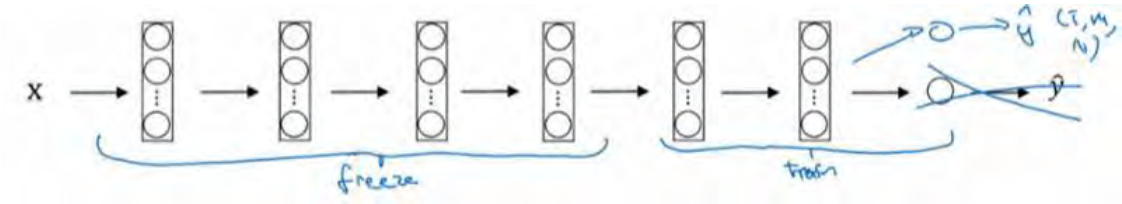


你可以去掉这个 **Softmax** 层，创建你自己的 **Softmax** 单元，用来输出 **Tigger**、**Misty** 和 **neither** 三个类别。就网络而言，我建议你把所有的层看作是冻结的，你冻结网络中所有层的参数，你只需要训练和你的 **Softmax** 层有关的参数。这个 **Softmax** 层有三种可能的输出，**Tigger**、**Misty** 或者都不是。

通过使用其他人预训练的权重，你很可能得到很好的性能，即使只有一个小的数据集。幸运的是，大多数深度学习框架都支持这种操作，事实上，取决于用的框架，它也许会有 trainableParameter=0 这样的参数，对于这些前面的层，你可能会设置这个参数。为了不训练这些权重，有时也会有 freeze=1 这样的参数。不同的深度学习编程框架有不同的方式，允许你指定是否训练特定层的权重。在这个例子中，你只需要训练 **softmax** 层的权重，把前面这些层的权重都冻结。

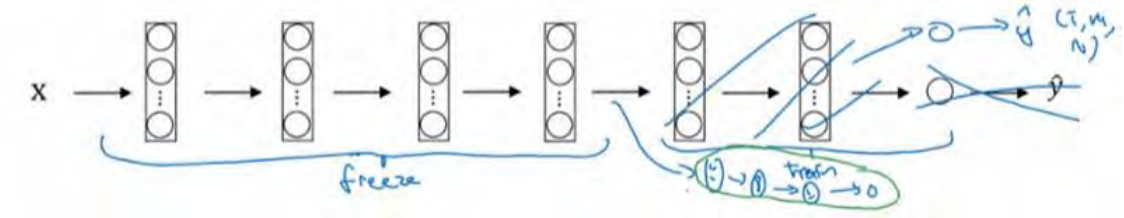


另一个技巧，也许对一些情况有用，由于前面的层都冻结了，相当于一个固定的函数，不需要改变。因为你不需要改变它，也不训练它，取输入图像X，然后把它映射到这层（**softmax** 的前一层）的激活函数。所以这个能加速训练的技巧就是，如果我们先计算这一层（紫色箭头标记），计算特征或者激活值，然后把它们存到硬盘里。你所做的就是用这个固定的函数，在这个神经网络的前半部分（**softmax** 层之前的所有层视为一个固定映射），取任意输入图像X，然后计算它的某个特征向量，这样你训练的就是一个很浅的 **softmax** 模型，用这个特征向量来做预测。对你的计算有用的一步就是对你的训练集中所有样本的这一层的激活值进行预计算，然后存储到硬盘里，然后在此之上训练 **softmax** 分类器。所以，存储到硬盘或者说预计算方法的优点就是，你不需要每次遍历训练集再重新计算这个激活值了。

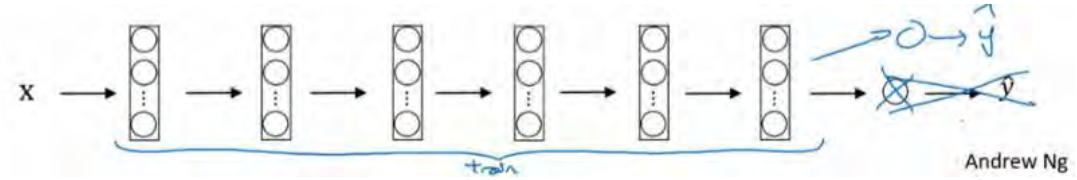


因此如果你的任务只有一个很小的数据集，你可以这样做。要有一个更大的训练集怎么办呢？根据经验，如果你有一个更大的标定的数据集，也许你有大量的 **Tigger** 和 **Misty** 的照

片，还有两者都不是的，这种情况，你应该冻结更少的层，比如只把这些层冻结，然后训练后面的层。如果你的输出层的类别不同，那么你需要构建自己的输出单元，**Tigger**、**Misty** 或者两者都不是三个类别。有很多方式可以实现，你可以取后面几层的权重，用作初始化，然后从这里开始梯度下降。



或者你可以直接去掉这几层，换成你自己的隐藏单元和你自己的 **softmax** 输出层，这些方法值得一试。但是有一个规律，如果你有越来越多的数据，你需要冻结的层数越少，你能够训练的层数就越多。这个理念就是，如果你有一个更大的数据集，也许有足够多的数据，那么不要单单训练一个 softmax 单元，而是考虑训练中等大小的网络，包含你最终要用的网络的后面几层。



最后，如果你有大量数据，你应该做的就是用开源的网络和它的权重，把这、所有的权重当作初始化，然后训练整个网络。再次注意，如果这是一个 1000 节点的 **softmax**，而你只有三个输出，你需要你自己的 **softmax** 输出层来输出你要的标签。

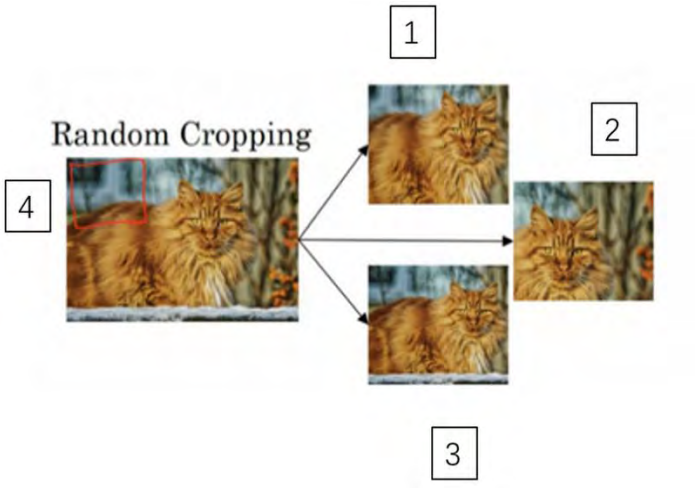
如果你有越多的标定的数据，或者越多的 **Tigger**、**Misty** 或者两者都不是的图片，你可以训练越多的层。极端情况下，你可以用下载的权重只作为初始化，用它们来代替随机初始化，接着你可以用梯度下降训练，更新网络所有层的所有权重。

这就是卷积网络训练中的迁移学习，事实上，网上的公开数据集非常庞大，并且你下载的其他人已经训练好几周的权重，已经从数据中学习了很多了，你会发现，对于很多计算机视觉的应用，如果你下载其他人的开源的权重，并用作你问题的初始化，你会做的更好。在所有不同学科中，在所有深度学习不同的应用中，我认为计算机视觉是一个你经常用到迁移学习的领域，除非你有非常非常大的的数据集，你可以从头开始训练所有的东西。总之，迁移学习是非常值得你考虑的，除非你有一个极其大的数据集和非常大的计算量预算来从头训练你的网络。

2.10 数据扩充（Data augmentation）

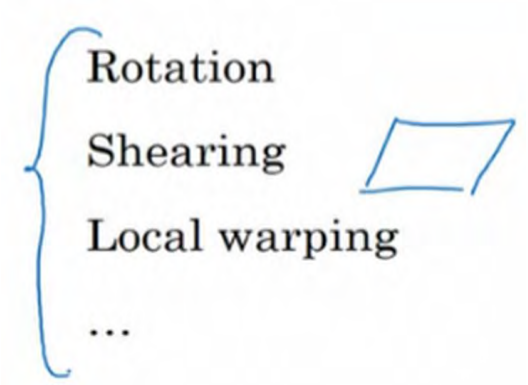
大部分的计算机视觉任务使用很多的数据，所以数据扩充是经常使用的一种技巧来提高计算机视觉系统的表现。我认为计算机视觉是一个相当复杂的工作，你需要输入图像的像素值，然后弄清楚图片中有什么，似乎你需要学习一个复杂方程来做这件事。在实践中，更多的数据对大多数计算机视觉任务都有所帮助，不像其他领域，有时候得到充足的数据，但是效果并不怎么样。但是，当下在计算机视觉方面，计算机视觉的主要问题是没有办法得到充足的数据。对大多数机器学习应用，这不是问题，但是对计算机视觉，数据就远远不够。所以这就意味着当你训练计算机视觉模型的时候，数据扩充会有所帮助，这是可行的，无论你是使用迁移学习，使用别人的预训练模型开始，或者从源代码开始训练模型。让我们来看一下计算机视觉中常见的数据扩充的方法。

或许最简单的数据扩充方法就是垂直镜像对称，假如，训练集中有这张图片，然后将其翻转得到右边的图像。对大多数计算机视觉任务，左边的图片是猫，然后镜像对称仍然是猫，如果镜像操作保留了图像中想识别的物体的前提下，这是个很实用的数据扩充技巧。



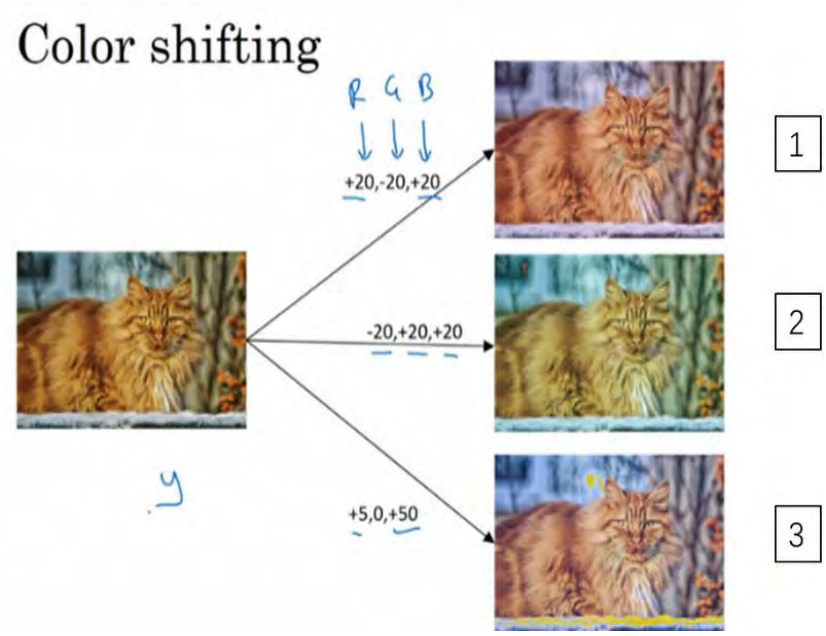
另一个经常使用的技巧是随机裁剪，给定一个数据集，然后开始随机裁剪，可能修剪这个（编号 1），选择裁剪这个（编号 2），这个（编号 3），可以得到不同的图片放在数据集中，你的训练集中有不同的裁剪。随机裁剪并不是一个完美的数据扩充的方法，如果你随机裁剪的那一部分（红色方框标记部分，编号 4），哪一个看起来更像猫。但在实践中，这个

方法还是很实用的，随机裁剪构成了很大一部分的真实图片。



镜像对称和随机裁剪是经常被使用的。当然，理论上，你也可以使用旋转，剪切（**shearing**：此处并非裁剪的含义，图像仅水平或垂直坐标发生变化）图像，可以对图像进行这样的扭曲变形，引入很多形式的局部弯曲等等。当然使用这些方法并没有坏处，尽管在实践中，因为太复杂了所以使用的很少。

第二种经常使用的方法是彩色转换，有这样一张图片，然后给 R、G 和 B 三个通道上加上不同的失真值。

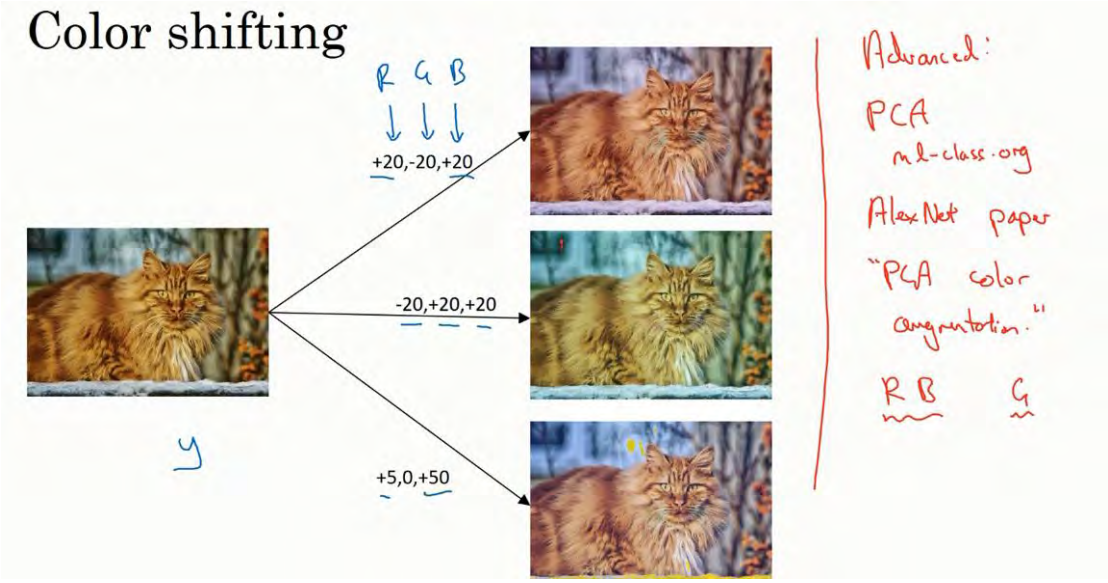


在这个例子中（编号 1），要给红色、蓝色通道加值，给绿色通道减值。红色和蓝色会产生紫色，使整张图片看起来偏紫，这样训练集中就有失真的图片。为了演示效果，我对图片的颜色进行改变比较夸张。在实践中，对 R、G 和 B 的变化是基于某些分布的，这样的改变也可能很小。

这么做的目的就是使用不同的 R、G 和 B 的值，使用这些值来改变颜色。在第二个例子

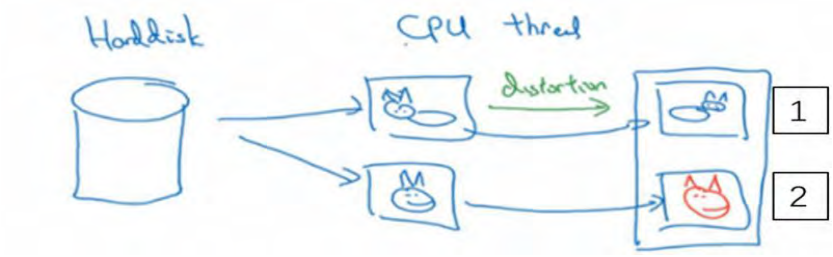
中（编号 2），我们少用了一点红色，更多的绿色和蓝色色调，这就使得图片偏黄一点。

在这（编号 3）使用了更多的蓝色，仅仅多了点红色。在实践中，R、G 和 B 的值是根据某种概率分布来决定的。这么做的理由是，可能阳光会有一点偏黄，或者是灯光照明有一点偏黄，这些可以轻易的改变图像的颜色，但是对猫的识别，或者是内容的识别，以及标签y，还是保持不变的。所以介绍这些，颜色失真或者是颜色变换方法，这样会使得你的学习算法对照片的颜色更改更具鲁棒性。

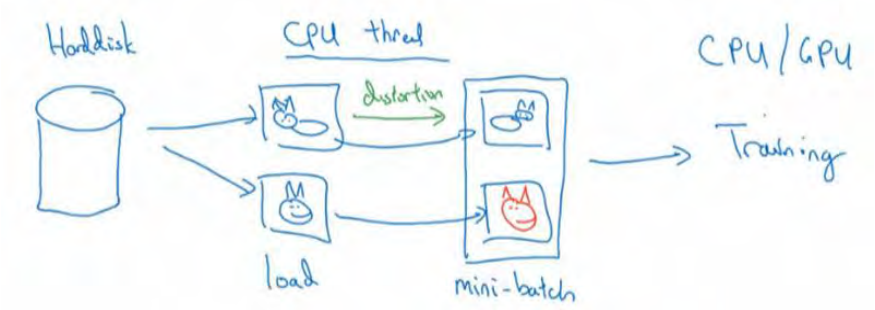


这是对更高级的学习者的一些注意提醒，你可以不理解我用红色标出来的内容。对 **R**、**G** 和 **B** 有不同的采样方式，其中一种影响颜色失真的算法是 **PCA**，即**主成分分析**，我在机器学习的 **mooc** 中讲过，在 **Coursera ml-class.Org** 机器学习这门课中。但具体颜色改变的细节在 **AlexNet** 的论文中有时候被称作 **PCA** 颜色增强，**PCA** 颜色增强的大概含义是，比如说，如果你的图片呈现紫色，即主要含有红色和蓝色，绿色很少，然后 **PCA** 颜色增强算法就会对红色和蓝色增减很多，绿色变化相对少一点，所以使总体的颜色保持一致。如果这些你都不懂，不需要担心，可以在网上搜索你想要了解的东西，如果你愿意的话可以阅读 **AlexNet** 论文中的细节，你也能找到 **PCA** 颜色增强的开源实现方法，然后直接使用它。

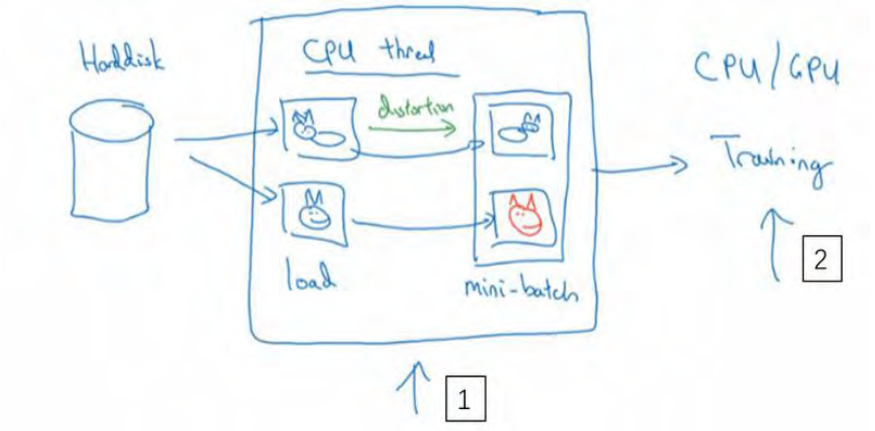
你可能有存储好的数据，你的训练数据存在硬盘上，然后使用符号，这个圆桶来表示你的硬盘。如果你有一个小的训练数据，你可以做任何事情，这些数据集就够了。



但是你有特别大的训练数据，接下来这些就是人们经常使用的方法。你可能会使用 **CPU** 线程，然后它不停的从硬盘中读取数据，所以你有一个从硬盘过来的图片数据流。你可以用 **CPU** 线程来实现这些失真变形，可以是随机裁剪、颜色变化，或者是镜像。但是对每张图片得到对应的某一种变形失真形式，看这张图片（编号 1），对其进行镜像变换，以及使用颜色失真，这张图最后会颜色变化（编号 2），从而得到不同颜色的猫。



与此同时，**CPU** 线程持续加载数据，然后实现任意失真变形，从而构成批数据或者最小批数据，这些数据持续的传输给其他线程或者其他的进程，然后开始训练，可以在 **CPU** 或者 **GPU** 上实现训一个大型网络的训练。



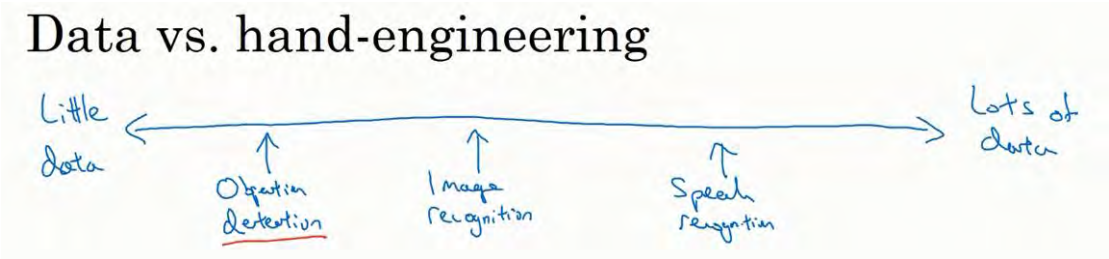
常用的实现数据扩充的方法是使用一个线程或者是多线程，这些可以用来加载数据，实现变形失真，然后传给其他的线程或者其他进程，来训练这个（编号 2）和这个（编号 1），

可以并行实现。

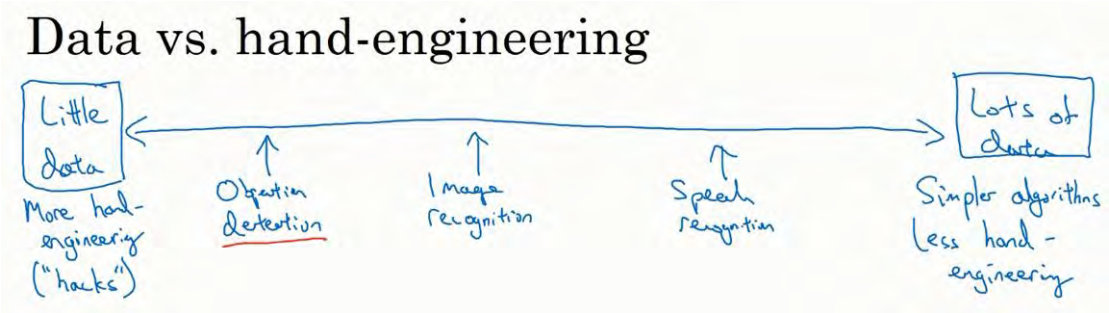
这就是数据扩充，与训练深度神经网络的其他部分类似，在数据扩充过程中也有一些超参数，比如说颜色变化了多少，以及随机裁剪的时候使用的参数。与计算机视觉其他部分类似，一个好的开始可能是使用别人的开源实现，了解他们如何实现数据扩充。当然如果你想获得更多的不变特性，而其他人的开源实现并没有实现这个，你也可以去调整这些参数。因此，我希望你们可以使用数据扩充使你的计算机视觉应用效果更好。

2.11 计算机视觉现状（The state of computer vision）

深度学习已经成功地应用于计算机视觉、自然语言处理、语音识别、在线广告、物流还有其他许多问题。在计算机视觉的现状下，深度学习应用于计算机视觉应用有一些独特之处。在这个视频中，我将和你们分享一些我对深度学习在计算机视觉方面应用的认识，希望能帮助你们更好地理解计算机视觉作品（此处指计算机视觉或者数据竞赛中的模型）以及其中的想法，以及如何自己构建这些计算机视觉系统。

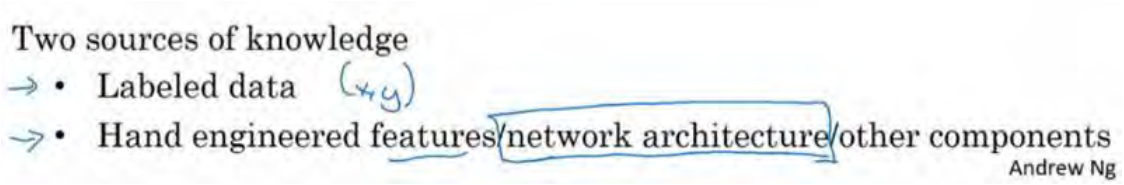


你可以认为大部分机器学习问题是介于少量数据和大量数据范围之间的。举个例子，我认为今天我们有相当数量的语音识别数据，至少相对于这个问题的复杂性而言。虽然现在图像识别或图像分类方面有相当大的数据集，因为图像识别是一个复杂的问题，通过分析像素并识别出它是什么，感觉即使在线数据集非常大，如超过一百万张图片，我们仍然希望我们能有更多的数据。还有一些问题，比如物体检测，我们拥有的数据更少。提醒一下，图像识别其实是如何看图片的问题，并且告诉你这张图是不是猫，而对象检测则是看一幅图，你画一个框，告诉你图片里的物体，比如汽车等等。因为获取边框的成本比标记对象的成本更高，所以我们进行对象检测的数据往往比图像识别数据要少，对象检测是我们下周要讨论的内容。



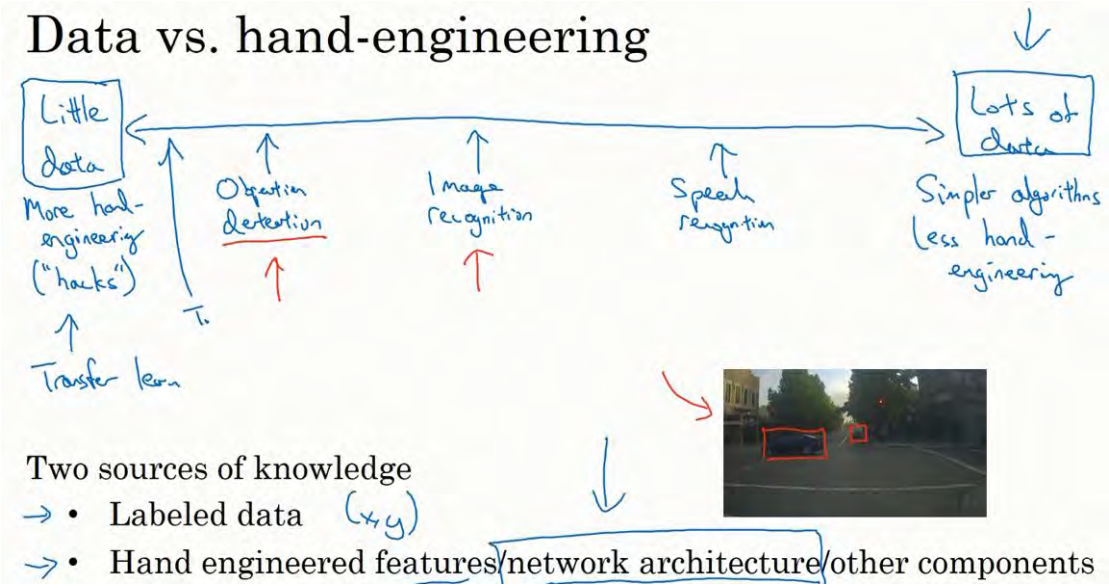
所以，观察一下机器学习数据范围图谱，你会发现当你有很多数据时，人们倾向于使用更简单的算法和更少的手工工程，因为我们不需要为这个问题精心设计特征。当你有大量的数据时，只要有一个大型的神经网络，甚至一个更简单的架构，可以是一个神经网络，就可以去学习它想学习的东西。

相反当你没有那么多的数据时，那时你会看到人们从事更多的是手工工程，低调点说就是你有很多小技巧可用（整理者注：在机器学习或者深度学习中，一般更崇尚更少的人工处理，而手工工程更多依赖人工处理，注意领会 Andrew NG 的意思）。但我认为你没有太多数据时，手工工程实际上是获得良好表现的最佳方式。



所以当我看机器学习应用时，我们认为通常我们的学习算法有两种知识来源，一个来源是被标记的数据，就像$(x，y)$应用在监督学习。第二个知识来源是手工工程，有很多方法去建立一个手工工程系统，它可以是源于精心设计的特征，手工精心设计的网络体系结构或者是系统的其他组件。所以当你没有太多标签数据时，你只需要更多地考虑手工工程。

所以我认为计算机视觉是在试图学习一个非常复杂的功能，我们经常感觉我们没有足够的数据，即使获得了更多数据，我们还是经常觉得还是没有足够的数据来满足需求。这就是为什么计算机视觉，从过去甚至到现在都更多地依赖于手工工程。我认为这也是计算机视觉领域发展相当复杂网络架构地原因，因为在缺乏更多数据的情况下，获得良好表现的方式还是花更多时间进行架构设计，或者说在网络架构设计上浪费（贬义褒用，即需要花费更多时间的意思）更多时间。



如果你认为我是在贬低手工工程，那并不是我的意思，当你没有足够的数据时，手工工程是一项非常困难，非常需要技巧的任务，它需要很好的洞察力，那些对手工工程有深刻见

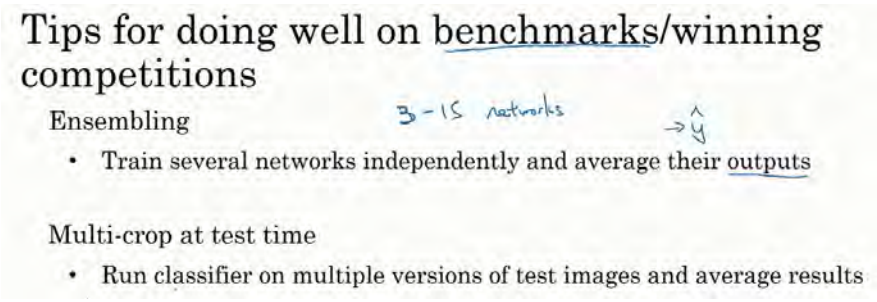
解的人将会得到更好的表现。当你没有足够的数据时，手工工程对一个项目来说贡献就很大。当你有很多数据的时候我就不会花时间去做手工工程，我会花时间去建立学习系统。但我认为从历史而言，计算机视觉领域还只是使用了非常小的数据集，因此从历史上来看计算机视觉还是依赖于大量的手工工程。甚至在过去的几年里，计算机视觉任务的数据量急剧增加，我认为这导致了手工工程量大幅减少，但是在计算机视觉上仍然有很多的网络架构使用手工工程，这就是为什么你会在计算机视觉中看到非常复杂的超参数选择，比你在其他领域中要复杂的多。实际上，因为你通常有比图像识别数据集更小的对象检测数据集，当我们谈论对象检测时，其实这是下周的任务，你会看到算法变得更加复杂，而且有更多特殊的组件。

幸运的是，当你有少量的数据时，有一件事对你很有帮助，那就是迁移学习。我想说的是，在之前的幻灯片中，**Tigger**、**Misty** 或者二者都不是的检测问题中，我们有这么少的数据，迁移学习会有很大帮助。这是另一套技术，当你有相对较少的数据时就可以用很多相似的数据。

如果你看一下计算机视觉方面的作品，看看那里的创意，你会发现人们真的是踌躇满志，他们在基准测试中和竞赛中表现出色。对计算机视觉研究者来说，如果你在基准上做得很好了，那就更容易发表论文了，所以有许多人致力于这些基准上，把它做得很好。积极的一面是，它有助于整个社区找出最有效得算法。但是你在论文上也看到，人们所做的事情让你在数据基准上表现出色，但你不会真正部署在一个实际得应用程序用在生产或一个系统上。

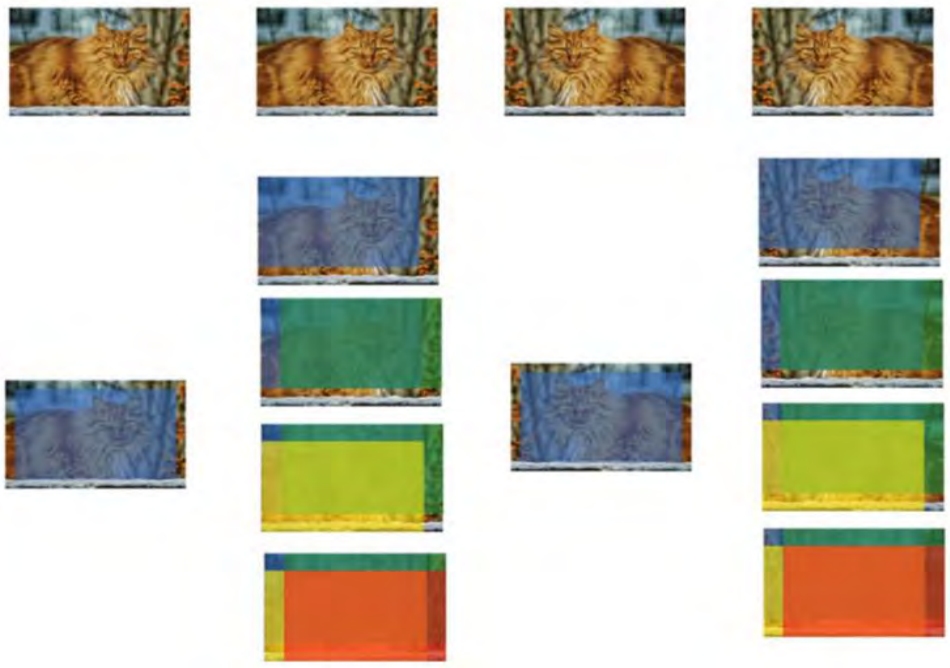
（整理着注：**Benchmark** 基准测试，**Benchmark** 是一个评价方式，在整个计算机领域有着长期的应用。维基百科上解释：“As computer architecture advanced, it became more difficult to compare the performance of various computer systems simply by looking at their specifications.Therefore, tests were developed that allowed comparison of different architectures.”Benchmark 在计算机领域应用最成功的就是性能测试，主要测试负载的执行时间、传输速度、吞吐量、资源占用率等。）

下面是一些有助于在基准测试中表现出色的小技巧，这些都是我自己从来没使用过的东西，如果我把一个系统投入生产，那就是为客户服务。

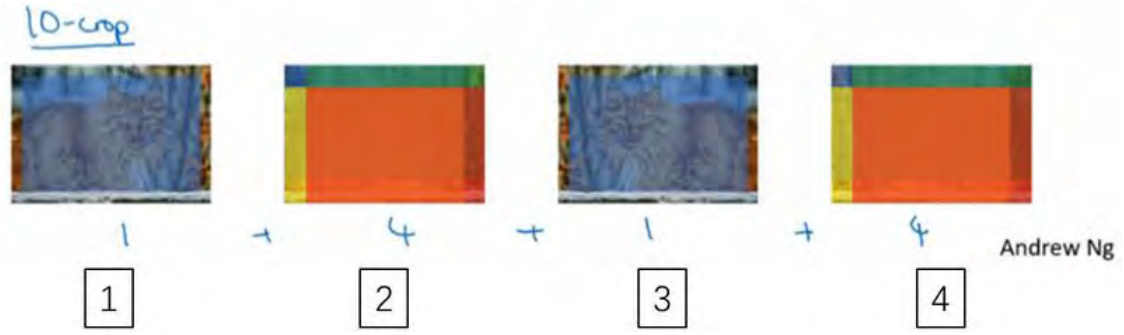


其中一个是集成，这就意味着在你想好了你想要的神经网络之后，可以独立训练几个神经网络，并平均它们的输出。比如说随机初始化三个、五个或者七个神经网络，然后训练所有这些网络，然后平均它们的输出。另外对他们的输出 $\hat{y}$ 进行平均计算是很重要的，不要平均他们的权重，这是行不通的。看看你的 7 个神经网络，它们有 7 个不同的预测，然后平均他们，这可能会让你在基准上提高 1%，2%或者更好。这会让你做得更好，也许有时会达到 1%或 2%，这真的能帮助你赢得比赛。但因为集成意味着要对每张图片进行测试，你可能需要在从 3 到 15 个不同的网络中运行一个图像，这是很典型的，因为这 3 到 15 个网络可能会让你的运行时间变慢，甚至更多时间，所以技巧之一的集成是人们在基准测试中表现出色和赢得比赛的利器，但我认为这几乎不用于生产服务于客户的，我想除非你有一个巨大的计算预算而且不介意在每个用户图像数据上花费大量的计算。

你在论文中可以看到在测试时，对进准测试有帮助的另一个技巧就是 **Multi-crop at test time**，我的意思是你已经看到了如何进行数据扩充，**Multi-crop** 是一种将数据扩充应用到你的测试图像中的一种形式。



举个例子，让我们看看猫的图片，然后把它复制四遍，包括它的两个镜像版本。有一种叫作 **10-crop** 的技术（crop 理解为裁剪的意思），它基本上说，假设你取这个中心区域，裁剪，然后通过你的分类器去运行它，然后取左上角区域，运行你的分类器，右上角用绿色表示，左下方用黄色表示，右下方用橙色表示，通过你的分类器来运行它，然后对镜像图像做同样的事情对吧？所以取中心的 **crop**，然后取四个角落的 **crop**。



这是这里（编号 1）和这里（编号 3）就是中心 **crop**，这里（编号 2）和这里（编号 4）就是四个角落的 **crop**。如果把这些加起来，就会有 10 种不同的图像的 **crop**，因此命名为 **10-crop**。所以你要做的就是，通过你的分类器来运行这十张图片，然后对结果进行平均。如果你有足够的计算预算，你可以这么做，也许他们需要 10 个 **crops**，你可以使用更多，这可能会让你在生产系统中获得更好的性能。如果是生产的话，我的意思还是实际部署用户的系统。但这是另一种技术，它在基准测试上的应用，要比实际生产系统中好得多。

集成的一个大问题是你需要保持所有这些不同的神经网络，这就占用了更多的计算机内存。对于 **multi-crop**，我想你只保留一个网络，所以它不会占用太多的内存，但它仍然会让你的运行时间变慢。

这些是你看到的小技巧，研究论文也可以参考这些，但我个人并不倾向于在构建生产系统时使用这些方法，尽管它们在基准测试和竞赛上做得很好。

由于计算机视觉问题建立在小数据集之上，其他人已经完成了大量的网络架构的手工工程。一个神经网络在某个计算机视觉问题上很有效，但令人惊讶的是它通常也会解决其他计算机视觉问题。

所以，要想建立一个实用的系统，

你最好先从其他人的神经网络架构入手。

如果可能的话，你可以使用开源的一些应用，因为开放的源码实现可能已经找到了所有繁琐的细节，比如学习率衰减方式或者超参数。

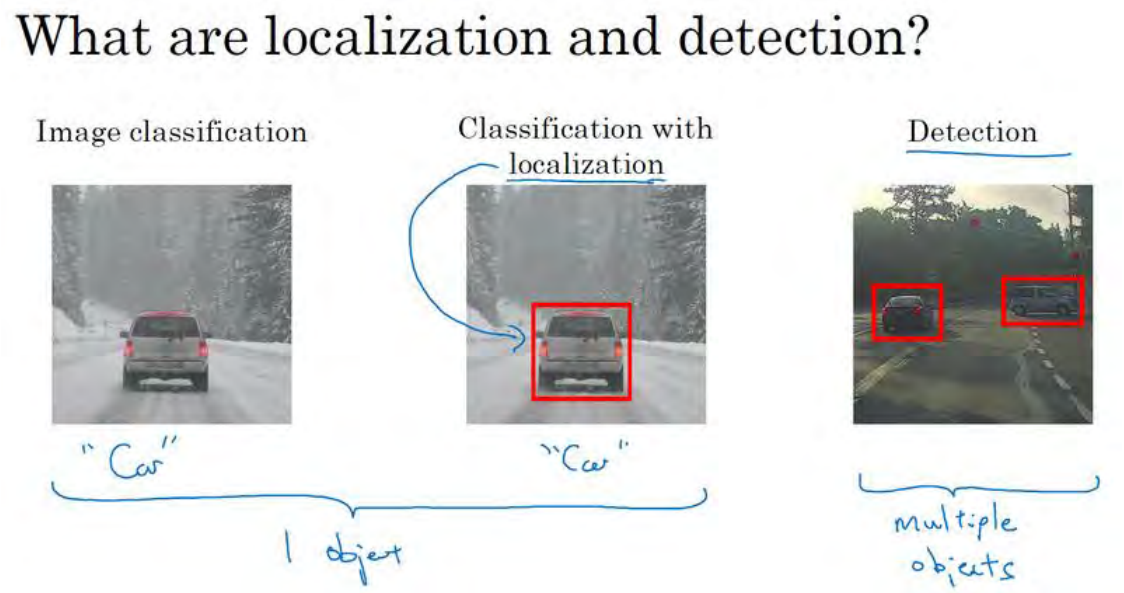
最后，其他人可能已经在几路 **GPU** 上花了几个星期的时间来训练一个模型，训练超过一百万张图片，所以通过使用其他人的预先训练得模型，然后在数据集上进行微调，你可以在应用程序上运行得更快。当然如果你有电脑资源并且有意愿，我不会阻止你从头开始训练你自己的网络。事实上，如果你想发明你自己的计算机视觉算法，这可能是你必须要做的。

这就是本周的学习，我希望看到大量的计算机视觉架构能够帮助你理解什么是有效的。在本周的编程练习中，你实际上会学习另一种编程框架，并使用它来实现 **ResNets**。所以我希望你们喜欢这个编程练习，我期待下周还能见到你们。

第三周 目标检测（Object detection）

3.1 目标定位（Object localization）

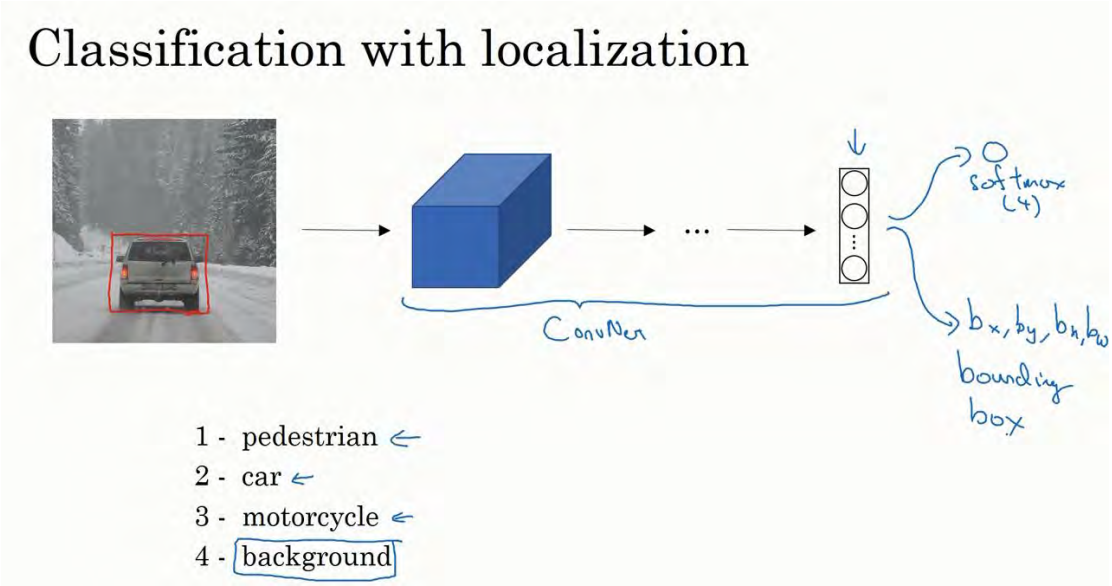
这一周我们学习的主要内容是对象检测，它是计算机视觉领域中一个新兴的应用方向，相比前两年，它的性能越来越好。在构建对象检测之前，我们先了解一下对象定位，首先我们看看它的定义。



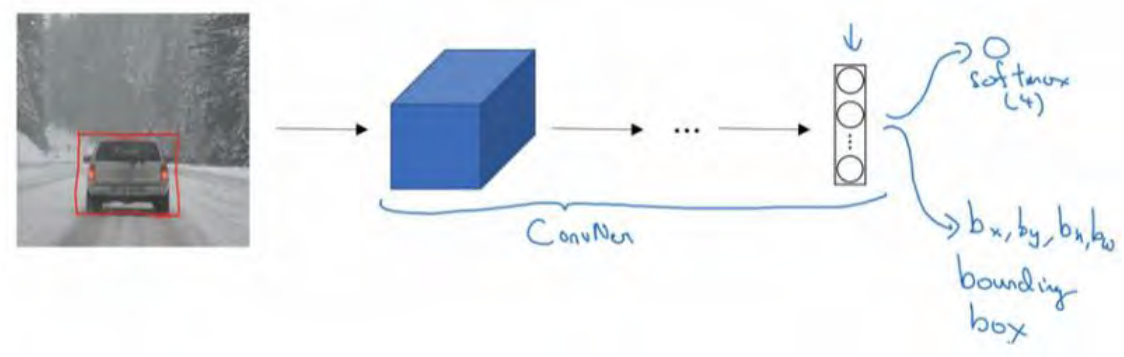
图片分类任务我们已经熟悉了，就是算法遍历图片，判断其中的对象是不是汽车，这就是图片分类。这节课我们要学习构建神经网络的另一个问题，即定位分类问题。这意味着，我们不仅要用算法判断图片中是不是一辆汽车，还要在图片中标记出它的位置，用边框或红色方框把汽车圈起来，这就是定位分类问题。其中“定位”的意思是判断汽车在图片中的具体位置。这周后面几天，我们再讲讲当图片中有多个对象时，应该如何检测它们，并确定出位置。比如，你正在做一个自动驾驶程序，程序不但要检测其它车辆，还要检测其它对象，如行人、摩托车等等，稍后我们再详细讲。

本周我们要研究的分类定位问题，通常只有一个较大的对象位于图片中间位置，我们要对它进行识别和定位。而在对象检测问题中，图片可以含有多个对象，甚至单张图片中会有多个不同分类的对象。因此，图片分类的思路可以帮助学习分类定位，而对象定位的思路又有助于学习对象检测，我们先从分类和定位开始讲起。

图片分类问题你已经并不陌生了，例如，输入一张图片到多层卷积神经网络。这就是卷积神经网络，它会输出一个特征向量，并反馈给 **softmax** 单元来预测图片类型。

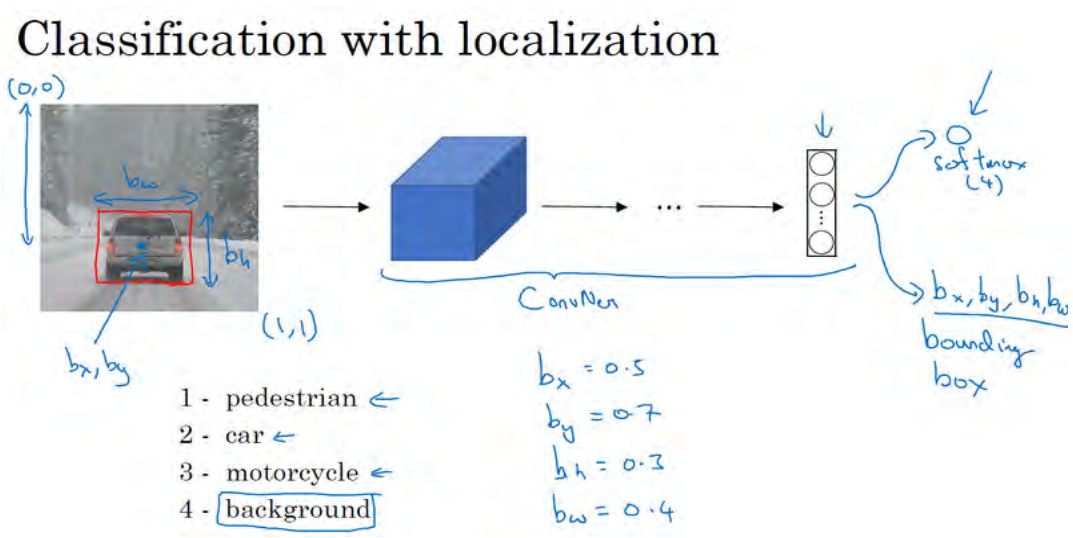


如果你正在构建汽车自动驾驶系统，那么对象可能包括以下几类：行人、汽车、摩托车和背景，这意味着图片中不含有前三种对象，也就是说图片中没有行人、汽车和摩托车，输出结果会是背景对象，这四个分类就是 softmax 函数可能输出的结果。

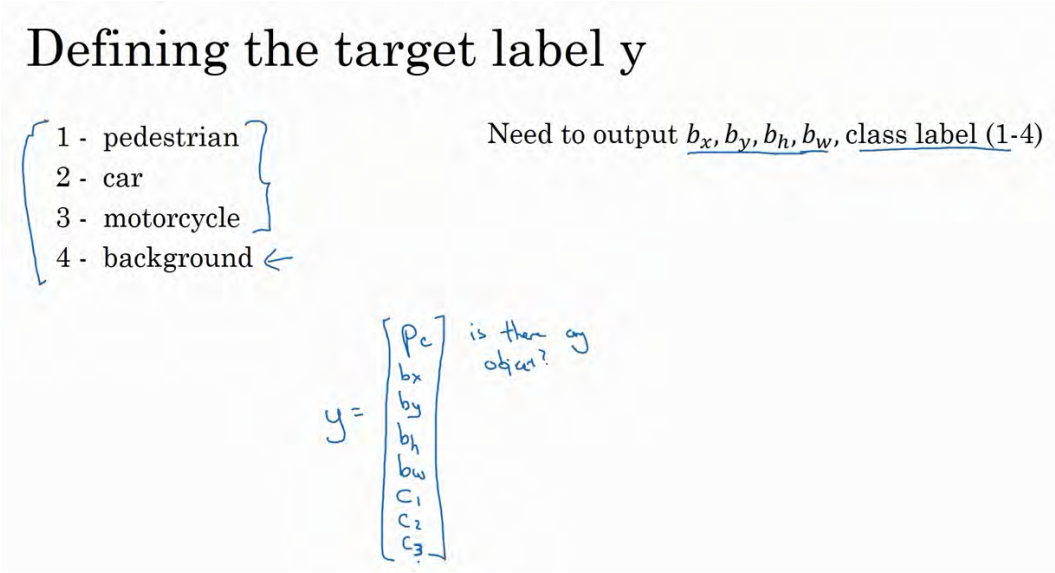


这就是标准的分类过程，如果你还想定位图片中汽车的位置，该怎么做呢？我们可以让神经网络多输出几个单元，输出一个边界框。具体说就是让神经网络再多输出 4 个数字，标记为b_x,b_y,b_h和b_w，这四个数字是被检测对象的边界框的参数化表示。

我们先来约定本周课程将使用的符号表示，图片左上角的坐标为(0,0)，右下角标记为(1,1)。要确定边界框的具体位置，需要指定红色方框的中心点，这个点表示为(b_x,b_y)，边界框的高度为b_h，宽度为b_w。因此训练集不仅包含神经网络要预测的对象分类标签，还要包含表示边界框的这四个数字，接着采用监督学习算法，输出一个分类标签，还有四个参数值，从而给出检测对象的边框位置。此例中，b_x的理想值是 0.5，因为它表示汽车位于图片水平方向的中间位置；b_y大约是 0.7，表示汽车位于距离图片底部$\frac{3}{10}$的位置；b_h约为 0.3，因为红色方框的高度是图片高度的 0.3 倍；b_w约为 0.4，红色方框的宽度是图片宽度的 0.4 倍。



下面我再具体讲讲如何为监督学习任务定义目标标签 y。

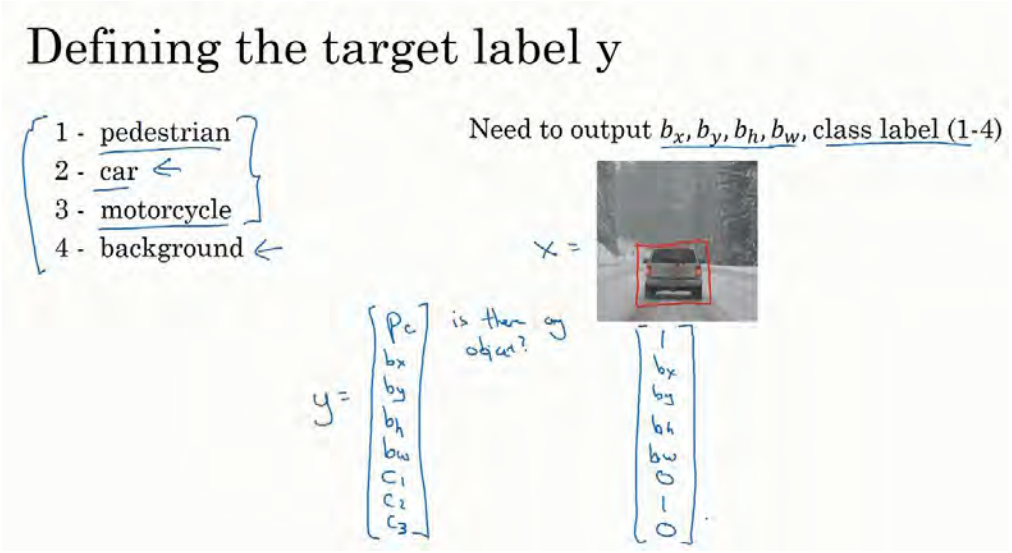


请注意，这有四个分类，神经网络输出的是这四个数字和一个分类标签，或分类标签出现的概率。目标标签y的定义如下：$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$

它是一个向量，第一个组件p_c表示是否含有对象，如果对象属于前三类（行人、汽车、摩托车），则$p_c = 1$，如果是背景，则图片中没有要检测的对象，则$p_c = 0$。我们可以这样理解p_c，它表示被检测对象属于某一分类的概率，背景分类除外。

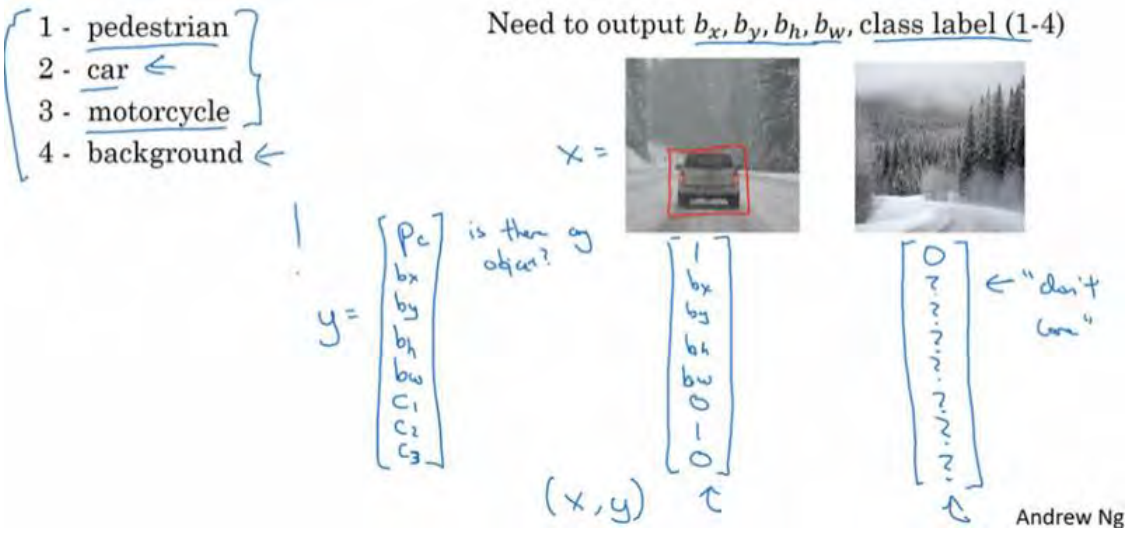
如果检测到对象，就输出被检测对象的边界框参数b_x、b_y、b_h和b_w。最后，如果存在某个对象，那么$p_c = 1$，同时输出c_1、c_2和c_3，表示该对象属于 1-3 类中的哪一类，是行人，汽

车还是摩托车。鉴于我们所要处理的问题，我们假设图片中只含有一个对象，所以针对这个分类定位问题，图片最多只会出现其中一个对象。



我们再看几个样本，假如这是一张训练集图片，标记为x，即上图的汽车图片。而在y当中，第一个元素$p_c = 1$，因为图中有一辆车，b_x、b_y、b_h和b_w会指明边界框的位置，所以标签训练集需要标签的边界框。图片中是一辆车，所以结果属于分类 2，因为定位目标不是行人或摩托车，而是汽车，所以$c_1 = 0$，$c_2 = 1$，$c_3 = 0$，c_1、c_2和c_3中最多只有一个等于 1。

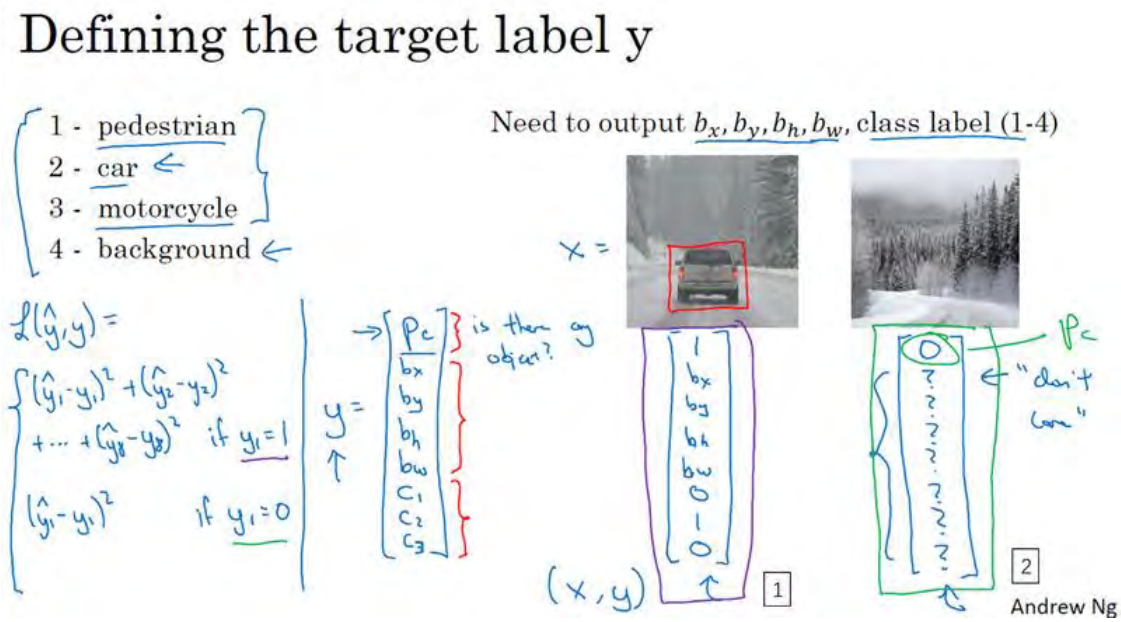
这是图片中只有一个检测对象的情况，如果图片中没有检测对象呢？如果训练样本是这样一张图片呢？



这种情况下，$p_c = 0$，y的其它参数将变得毫无意义，这里我全部写成问号，表示“毫无意义”的参数，因为图片中不存在检测对象，所以不用考虑网络输出中边界框的大小，也不用考虑图片中的对象是属于c_1、c_2和c_3中的哪一类。针对给定的被标记的训练样本，不论图片中是否含有定位对象，构建输入图片 X 和分类标签Y 的具体过程都是如此。这些数据最

终定义了训练集。

最后，我们介绍一下神经网络的损失函数，其参数为类别y和网络输出$\hat{y}$，如果采用平方误差策略，则$L(\hat{y},y)=(\hat{y_1}-y_1)^2+(\hat{y_2}-y_2)^2+\cdots(\hat{y_8}-y_8)^2$，损失值等于每个元素相应差值的平方和。



如果图片中存在定位对象，那么$y_1=1$，所以$y_1=p_c$，同样地，如果图片中存在定位对象，$p_c=1$，损失值就是不同元素的平方和。

另一种情况是，$y_1=0$，也就是$p_c=0$，损失值是$(\hat{y_1}-y_1)^2$，因为对于这种情况，我们不用考虑其它元素，只需要关注神经网络输出p_c的准确度。

回顾一下，当$y_1=1$时，也就是这种情况（编号 1），平方误差策略可以减少这 8 个元素预测值和实际输出结果之间差值的平方。如果$y_1=0$，y 矩阵中的后 7 个元素都不用考虑（编号 2），只需要考虑神经网络评估y_1（即p_c）的准确度。

为了让大家了解对象定位的细节，这里我用平方误差简化了描述过程。实际应用中，你可以不对c_1、c_2、c_3和 **softmax** 激活函数应用对数损失函数，并输出其中一个元素值，通常做法是对边界框坐标应用平方差或类似方法，对p_c应用逻辑回归函数，甚至采用平方预测误差也是可以的。

以上就是利用神经网络解决对象分类和定位问题的详细过程，结果证明，利用神经网络输出批量实数来识别图片中的对象是个非常有用的算法。下节课，我想和大家分享另一种思路，就是把神经网络输出的实数集作为一个回归任务，这个思想也被应用于计算机视觉的其它领域，也是非常有效的。

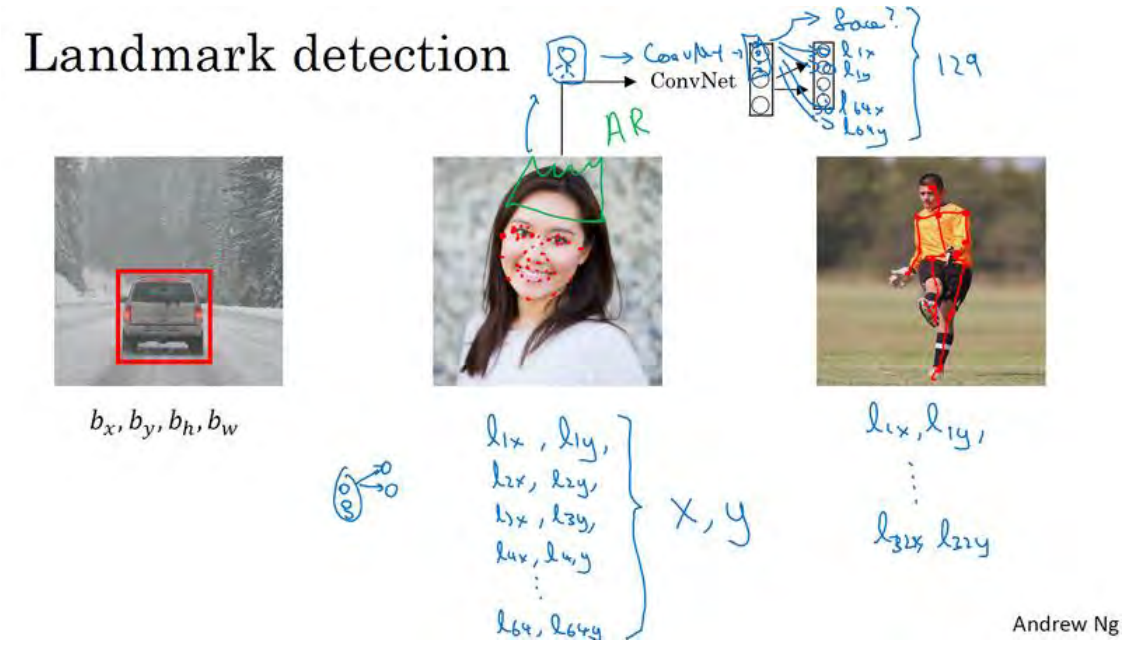
3.2 特征点检测（Landmark detection）

上节课，我们讲了如何利用神经网络进行对象定位，即通过输出四个参数值b_x、b_y、b_h和b_w给出图片中对象的边界框。更概括地说，神经网络可以通过输出图片上特征点的(x, y)坐标来实现对目标特征的识别，我们看几个例子。

假设你正在构建一个人脸识别应用，出于某种原因，你希望算法可以给出眼角的具体位置。眼角坐标为(x, y)，你可以让神经网络的最后一层多输出两个数字l_x和l_y，作为眼角的坐标值。如果你想知道两只眼睛的四个眼角的具体位置，那么从左到右，依次用四个特征点来表示这四个眼角。对神经网络稍做些修改，输出第一个特征点（l_{1x}，l_{1y}），第二个特征点（l_{2x}，l_{2y}），依此类推，这四个脸部特征点的位置就可以通过神经网络输出了。

也许除了这四个特征点，你还想得到更多的特征点输出值，这些（图中眼眶上的红色特征点）都是眼睛的特征点，你还可以根据嘴部的关键点输出值来确定嘴的形状，从而判断人物是在微笑还是皱眉，也可以提取鼻子周围的关键特征点。为了便于说明，你可以设定特征点的个数，假设脸部有 64 个特征点，有些点甚至可以帮助你定义脸部轮廓或下颌轮廓。选定特征点个数，并生成包含这些特征点的标签训练集，然后利用神经网络输出脸部关键特征点的位置。

具体做法是，准备一个卷积网络和一些特征集，将人脸图片输入卷积网络，输出 1 或 0，1 表示有人脸，0 表示没有人脸，然后输出（l_{1x}，l_{1y}）……直到（l_{64x}，l_{64y}）。这里我用l代表一个特征，这里有 129 个输出单元，其中 1 表示图片中有人脸，因为有 64 个特征，64×2=128，所以最终输出 128+1=129 个单元，由此实现对图片的人脸检测和定位。这只是一个识别脸部表情的基本构造模块，如果你玩过 Snapchat 或其它娱乐类应用，你应该对 AR（增强现实）过滤器多少有些了解，Snapchat 过滤器实现了在脸上画皇冠和其他一些特殊效果。检测脸部特征也是计算机图形效果的一个关键构造模块，比如实现脸部扭曲，头戴皇冠等等。当然为了构建这样的网络，你需要准备一个标签训练集，也就是图片x和标签y的集合，这些点都是人为辛苦标注的。



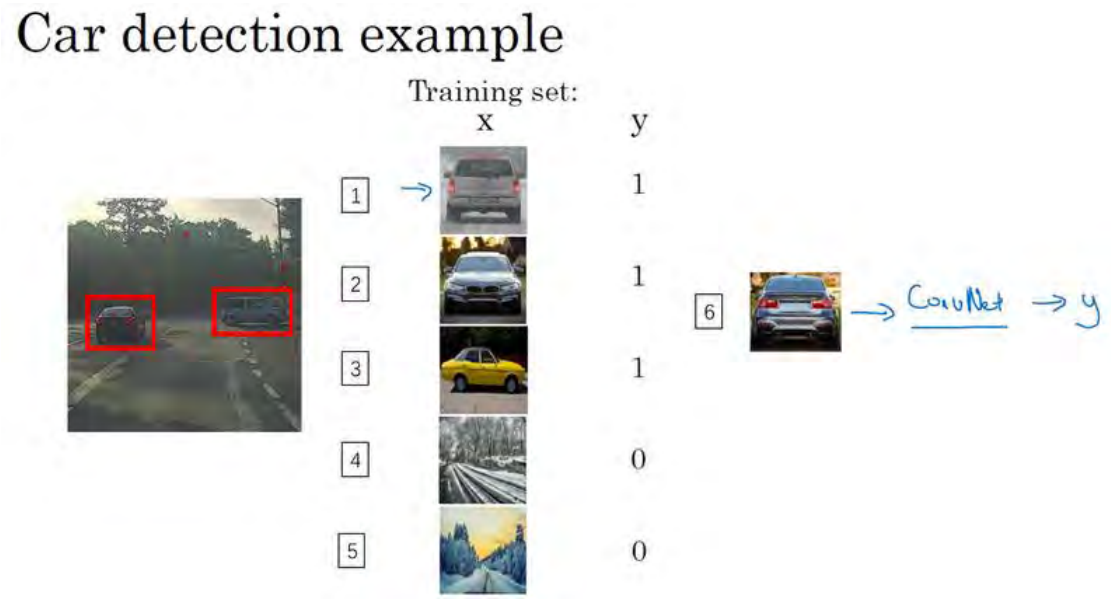
最后一个例子，如果你对人体姿态检测感兴趣，你还可以定义一些关键特征点，如胸部的中点，左肩，左肘，腰等等。然后通过神经网络标注人物姿态的关键特征点，再输出这些标注过的特征点，就相当于输出了人物的姿态动作。当然，要实现这个功能，你需要设定这些关键特征点，从胸部中心点(l_{1x}，l_{1y})一直往下，直到(l_{32x}，l_{32y})。

一旦了解如何用二维坐标系定义人物姿态，操作起来就相当简单了，批量添加输出单元，用以输出要识别的各个特征点的(x, y)坐标值。要明确一点，特征点 1 的特性在所有图片中必须保持一致，就好比，特征点 1 始终是右眼的外眼角，特征点 2 是右眼的内眼角，特征点 3 是左眼内眼角，特征点 4 是左眼外眼角等等。所以标签在所有图片中必须保持一致，假如你雇用他人或自己标记了一个足够大的数据集，那么神经网络便可以输出上述所有特征点，你可以利用它们实现其他有趣的效果，比如判断人物的动作姿态，识别图片中的人物表情等等。

以上就是特征点检测的内容，下节课我们将利用这些构造模块来构建对象检测算法。

3.3 目标检测（Object detection）

学过了对象定位和特征点检测，今天我们来构建一个对象检测算法。这节课，我们将学习如何通过卷积网络进行对象检测，采用的是基于滑动窗口的目标检测算法。



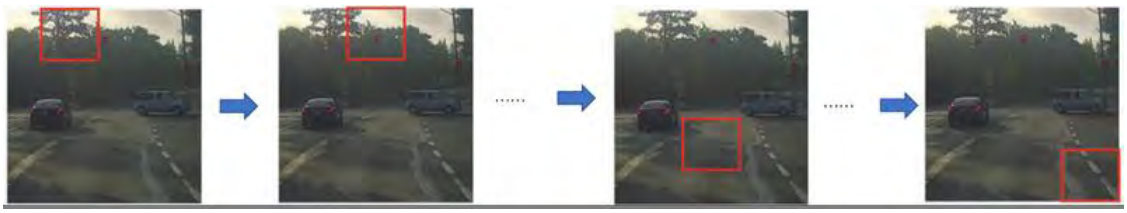
假如你想构建一个汽车检测算法，步骤是，首先创建一个标签训练集，也就是x和y表示适当剪切的汽车图片样本，这张图片（编号 1）x是一个**正样本？？？**，因为它是一辆汽车图片，这几张图片（编号 2、3）也有汽车，但这两张（编号 4、5）没有汽车。出于我们对这个训练集的期望，你一开始可以使用适当剪切的图片，就是整张图片x几乎都被汽车占据，你可以照张照片，然后剪切，剪掉汽车以外的部分，使汽车居于中间位置，并基本占据整张图片。有了这个标签训练集，你就可以开始训练卷积网络了，输入这些适当剪切过的图片（编号 6），卷积网络输出y，0 或 1 表示图片中有汽车或没有汽车。训练完这个卷积网络，就可以用它来实现滑动窗口目标检测，具体步骤如下。

假设这是一张测试图片，首先选定一个特定大小的窗口，比如图片下方这个窗口，将这个红色小方块输入卷积神经网络，卷积网络开始进行预测，即判断红色方框内有没有汽车。

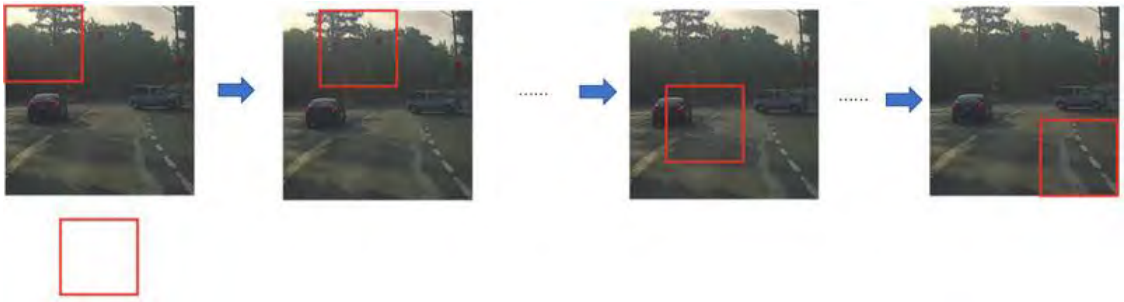
滑动窗口目标检测算法接下来会继续处理第二个图像，即红色方框稍向右滑动之后的区域，并输入给卷积网络，因此输入给卷积网络的只有红色方框内的区域，再次运行卷积网络，然后处理第三个图像，依次重复操作，直到这个窗口滑过图像的每一个角落。

为了滑动得更快，我这里选用的步幅比较大，思路是以固定步幅移动窗口，遍历图像的每个区域，把这些剪切后的小图像输入卷积网络，对每个位置按 0 或 1 进行分类，这就是所谓的图像滑动窗口操作。

重复上述操作，不过这次我们选择一个更大的窗口，截取更大的区域，并输入给卷积神经网络处理，你可以根据卷积网络对输入大小调整这个区域，然后输入给卷积网络，输出 0 或 1。



再以某个固定步幅滑动窗口，重复以上操作，遍历整个图像，输出结果。



然后第三次重复操作，这次选用更大的窗口。

如果你这样做，不论汽车在图片的什么位置，总有一个窗口可以检测到它。

比如，将这个窗口（编号 1）输入卷积网络，希望卷积网络对该输入区域的输出结果为 1，说明网络检测到图上有辆车。

这种算法叫作滑动窗口目标检测，因为我们以某个步幅滑动这些方框窗口遍历整张图片，对这些方形区域进行分类，判断里面有没有汽车。

滑动窗口目标检测算法也有很明显的缺点，就是计算成本，因为你在图片中剪切出太多小方块，卷积网络要一个个地处理。如果你选用的步幅很大，显然会减少输入卷积网络的窗口个数，但是粗糙间隔尺寸可能会影响性能。反之，如果采用小粒度或小步幅，传递给卷积网络的小窗口会特别多，这意味着超高的计算成本。

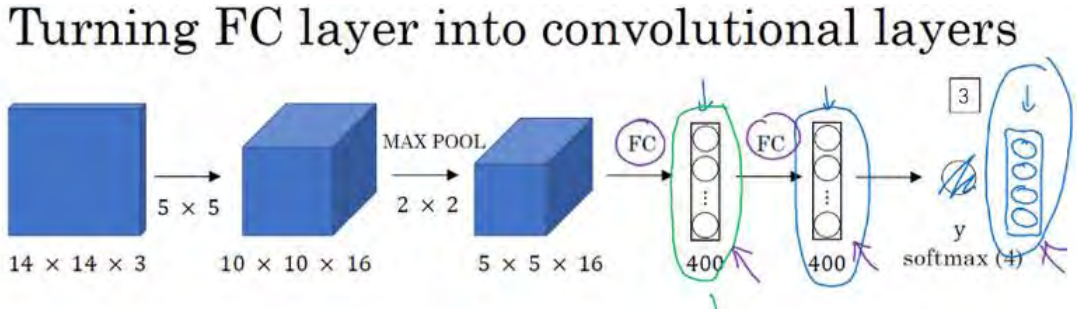
所以在神经网络兴起之前，人们通常采用更简单的分类器进行对象检测，比如通过采用手工处理工程特征的简单的线性分类器来执行对象检测。至于误差，因为每个分类器的计算成本都很低，它只是一个线性函数，所以滑动窗口目标检测算法表现良好，是个不错的算法。然而，卷积网络运行单个分类人物的成本却高得多，像这样滑动窗口太慢。除非采用超细粒度或极小步幅，否则无法准确定位图片中的对象。

不过，庆幸的是，计算成本问题已经有了很好的解决方案，大大提高了卷积层上应用滑动窗口目标检测器的效率，关于它的具体实现，我们下节课再讲。

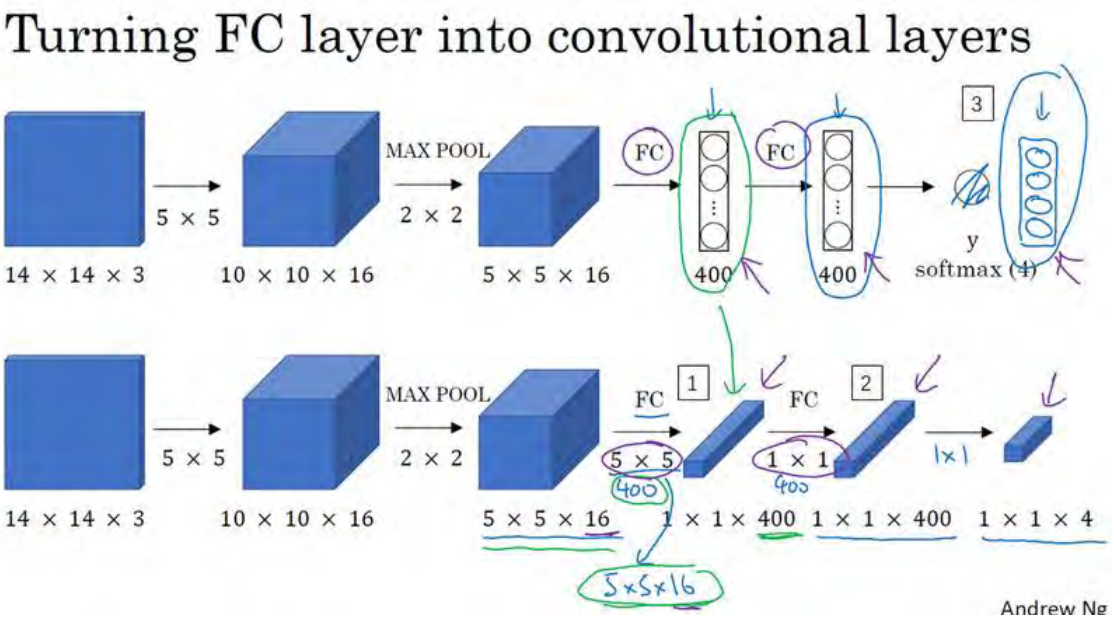
3.4 卷积的滑动窗口实现（Convolutional implementation of sliding windows）

上节课，我们学习了如何通过卷积网络实现滑动窗口对象检测算法，但效率很低。这节课我们讲讲如何在卷积层上应用这个算法。

为了构建滑动窗口的卷积应用，首先要知道如何把神经网络的全连接层转化成卷积层。我们先讲解这部分内容，下一张幻灯片，我们将按照这个思路来演示卷积的应用过程。



假设对象检测算法输入一个 14×14×3 的图像，图像很小，不过演示起来方便。在这里过滤器大小为 5×5，数量是 16，14×14×3 的图像在过滤器处理之后映射为 10×10×16。然后通过参数为 2×2 的最大池化操作，图像减小到 5×5×16。然后添加一个连接 400 个单元的全连接层，接着再添加一个全连接层，最后通过 **softmax** 单元输出y。为了跟下图区分开，我先做一点改动，用 4 个数字来表示y，它们分别对应 **softmax** 单元所输出的 4 个分类出现的概率。这 4 个分类可以是行人、汽车、摩托车和背景或其它对象。



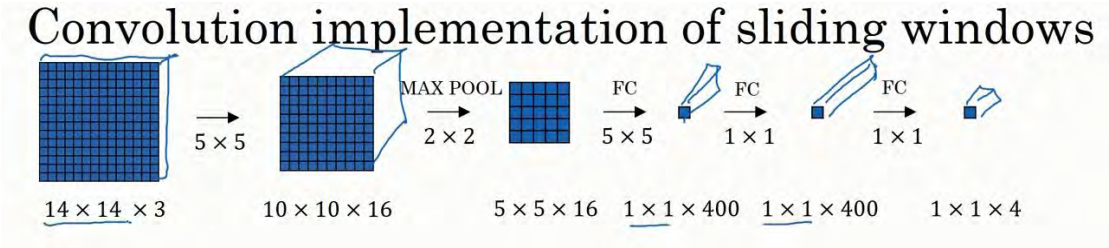
现在我要演示的就是如何把这些全连接层转化为卷积层，画一个这样的卷积网络，它的前几层和之前的一样，而对于下一层，也就是这个全连接层，我们可以用 5×5 的过滤器来实现，数量是 400 个（编号 1 所示），输入图像大小为 5×5×16，用 5×5 的过滤器对它进行卷积操作，过滤器实际上是 5×5×16，因为在卷积过程中，过滤器会遍历这 16 个通道，所以这两处的通道数量必须保持一致，输出结果为 1×1。假设应用 400 个这样的 5×5×16 过滤器，输出维度就是 1×1×400，我们不再把它看作一个含有 400 个节点的集合，而是一个 1×1×400 的输出层。从数学角度看，它和全连接层是一样的，因为这 400 个节点中每个节点都有一个 5×5×16 维度的过滤器，所以每个值都是上一层这些 5×5×16 激活值经过某个任意线性函数的输出结果。

我们再添加另外一个卷积层（编号 2 所示），这里用的是 1×1 卷积，假设有 400 个 1×1 的过滤器，在这 400 个过滤器的作用下，下一层的维度是 1×1×400，它其实就是上个网络中的这一全连接层。最后经由 1×1 过滤器的处理，得到一个 **softmax** 激活值，通过卷积网络，我们最终得到这个 1×1×4 的输出层，而不是这 4 个数字（编号 3 所示）。

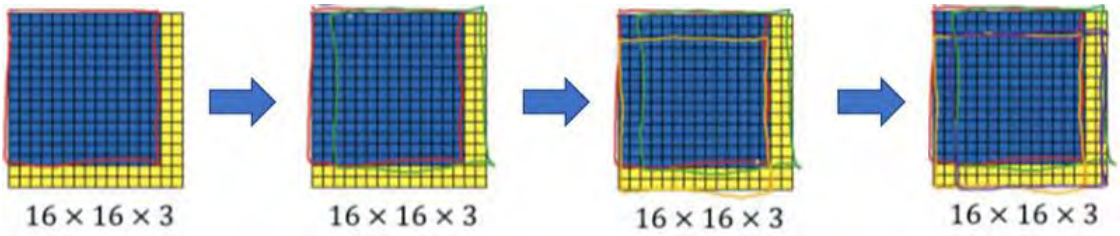
以上就是用卷积层代替全连接层的过程，结果这几个单元集变成了 1×1×400 和 1×1×4 的维度。

参考论文：**Sermanet, Pierre, et al. "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks." *Eprint Arxiv* (2013).**

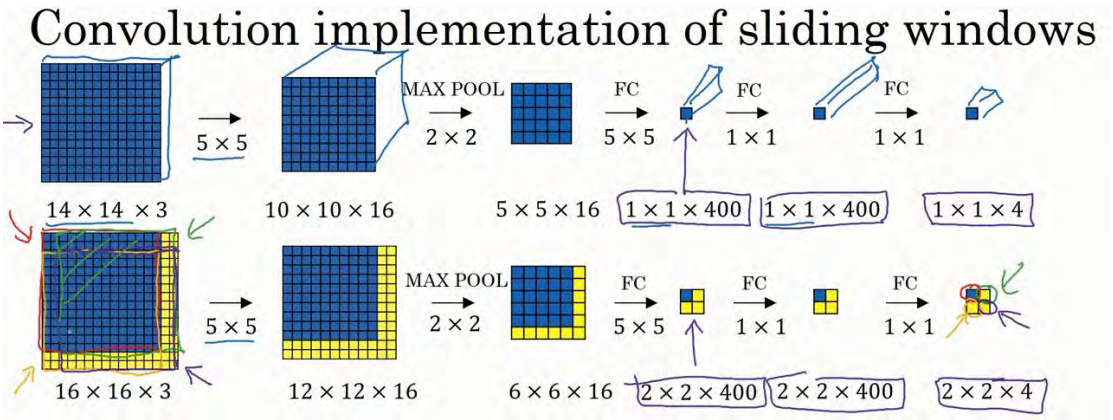
掌握了卷积知识，我们再看看如何通过卷积实现滑动窗口对象检测算法。讲义中的内容借鉴了屏幕下方这篇关于 **OverFeat** 的论文，它的作者包括 **Pierre Sermanet**，**David Eigen**，**张翔**，**Michael Mathieu**，**Rob Fergus**，**Yann LeCun**。



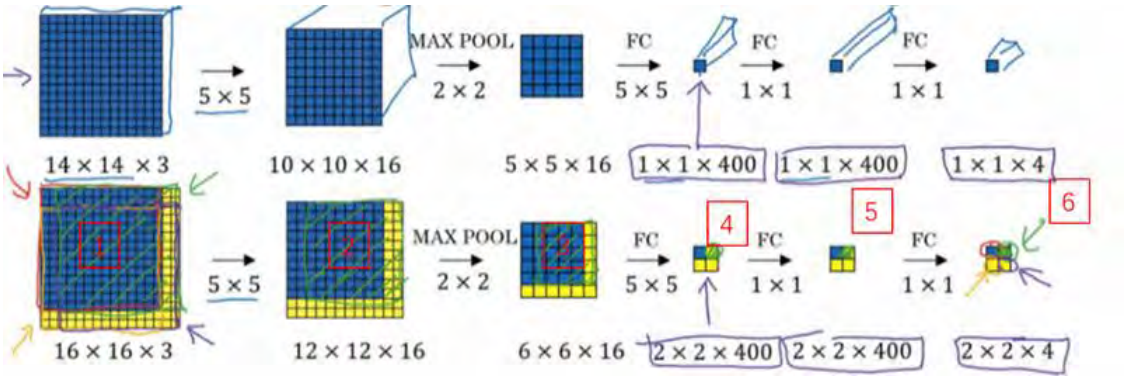
假设向滑动窗口卷积网络输入 14×14×3 的图片，为了简化演示和计算过程，这里我们依然用 14×14 的小图片。和前面一样，神经网络最后的输出层，即 **softmax** 单元的输出是 1×1×4，我画得比较简单，严格来说，14×14×3 应该是一个长方体，第二个 10×10×16 也是一个长方体，但为了方便，我只画了正面。所以，对于 1×1×400 的这个输出层，我也只画了它 1×1 的那一面，所以这里显示的都是平面图，而不是 3D 图像。



假设输入给卷积网络的图片大小是 14×14×3，测试集图片是 16×16×3，现在给这个输入图片加上黄色条块，在最初的滑动窗口算法中，你会把这片蓝色区域输入卷积网络（红色笔标记）生成 0 或 1 分类。接着滑动窗口，步幅为 2 个像素，向右滑动 2 个像素，将这个绿框区域输入给卷积网络，运行整个卷积网络，得到另外一个标签 0 或 1。继续将这个橘色区域输入给卷积网络，卷积后得到另一个标签，最后对右下方的紫色区域进行最后一次卷积操作。我们在这个 16×16×3 的小图像上滑动窗口，卷积网络运行了 4 次，于是输出了了 4 个标签。

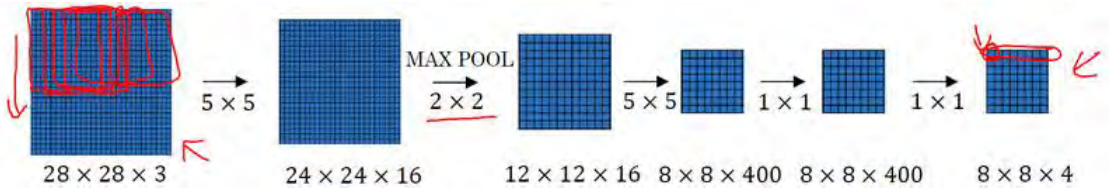


结果发现，这 4 次卷积操作中很多计算都是重复的。所以执行滑动窗口的卷积时使得卷积网络在这 4 次前向传播过程中共享很多计算，尤其是在这一步操作中（编号 1），卷积网络运行同样的参数，使得相同的 5×5×16 过滤器进行卷积操作，得到 12×12×16 的输出层。然后执行同样的最大池化（编号 2），输出结果 6×6×16。照旧应用 400 个 5×5 的过滤器（编号 3），得到一个 2×2×400 的输出层，现在输出层为 2×2×400，而不是 1×1×400。应用 1×1 过滤器（编号 4）得到另一个 2×2×400 的输出层。再做一次全连接的操作（编号 5），最终得到 2×2×4 的输出层，而不是 1×1×4。最终，在输出层这 4 个子方块中，蓝色的是图像左上部分 14×14 的输出（红色箭头标识），右上角方块是图像右上部分（绿色箭头标识）的对应输出，左下角方块是输入层左下角（橘色箭头标识），也就是这个 14×14 区域经过卷积网络处理后的结果，同样，右下角这个方块是卷积网络处理输入层右下角 14×14 区域(紫色箭头标识)的结果。

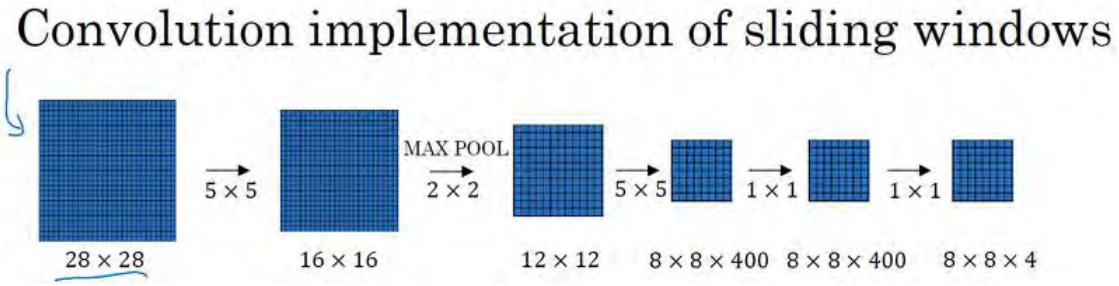


如果你想了解具体的计算步骤，以绿色方块为例，假设你剪切出这块区域（编号 1），传递给卷积网络，第一层的激活值就是这块区域（编号 2），最大池化后的下一层的激活值是这块区域（编号 3），这块区域对应着后面几层输出的右上角方块（编号 4，5，6）。

所以该卷积操作的原理是我们不需要把输入图像分割成四个子集，分别执行前向传播，而是把它们作为一张图片输入给卷积网络进行计算，其中的公共区域可以共享很多计算，就像这里我们看到的这个 4 个 14×14 的方块一样。

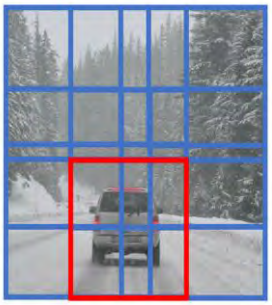


下面我们再看一个更大的图片样本，假如对一个 28×28×3 的图片应用滑动窗口操作，如果以同样的方式运行前向传播，最后得到 8×8×4 的结果。跟上一个范例一样，以 14×14 区域滑动窗口，首先在这个区域应用滑动窗口，其结果对应输出层的左上角部分。接着以大小为 2 的步幅不断地向右移动窗口，直到第 8 个单元格，得到输出层的第一行。然后向图片下方移动，最终输出这个 8×8×4 的结果。因为最大池化参数为 2，相当于以大小为 2 的步幅在原始图片上应用神经网络。



总结一下滑动窗口的实现过程，在图片上剪切出一块区域，假设它的大小是 14×14，把它输入到卷积网络。继续输入下一块区域，大小同样是 14×14，重复操作，直到某个区域识别到汽车。

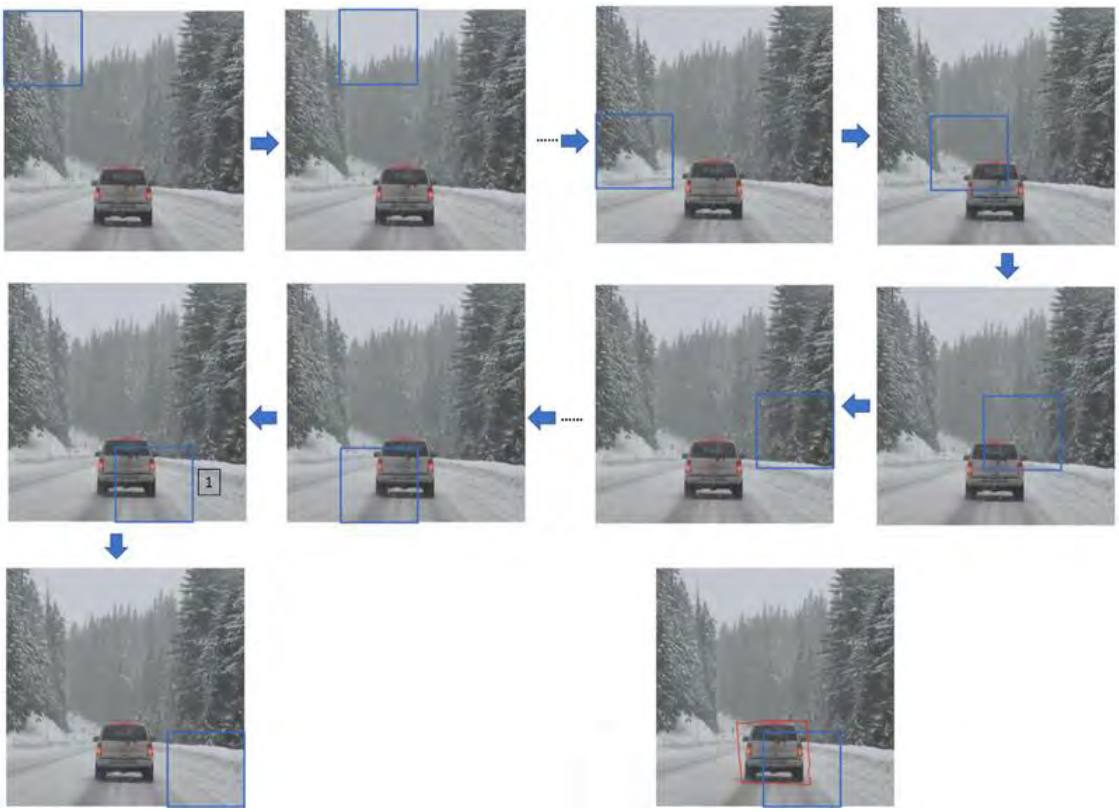
但是正如在前一页所看到的，我们不能依靠连续的卷积操作来识别图片中的汽车，比如，我们可以对大小为 28×28 的整张图片进行卷积操作，一次得到所有预测值，如果足够幸运，神经网络便可以识别出汽车的位置。



以上就是在卷积层上应用滑动窗口算法的内容，它提高了整个算法的效率。不过这种算法仍然存在一个缺点，就是边界框的位置可能不够准确。下节课，我们将学习如何解决这个问题。

3.5 Bounding Box 预测（Bounding box predictions）

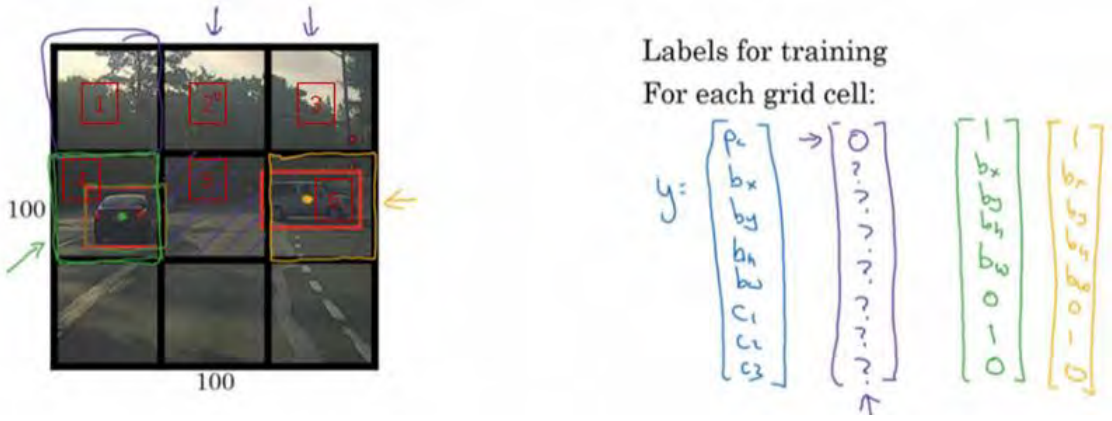
在上一个视频中，你们学到了滑动窗口法的卷积实现，这个算法效率更高，但仍然存在问题，不能输出最精准的边界框。在这个视频中，我们看看如何得到更精准的边界框。



在滑动窗口法中，你取这些离散的位置集合，然后在它们上运行分类器，在这种情况下，这些边界框没有一个能完美匹配汽车位置，也许这个框（编号 1）是最匹配的了。还有看起来这个真实值，最完美的边界框甚至不是方形，稍微有点长方形（红色方框所示），长宽比有点向水平方向延伸，有没有办法让这个算法输出更精准的边界框呢？

其中一个能得到更精准边界框的算法是 **YOLO** 算法，**YOLO(You only look once)**意思是你只看一次，这是由 **Joseph Redmon**，**Santosh Divvala**，**Ross Girshick** 和 **Ali Farhadi** 提出的算法。

是这么做的，比如你的输入图像是 100×100 的，然后在图像上放一个网格。为了介绍起来简单一些，我用 3×3 网格，实际实现时会用更精细的网格，可能是 19×19。基本思路是使用图像分类和定位算法，前几个视频介绍过的，然后将算法应用到 9 个格子上。（基本思路是，采用图像分类和定位算法，本周第一个视频中介绍过的，逐一应用在图像的 9 个格子中。）更具体一点，你需要这样定义训练标签，所以对于 9 个格子中的每一个指定一个标签 y，y是 8 维的，和你之前看到的一样，$y=\begin{bmatrix}p_c\\b_x\\b_y\\b_h\\b_w\\c_1\\c_2\\c_3\end{bmatrix}$，$p_c$等于 0 或 1 取决于这个绿色格子中是否有图像。然后b_x、b_y、b_h和b_w作用就是，如果那个格子里有对象，那么就给出边界框坐标。然后c_1、c_2和c_3就是你想要识别的三个类别，背景类别不算，所以你尝试在背景类别中识别行人、汽车和摩托车，那么c_1、c_2和c_3可以是行人、汽车和摩托车类别。这张图里有 9 个格子，所以对于每个格子都有这么一个向量。



我们看看左上方格子，这里这个（编号 1），里面什么也没有，所以左上格子的标签向量y是$\begin{bmatrix}0\\?\\?\\?\\?\\?\\?\\?\end{bmatrix}$。然后这个格子（编号 2）的输出标签$y$也是一样，这个格子（编号 3），还有其他

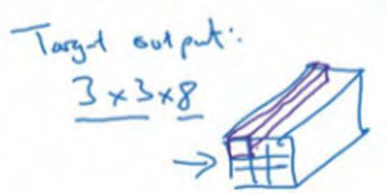
什么也没有的格子都一样。

现在这个格子呢？讲的更具体一点，这张图有两个对象，YOLO 算法做的就是，取两个对象的中点，然后将这个对象分配给包含对象中点的格子。所以左边的汽车就分配到这个格子上（编号 4），然后这辆 Condor（车型：神鹰）中点在这里，分配给这个格子（编号 6）。所以即使中心格子（编号 5）同时有两辆车的一部分，我们就假装中心格子没有任何我们感兴趣的对象，所以对于中心格子，分类标签 y 和这个向量类似，和这个没有对象的向量类似，

即$y=\begin{bmatrix}0\\?\\?\\?\\?\\?\\?\\?\end{bmatrix}$。而对于这个格子，这个用绿色框起来的格子（编号 4），目标标签就是这样的，

这里有一个对象，$p_c=1$，然后你写出b_x、b_y、b_h和b_w来指定边界框位置，然后还有类别 1 是行人，那么$c_1=0$，类别 2 是汽车，所以$c_2=1$，类别 3 是摩托车，则数值$c_3=0$，即$y=\begin{bmatrix}1\\b_x\\b_y\\b_h\\b_w\\0\\1\\0\end{bmatrix}$。右边这个格子（编号 6）也是类似的，因为这里确实有一个对象，它的向量应该是这个样子的，$y=\begin{bmatrix}1\\b_x\\b_y\\b_h\\b_w\\0\\1\\0\end{bmatrix}$作为目标向量对应右边的格子。

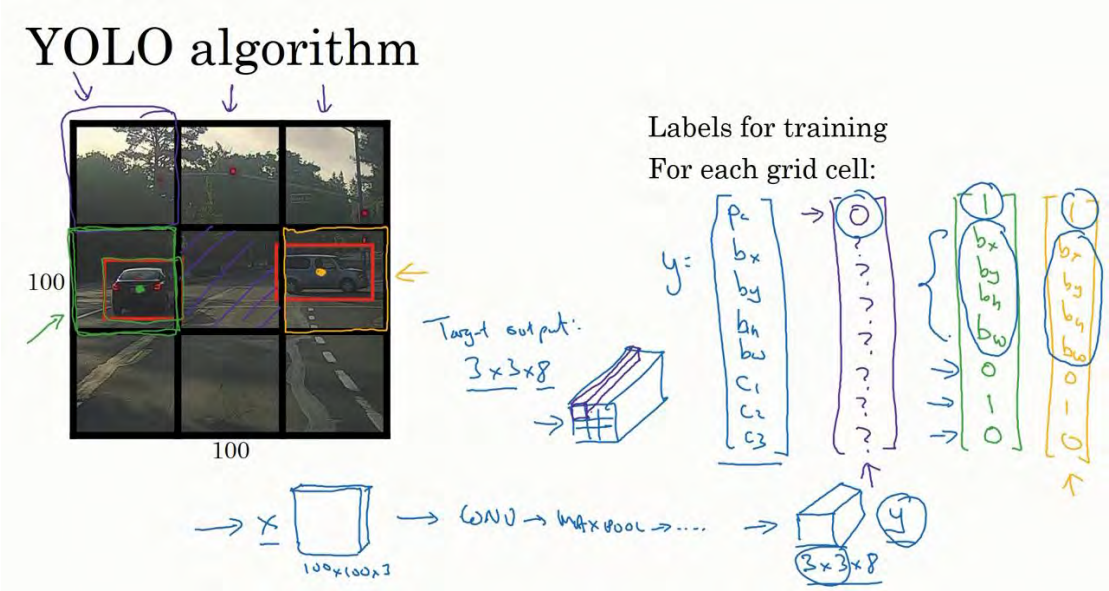
所以对于这里 9 个格子中任何一个，你都会得到一个 8 维输出向量，因为这里是 3×3 的网格，所以有 9 个格子，总的输出尺寸是 3×3×8，所以目标输出是 3×3×8。因为这里有 3×3 格子，然后对于每个格子，你都有一个 8 维向量y，所以目标输出尺寸是 3×3×8。



对于这个例子中，左上格子是 1×1×8，对应的是 9 个格子中左上格子的输出向量。所以

对于这 3×3 中每一个位置而言，对于这 9 个格子，每个都对应一个 8 维输出目标向量y，其中一些值可以是 **dont care-s**（即？），如果这里没有对象的话。所以总的目标输出，这个图片的输出标签尺寸就是 3×3×8。

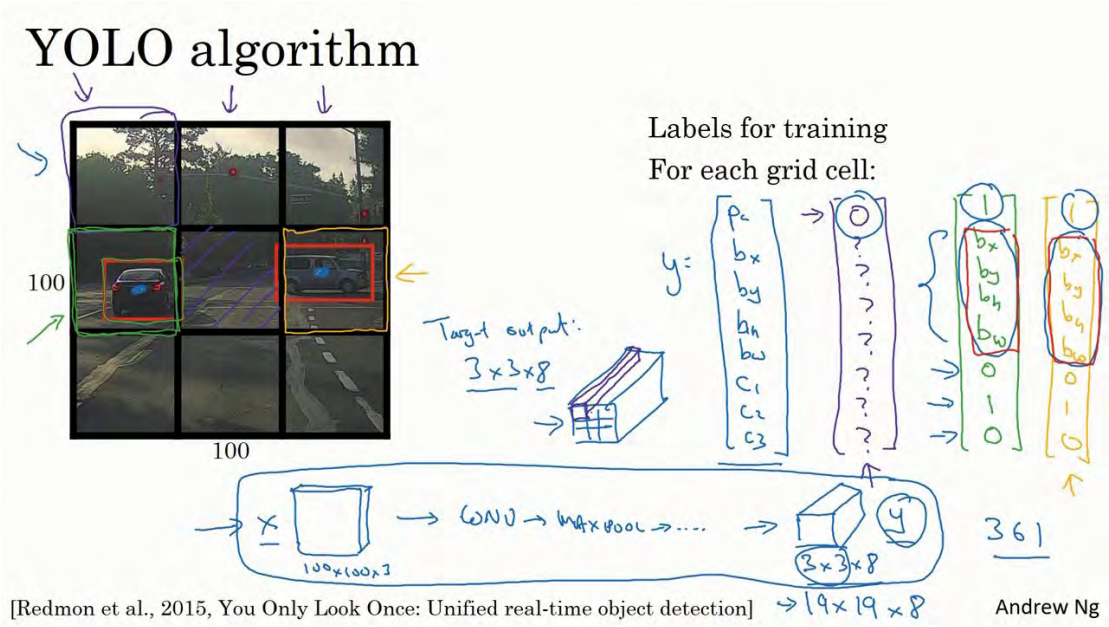
如果你现在要训练一个输入为 100×100×3 的神经网络，现在这是输入图像，然后你有一个普通的卷积网络，卷积层，最大池化层等等，最后你会有这个，选择卷积层和最大池化层，这样最后就映射到一个 3×3×8 输出尺寸。所以你要做的是，有一个输入x，就是这样的输入图像，然后你有这些 3×3×8 的目标标签y。当你用反向传播训练神经网络时，将任意输入x映射到这类输出向量y。



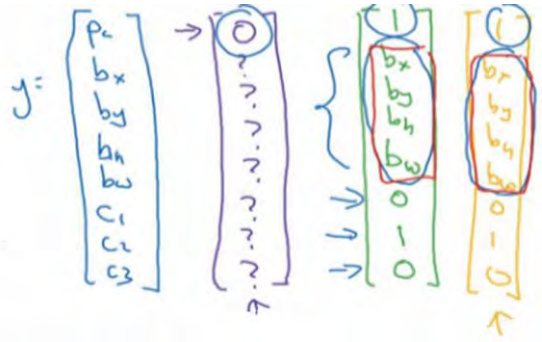
所以这个算法的优点在于神经网络可以输出精确的边界框，所以测试的时候，你做的是喂入输入图像x，然后跑正向传播，直到你得到这个输出y。然后对于这里 3×3 位置对应的 9 个输出，我们在输出中展示过的，你就可以读出 1 或 0（编号 1 位置），你就知道 9 个位置之一有个对象。如果那里有个对象，那个对象是什么（编号 3 位置），还有格子中这个对象的边界框是什么（编号 2 位置）。只要每个格子中对象数目没有超过 1 个，这个算法应该是没问题的。一个格子中存在多个对象的问题，我们稍后再讨论。但实践中，我们这里用的是比较小的 3×3 网格，实践中你可能会使用更精细的 19×19 网格，所以输出就是 19×19×8。这样的网格精细得多，那么多个对象分配到同一个格子得概率就小得多。

重申一下，把对象分配到一个格子的过程是，你观察对象的中点，然后将这个对象分配

到其中点所在的格子，所以即使对象可以横跨多个格子，也只会被分配到 9 个格子其中之一，就是 3×3 网络的其中一个格子，或者 19×19 网络的其中一个格子。在 19×19 网格中，两个对象的中点（图中蓝色点所示）处于同一个格子的概率就会更低。



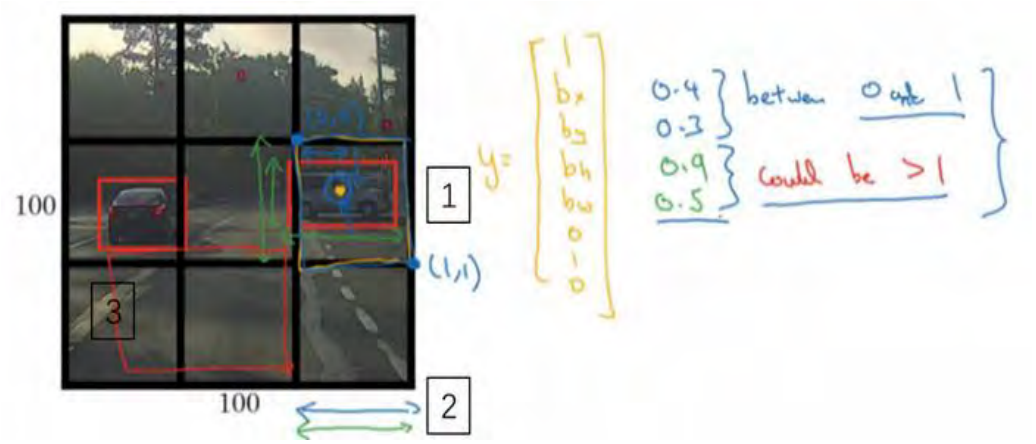
所以要注意，首先这和图像分类和定位算法非常像，我们在本周第一节课讲过的，就是它显式地输出边界框坐标，所以这能让神经网络输出边界框，可以具有任意宽高比，并且能输出更精确的坐标，不会受到滑动窗口分类器的步长大小限制。其次，这是一个卷积实现，你并没有在 3×3 网格上跑 9 次算法，或者，如果你用的是 19×19 的网格，19 平方是 361 次，所以你不需要让同一个算法跑 361 次。相反，这是单次卷积实现，但你使用了一个卷积网络，有很多共享计算步骤，在处理这 3×3 计算中很多计算步骤是共享的，或者你的 19×19 的网格，所以这个算法效率很高。



事实上 **YOLO** 算法有一个好处，也是它受欢迎的原因，因为这是一个卷积实现，实际上它的运行速度非常快，可以达到实时识别。在结束之前我还想给你们分享一个小细节，如何编码这些边界框b_x、b_y、b_h和b_w，我们在下一张幻灯片上讨论。

这里有两辆车，我们有个 3×3 网格，我们以右边的车为例（编号 1），红色格子里有个对象，所以目标标签y就是，$p_c = 1$，然后b_x、b_y、b_h和b_w，然后$c_1 = 0$，$c_2 = 1$，$c_3 = 0$，即$y = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$。你怎么指定这个边界框呢？

Specify the bounding boxes：



在 **YOLO** 算法中，对于这个方框（编号 1 所示），我们约定左上这个点是(0,0)，然后右下这个点是(1,1),要指定橙色中点的位置，b_x大概是 0.4，因为它的位置大概是水平长度的 0.4，然后b_y大概是 0.3，然后边界框的高度用格子总体宽度的比例表示，所以这个红框的宽度可能是蓝线（编号 2 所示的蓝线）的 90%，所以b_h是 0.9，它的高度也许是格子总体高度的一半，这样的话b_w就是 0.5。换句话说，b_x、b_y、b_h和b_w单位是相对于格子尺寸的比列，所以b_x和b_y必须在 0 和 1 之间，因为从定义上看，橙色点位于对象分配到格子的范围内，如果它不在 0 和 1 之间，如果它在方块外，那么这个对象就应该分配到另一个格子上。这个值（b_h和b_w）可能会大于 1，特别是如果有一辆汽车的边界框是这样的（编号 3 所示），那么边界框的宽度和高度有可能大于 1。

指定边界框的方式有很多，但这种约定是比较合理的，如果你去读 **YOLO** 的研究论文，**YOLO** 的研究工作有其他参数化的方式，可能效果会更好，我这里就只给出了一个合理的约定，用起来应该没问题。不过还有其他更复杂的参数化方式，涉及到 **sigmoid** 函数，确保这个值（b_x和b_y）介于 0 和 1 之间，然后使用指数参数化来确保这些（b_h和b_w）都是非负数，因为 0.9 和 0.5，这个必须大于等于 0。还有其他更高级的参数化方式，可能效果要更好一点，但我这里讲的办法应该是管用的。

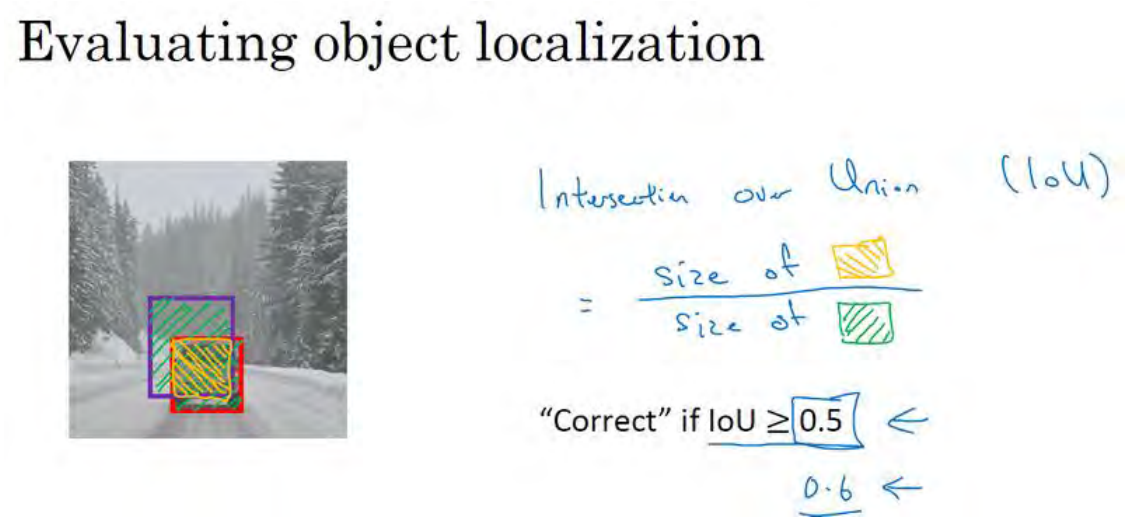
这就是 **YOLO** 算法，你只看一次算法，在接下来的几个视频中，我会告诉你一些其他的思路可以让这个算法做的更好。在此期间，如果你感兴趣，也可以看看 **YOLO** 的论文，在前几张幻灯片底部引用的 **YOLO** 论文。

Redmon, Joseph, et al. "You Only Look Once: Unified, Real-Time Object Detection." (2015):779-788.

不过看这些论文之前，先给你们提个醒，**YOLO** 论文是相对难度较高的论文之一，我记得我第一次读这篇论文的时候，我真的很难搞清楚到底是怎么实现的，我最后问了一些我认识的研究员，看看他们能不能给我讲清楚，即使是他们，也很难理解这篇论文的一些细节。所以如果你看论文的时候，发现看不懂，这是没问题的，我希望这种场合出现的概率要更低才好，但实际上，即使是资深研究员也有读不懂研究论文的时候，必须去读源代码，或者联系作者之类的才能弄清楚这些算法的细节。但你们不要被我吓到，你们可以自己看看这些论文，如果你们感兴趣的话，但这篇论文相对较难。现在你们了解了 **YOLO** 算法的基础，我们继续讨论别的让这个算法效果更好的研究。

3.6 交并比（Intersection over union）

你如何判断对象检测算法运作良好呢？在本视频中，你将了解到并交比函数，可以用来评价对象检测算法。在下一个视频中，我们用它来插入一个分量来进一步改善检测算法，我们开始吧。



在对象检测任务中，你希望能够同时定位对象，所以如果实际边界框是这样的，你的算法给出这个紫色的边界框，那么这个结果是好还是坏？所以交并比（**IoU**）函数做的是计算两个边界框交集和并集之比。两个边界框的并集是这个区域，就是属于包含两个边界框区域（绿色阴影表示区域），而交集就是这个比较小的区域（橙色阴影表示区域），那么交并比就是交集的大小，这个橙色阴影面积，然后除以绿色阴影的并集面积。

一般约定，在计算机检测任务中，如果$IoU \geq 0.5$，就说检测正确，如果预测器和实际边界框完美重叠，**IoU** 就是 1，因为交集就等于并集。但一般来说只要$IoU \geq 0.5$，那么结果是可以接受的，看起来还可以。一般约定，0.5 是阈值，用来判断预测的边界框是否正确。一般是这么约定，但如果你希望更严格一点，你可以将 **IoU** 定得更高，比如说大于 0.6 或者更大的数字，但 **IoU** 越高，边界框越精确。

所以这是衡量定位精确度的一种方式，你只需要统计算法正确检测和定位对象的次数，你就可以用这样的定义判断对象定位是否准确。再次，0.5 是人为约定，没有特别深的理论依据，如果你想更严格一点，可以把阈值定为 0.6。有时我看到更严格的标准，比如 0.6 甚至 0.7，但很少见到有人将阈值降到 0.5 以下。

人们定义 **IoU** 这个概念是为了评价你的对象定位算法是否精准，但更一般地说，IoU 衡量了两个边界框重叠地相对大小。如果你有两个边界框，你可以计算交集，计算并集，然后

求两个数值的比值，所以这也可以判断两个边界框是否相似，我们将在下一个视频中再次用到这个函数，当我们讨论非最大值抑制时再次用到。

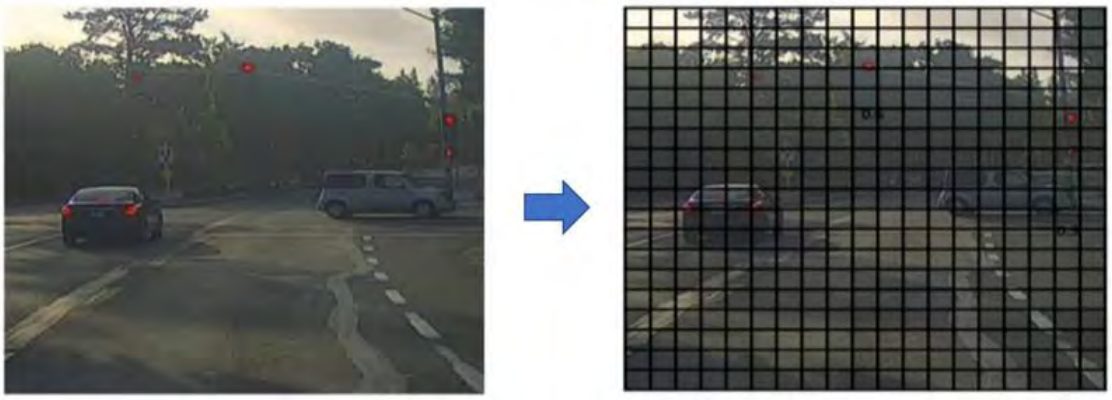
More generally, IoU is a measure of the overlap between two bounding boxes.

好，这就是 **IoU**，或者说交并比，不要和借据中提到的我欠你钱的这个概念所混淆，如果你借钱给别人，他们会写给你一个借据，说：“我欠你这么多钱（**I own you this much money**）。”，这也叫做 **IoU**。这是完全不同的概念，这两个概念重名。

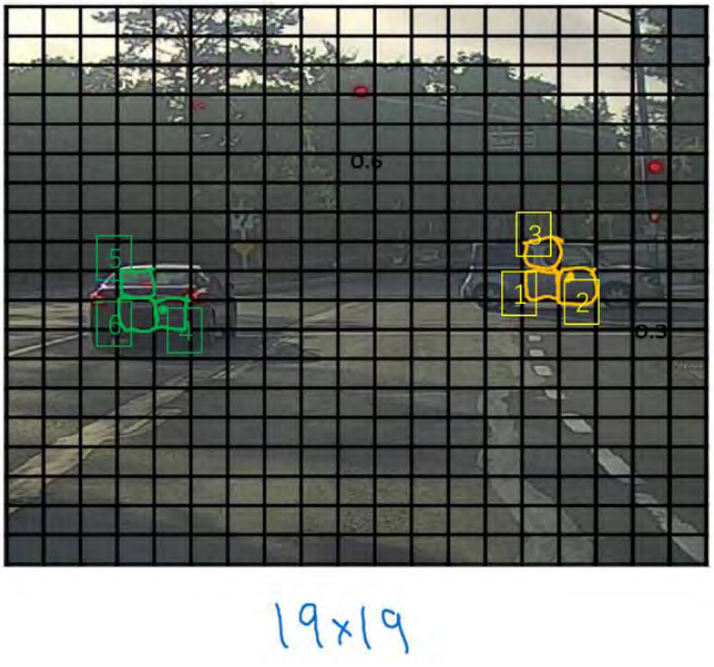
现在介绍了 **IoU** 交并比的定义之后，在下一个视频中，我想讨论非最大值抑制，这个工具可以让 **YOLO** 算法输出效果更好，我们下一个视频继续。

3.7 非极大值抑制（Non-max suppression）

到目前为止你们学到的对象检测中的一个问题是，你的算法可能对同一个对象做出多次检测，所以算法不是对某个对象检测出一次，而是检测出多次。非极大值抑制这个方法可以确保你的算法对每个对象只检测一次，我们讲一个例子。



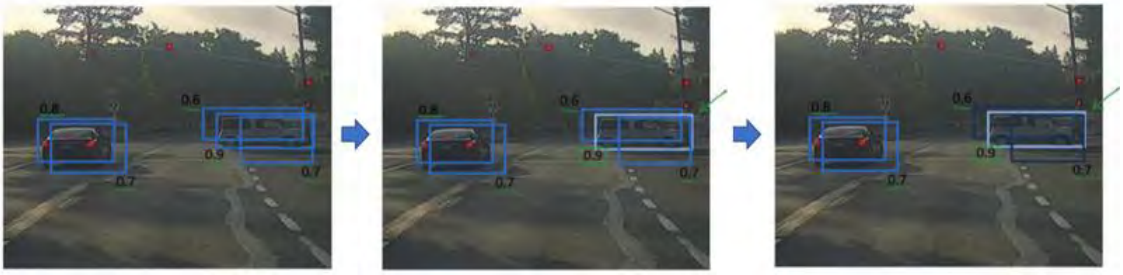
假设你需要在这张图片里检测行人和汽车，你可能会在上面放个 19×19 网格，理论上这辆车只有一个中点，所以它应该只被分配到一个格子里，左边的车子也只有一个中点，所以理论上应该只有一个格子做出有车的预测。



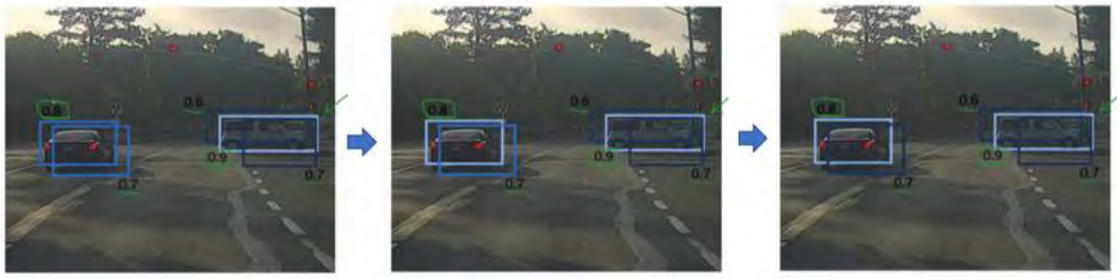
实践中当你运行对象分类和定位算法时，对于每个格子都运行一次，所以这个格子（编号 1）可能会认为这辆车中点应该在格子内部，这几个格子（编号 2、3）也会这么认为。对于左边的车子也一样，所以不仅仅是这个格子，如果这是你们以前见过的图像，不仅这个格（编号 4）子会认为它里面有车，也许这个格子（编号 5）和这个格子（编号 6）也会，也许

其他格子也会这么认为，觉得它们格子内有车。

我们分步介绍一下非极大抑制是怎么起效的，因为你要在 361 个格子上都运行一次图像检测和定位算法，那么可能很多格子都会举手说我的p_c，我这个格子里有车的概率很高，而不是 361 个格子中仅有两个格子会报告它们检测出一个对象。所以当你运行算法的时候，最后可能会对同一个对象做出多次检测，所以非极大值抑制做的就是清理这些检测结果。这样一辆车只检测一次，而不是每辆车都触发多次检测。



所以具体上，这个算法做的是，首先看看每次报告每个检测结果相关的概率p_c，在本周的编程练习中有更多细节，实际上是p_c乘以c_1、c_2或c_3。现在我们就说，这个p_c检测概率，首先看概率最大的那个，这个例子（右边车辆）中是 0.9，然后就说这是最可靠的检测，所以我们就用高亮标记，就说我这里找到了一辆车。这么做之后，非极大值抑制就会逐一审视剩下的矩形，所有和这个最大的边框有很高交并比，高度重叠的其他边界框，那么这些输出就会被抑制。所以这两个矩形p_c分别是 0.6 和 0.7，这两个矩形和淡蓝色矩形重叠程度很高，所以会被抑制，变暗，表示它们被抑制了。

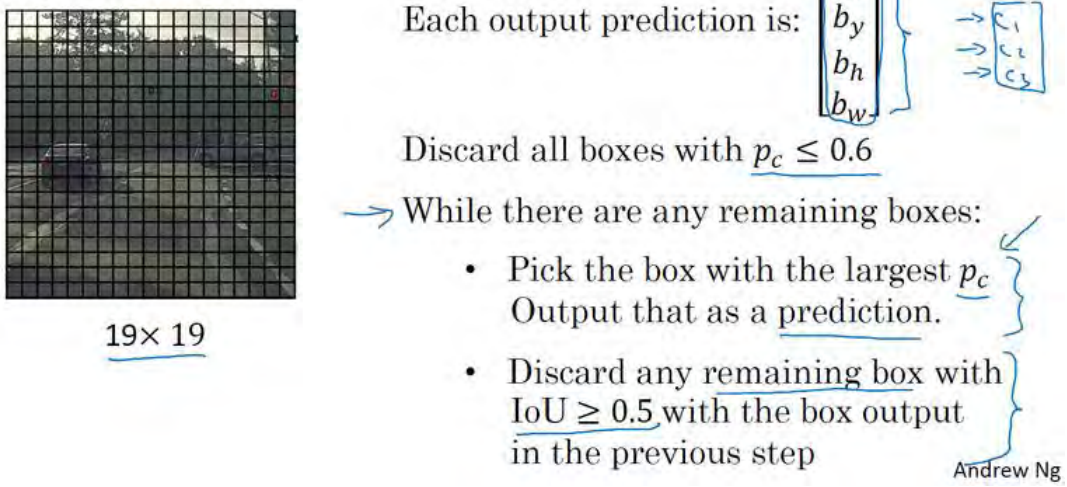


接下来，逐一审视剩下的矩形，找出概率最高，p_c最高的一个，在这种情况下是 0.8，我们就认为这里检测出一辆车（左边车辆），然后非极大值抑制算法就会去掉其他 IoU 值很高的矩形。所以现在每个矩形都会被高亮显示或者变暗，如果你直接抛弃变暗的矩形，那就剩下高亮显示的那些，这就是最后得到的两个预测结果。

所以这就是非极大值抑制，非最大值意味着你只输出概率最大的分类结果，但抑制很接近，但不是最大的其他预测结果，所以这方法叫做非极大值抑制。

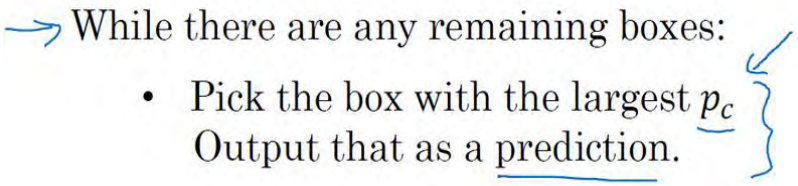
我们来看看算法的细节，首先这个 19×19 网格上执行一下算法，你会得到 19×19×8 的输

出尺寸。不过对于这个例子来说，我们简化一下，就说你只做汽车检测，我们就去掉c_1、c_2和c_3，然后假设这条线对于 19×19 的每一个输出，对于 361 个格子的每个输出，你会得到这样的输出预测，就是格子中有对象的概率（p_c），然后是边界框参数（b_x、b_y、b_h和b_w）。如果你只检测一种对象，那么就没有c_1、c_2和c_3这些预测分量。多个对象处于同一个格子中的情况，我会放到编程练习中，你们可以在本周末之前做做。

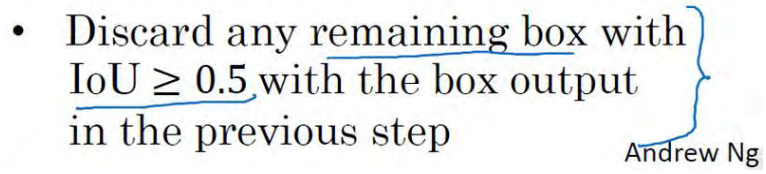


现在要实现非极大值抑制，你可以做的第一件事是，去掉所有边界框，我们就将所有的预测值，所有的边界框p_c小于或等于某个阈值，比如≤0.6 的边界框去掉。

我们就这样说，除非算法认为这里存在对象的概率至少有 0.6，否则就抛弃，所以这就抛弃了所有概率比较低的输出边界框。所以思路是对于这 361 个位置，你输出一个边界框，还有那个最好边界框所对应的概率，所以我们只是抛弃所有低概率的边界框。



接下来剩下的边界框，没有抛弃没有处理过的，你就一直选择概率p_c最高的边界框，然后把它输出成预测结果，这个过程就是上一张幻灯片，取一个边界框，让它高亮显示，这样你就可以确定输出做出有一辆车的预测。



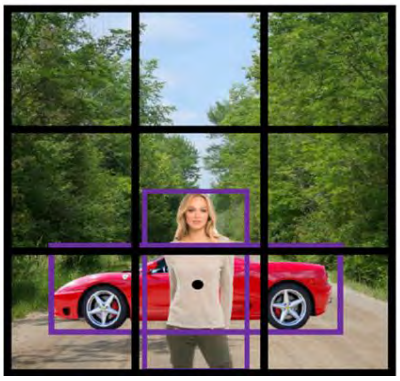
接下来去掉所有剩下的边界框，任何没有达到输出标准的边界框，之前没有抛弃的边界框，把这些和输出边界框有高重叠面积和上一步输出边界框有很高交并比的边界框全部抛弃。所以 **while** 循环的第二步是上一张幻灯片变暗的那些边界框，和高亮标记的边界重叠面积很高的那些边界框抛弃掉。在还有剩下边界框的时候，一直这么做，把没处理的都处理完，直到每个边界框都判断过了，它们有的作为输出结果，剩下的会被抛弃，它们和输出结果重叠面积太高，和输出结果交并比太高，和你刚刚输出这里存在对象结果的重叠程度过高。

在这张幻灯片中，我只介绍了算法检测单个对象的情况，如果你尝试同时检测三个对象，比如说行人、汽车、摩托，那么输出向量就会有三个额外的分量。事实证明，正确的做法是独立进行三次非极大值抑制，对每个输出类别都做一次，但这个细节就留给本周的编程练习吧，其中你可以自己尝试实现，我们可以自己试试在多个对象类别检测时做非极大值抑制。

这就是非极大值抑制，如果你能实现我们说过的对象检测算法，你其实可以得到相当不错的结果。但结束我们对 **YOLO** 算法的介绍之前，最后我还有一个细节想给大家分享，可以进一步改善算法效果，就是 **anchor box** 的思路，我们下一个视频再介绍。

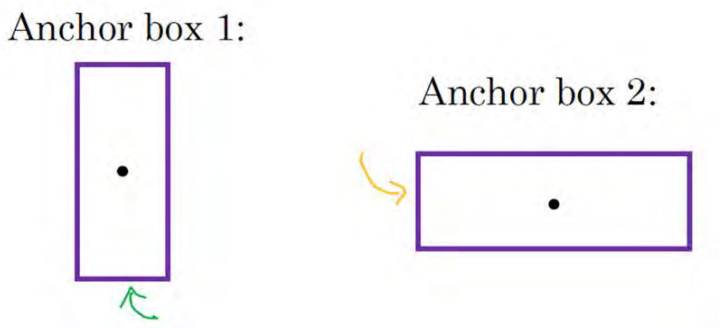
3.8 Anchor Boxes

到目前为止，对象检测中存在的一个问题是每个格子只能检测出一个对象，如果你想让一个格子检测出多个对象，你可以这么做，就是使用 **anchor box** 这个概念，我们从一个例子开始讲吧。

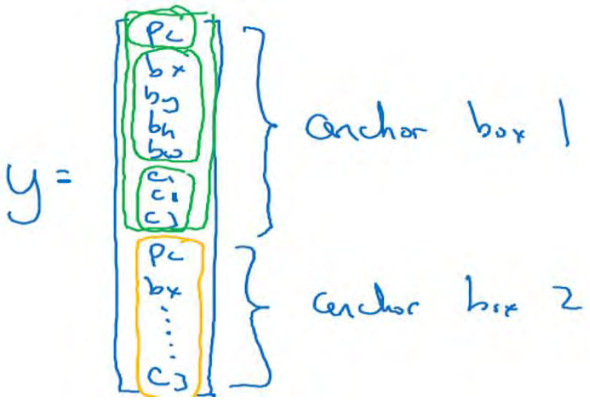


$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

假设你有这样一张图片，对于这个例子，我们继续使用 3×3 网格，注意行人的中点和汽车的中点几乎在同一个地方，两者都落入到同一个格子中。所以对于那个格子，如果 y 输出这个向量$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$，你可以检测这三个类别，行人、汽车和摩托车，它将无法输出检测结果，所以我必须从两个检测结果中选一个。



而 **anchor box** 的思路是，这样子，预先定义两个不同形状的 **anchor box**，或者 **anchor box** 形状，你要做的是把预测结果和这两个 **anchor box** 关联起来。一般来说，你可能会用更多的 **anchor box**，可能要 5 个甚至更多，但对于这个视频，我们就用两个 **anchor box**，这样介绍起来简单一些。

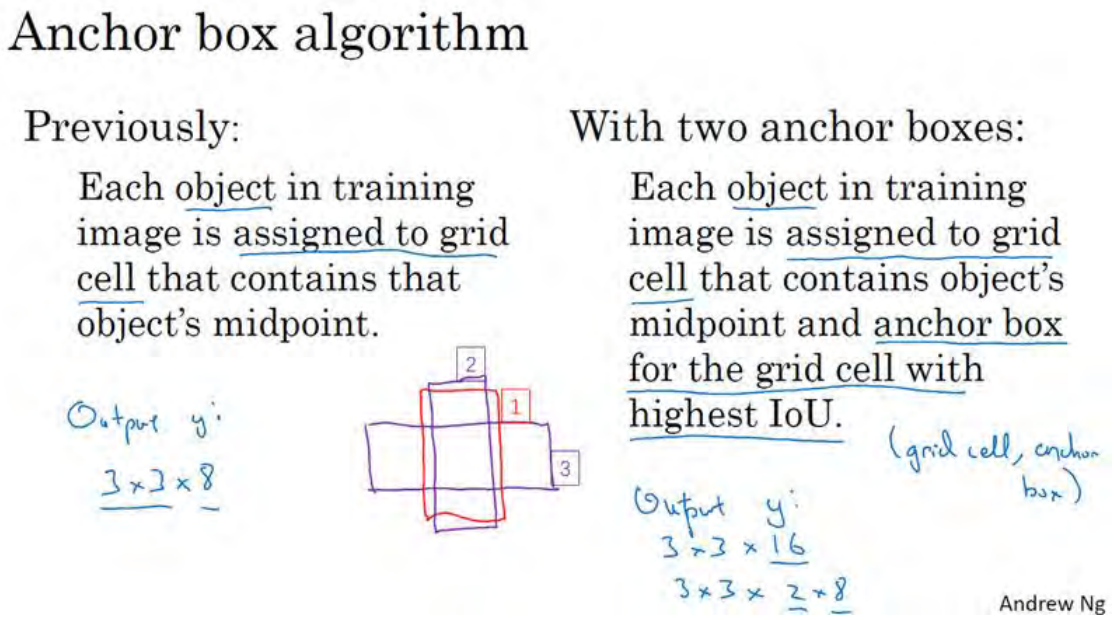


你要做的是定义类别标签，用的向量不再是上面这个$[p_c \quad b_x \quad b_y \quad b_h \quad b_w \quad c_1 \quad c_2 \quad c_3]^T$，

而是重复两次：

$y = [p_c \quad b_x \quad b_y \quad b_h \quad b_w \quad c_1 \quad c_2 \quad c_3 \quad p_c \quad b_x \quad b_y \quad b_h \quad b_w \quad c_1 \quad c_2 \quad c_3]^T$，前面的$p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3$（绿色方框标记的参数）是和 **anchor box 1** 关联的 8 个参数，后面的 8 个参数（橙色方框标记的元素）是和 anchor box 2 相关联。因为行人的形状更类似于 **anchor box 1** 的形状，而不是 **anchor box 2** 的形状，所以你可以用这 8 个数值（前 8 个参数），这么编码$p_c = 1$，是的，代表有个行人，用b_x, b_y, b_h和b_w来编码包住行人的边界框，然后用$c_1, c_2, c_3 (c_1 = 1, c_2 = 0, c_3 = 0)$来说明这个对象是个行人。

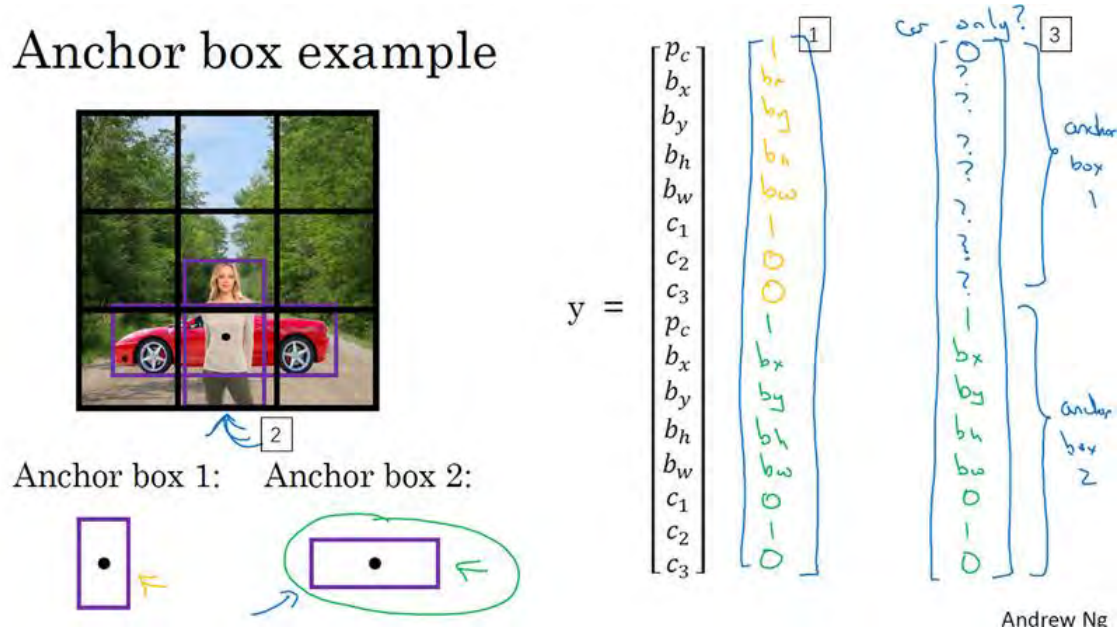
然后是车子，因为车子的边界框比起 **anchor box 1** 更像 **anchor box 2** 的形状，你就可以这么编码，这里第二个对象是汽车，然后有这样的边界框等等，这里所有参数都和检测汽车相关$(p_c = 1, b_x, b_y, b_h, b_w, c_1 = 0, c_2 = 1, c_3 = 0)$。



总结一下，用 **anchor box** 之前，你做的是这个，对于训练集图像中的每个对象，都根据

那个对象中点位置分配到对应的格子中，所以输出 y 就是 3×3×8，因为是 3×3 网格，对于每个网格位置，我们有输出向量，包含p_c，然后边界框参数b_x, b_y, b_h和b_w，然后c_1, c_2, c_3。

现在用到 anchor box 这个概念，是这么做的。现在每个对象都和之前一样分配到同一个格子中，分配到对象中点所在的格子中，以及分配到和对象形状交并比最高的 **anchor box** 中。所以这里有两个 **anchor box**，你就取这个对象，如果你的对象形状是这样的（编号 1，红色框），你就看看这两个 **anchor box**，**anchor box 1** 形状是这样（编号 2，紫色框），**anchor box 2** 形状是这样（编号 3，紫色框），然后你观察哪一个 **anchor box** 和实际边界框（编号 1，红色框）的交并比更高，不管选的是哪一个，这个对象不只分配到一个格子，而是分配到一对，即（**grid cell**，**anchor box**）对，这就是对象在目标标签中的编码方式。所以现在输出 y 就是 3×3×16，上一张幻灯片中你们看到 y 现在是 16 维的，或者你也可以看成是 3×3×2×8，因为现在这里有 2 个 **anchor box**，而 y 是 8 维的。y 维度是 8，因为我们有 3 个对象类别，如果你有更多对象，那么y 的维度会更高。



所以我们来看一个具体的例子，对于这个格子（编号 2），我们定义一下y，$y = [p_c \quad b_x \quad b_y \quad b_h \quad b_w \quad c_1 \quad c_2 \quad c_3 \quad p_c \quad b_x \quad b_y \quad b_h \quad b_w \quad c_1 \quad c_2 \quad c_3]^T$。所以行人更类似于 **anchor box 1** 的形状，所以对于行人来说，我们将她分配到向量的上半部分。是的，这里存在一个对象，即$p_c = 1$，有一个边界框包住行人，如果行人是类别 1，那么 $c_1 = 1, c_2 = 0, c_3 = 0$（编号 1 所示的橙色参数）。车子的形状更像 **anchor box 2**，所以这个向量剩下的部分是 $p_c = 1$，然后和车相关的边界框，然后$c_1 = 0, c_2 = 1, c_3 = 0$（编号 1 所示的绿色参数）。所以这就是对应中下格子的标签 y，这个箭头指向的格子（编号 2 所示）。

现在其中一个格子有车，没有行人，如果它里面只有一辆车，那么假设车子的边界框形

状是这样，更像 **anchor box 2**，如果这里只有一辆车，行人走开了，那么 **anchor box 2** 分量还是一样的，要记住这是向量对应 **anchor box 2** 的分量和 **anchor box 1** 对应的向量分量，你要填的就是，里面没有任何对象，所以 $p_c = 0$，然后剩下的就是 **don't care-s**(即？)（编号 3 所示）。

现在还有一些额外的细节，如果你有两个 **anchor box**，但在同一个格子中有三个对象，这种情况算法处理不好，你希望这种情况不会发生，但如果真的发生了，这个算法并没有很好的处理办法，对于这种情况，我们就引入一些打破僵局的默认手段。还有这种情况，两个对象都分配到一个格子中，而且它们的 anchor box 形状也一样，这是算法处理不好的另一种情况，你需要引入一些打破僵局的默认手段，专门处理这种情况，希望你的数据集里不会出现这种情况，其实出现的情况不多，所以对性能的影响应该不会很大。

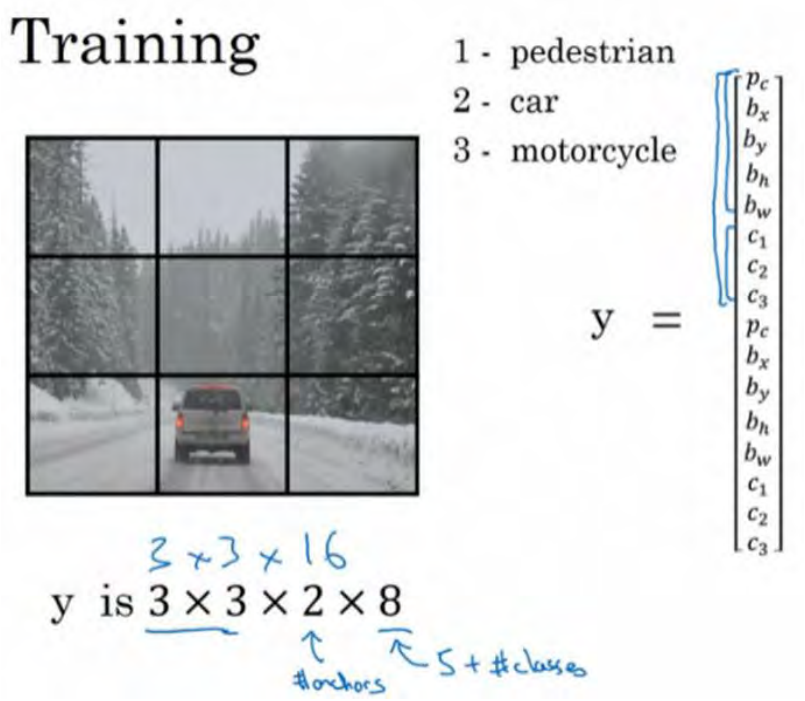
这就是 **anchor box** 的概念，我们建立 **anchor box** 这个概念，是为了处理两个对象出现在同一个格子的情况，实践中这种情况很少发生，特别是如果你用的是 19×19 网格而不是 3×3 的网格，两个对象中点处于 361 个格子中同一个格子的概率很低，确实会出现，但出现频率不高。也许设立 **anchor box** 的好处在于 **anchor box** 能让你的学习算法能够更有征对性，特别是如果你的数据集有一些很高很瘦的对象，比如说行人，还有像汽车这样很宽的对象，这样你的算法就能更有针对性的处理，这样有一些输出单元可以针对检测很宽很胖的对象，比如说车子，然后输出一些单元，可以针对检测很高很瘦的对象，比如说行人。

最后，你应该怎么选择 **anchor box** 呢？人们一般手工指定 **anchor box** 形状，你可以选择 5 到 10 个 **anchor box** 形状，覆盖到多种不同的形状，可以涵盖你想要检测的对象的各种形状。还有一个更高级的版本，我就简单说一句，你们如果接触过一些机器学习，可能知道后期 **YOLO** 论文中有更好的做法，就是所谓的 **k-平均算法**，可以将两类对象形状聚类，如果我们用它来选择一组 **anchor box**，选择最具有代表性的一组 **anchor box**，可以代表你试图检测的十几个对象类别，但这其实是自动选择 **anchor box** 的高级方法。如果你就人工选择一些形状，合理的考虑到所有对象的形状，你预计会检测的很高很瘦或者很宽很胖的对象，这应该也不难做。

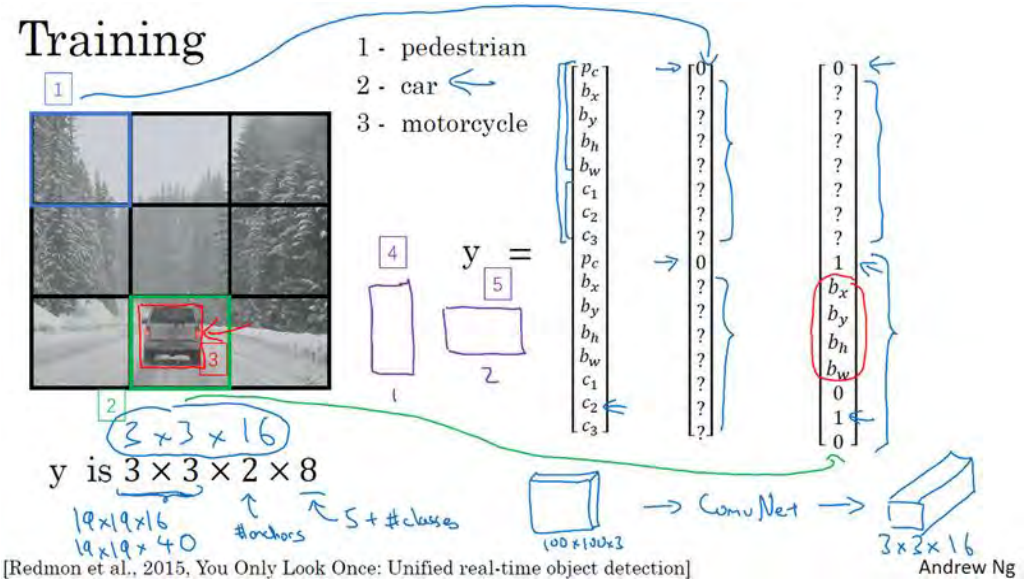
所以这就是 **anchor box**，在下一个视频中，我们把学到的所有东西一起融入到 **YOLO** 算法中。

3.9 YOLO 算法（Putting it together: YOLO algorithm）

你们已经学到对象检测算法的大部分组件了，在这个视频里，我们会把所有组件组装在一起构成 YOLO 对象检测算法。

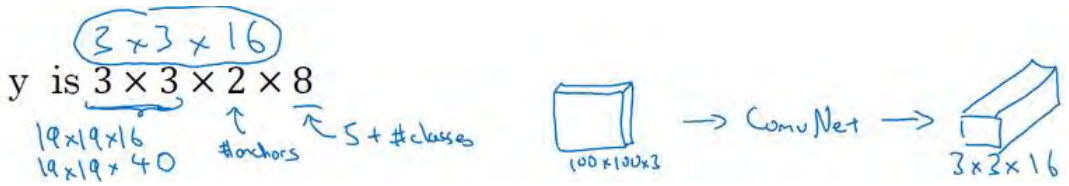


我们先看看如何构造你的训练集，假设你要训练一个算法去检测三种对象，行人、汽车和摩托车，你还需要显式指定完整的背景类别。这里有 3 个类别标签，如果你要用两个 **anchor box**，那么输出 y 就是 3×3×2×8，其中 3×3 表示 3×3 个网格，2 是 **anchor box** 的数量，8 是向量维度，8 实际上先是 5（p_c, b_x, b_y, b_h, b_w）再加上类别的数量（c_1, c_2, c_3）。你可以将它看成是 3×3×2×8，或者 3×3×16。要构造训练集，你需要遍历 9 个格子，然后构成对应的目标向量y。



所以先看看第一个格子（编号 1），里面没什么有价值的东西，行人、车子和摩托车，三个类别都没有出现在左上格子中，所以对应那个格子目标y就是这样的，$y = [0 \quad ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad 0 \quad ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad ?]^T$，第一个 **anchor box** 的 p_c 是 0，因为没什么和第一个 anchor box 有关的，第二个 **anchor box** 的 p_c 也是 0，剩下这些值是 **don't care-s**。

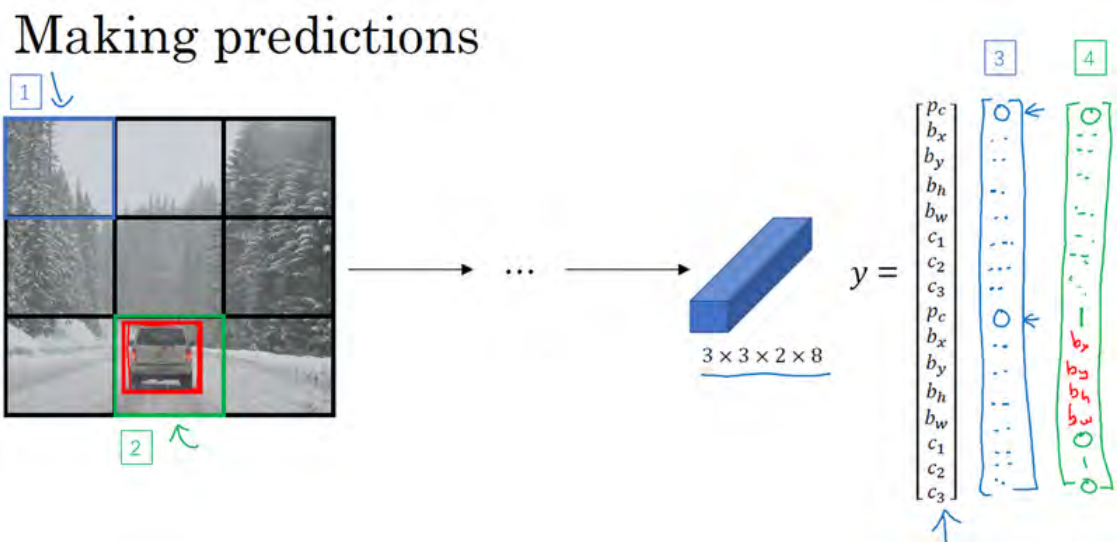
现在网格中大多数格子都是空的，但那里的格子（编号 2）会有这个目标向量y，$y = [0 \quad ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad 1 \quad b_x \quad b_y \quad b_h \quad b_w \quad 0 \quad 1 \quad 0]^T$，所以假设你的训练集中，对于车子有这样一个边界框（编号 3），水平方向更长一点。所以如果这是你的 **anchor box**，这是 **anchor box 1**（编号 4），这是 **anchor box 2**（编号 5），然后红框和 **anchor box 2** 的交并比更高，那么车子就和向量的下半部分相关。要注意，这里和 **anchor box 1** 有关的 p_c 是 0，剩下这些分量都是 **don't care-s**，然后你的第二个 $p_c = 1$，然后你要用这些（b_x, b_y, b_h, b_w）来指定红边界框的位置，然后指定它的正确类别是 $2(c_1 = 0, c_2 = 1, c_3 = 0)$，对吧，这是一辆汽车。



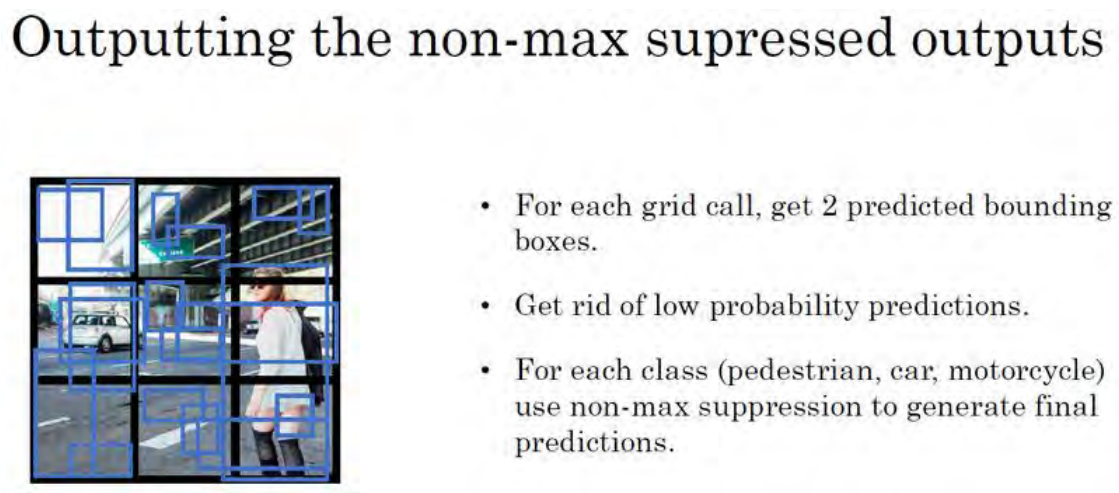
所以你这样遍历 9 个格子，遍历 3×3 网格的所有位置，你会得到这样一个向量，得到一个 16 维向量，所以最终输出尺寸就是 3×3×16。和之前一样，简单起见，我在这里用的是 3×3 网格，实践中用的可能是 19×19×16，或者需要用到更多的 **anchor box**，可能是 19×19×5×8，即 19×19×40，用了 5 个 **anchor box**。这就是训练集，然后你训练一个卷积网络，输入是图

片，可能是 100×100×3，然后你的卷积网络最后输出尺寸是，在我们例子中是 3×3×16 或者 3×3×2×8。

接下来我们看看你的算法是怎样做出预测的，输入图像，你的神经网络的输出尺寸是这个 3××3×2×8，对于 9 个格子，每个都有对应的向量。对于左上的格子（编号 1），那里没有任何对象，那么我们希望你的神经网络在那里（第一个p_c）输出的是 0，这里（第二个p_c）是 0，然后我们输出一些值，你的神经网络不能输出问号，不能输出 **don't care-s**，剩下的我输入一些数字，但这些数字基本上会被忽略，因为神经网络告诉你，那里没有任何东西，所以输出是不是对应一个类别的边界框无关紧要，所以基本上是一组数字，多多少少都是噪音（输出 y 如编号 3 所示）。



和这里的边界框不大一样，希望y的值，那个左下格子（编号 2）的输出y（编号 4 所示），形式是，对于边界框 1 来说（p_c）是 0，然后就是一组数字，就是噪音（**anchor box 1** 对应行人，此格子中无行人，$p_c = 0, b_x = ?, b_y = ?, b_h = ?, b_w = ?, c_1 = ? c_2 = ?, c_3 = ?$）。希望你的算法能输出一些数字，可以对车子指定一个相当准确的边界框（**anchor box 2** 对应汽车，此格子中有车，$p_c = 1, b_x, b_y, b_h, b_w, c_1 = 0, c_2 = 1, c_3 = 0$），这就是神经网络做出预测的过程。



最后你要运行一下这个非极大值抑制，为了让内容更有趣一些，我们看看一张新的测试图像，这就是运行非极大值抑制的过程。如果你使用两个 **anchor box**，那么对于 9 个格子中任何一个都会有两个预测的边界框，其中一个的概率p_c很低。但 9 个格子中，每个都有两个预测的边界框，比如说我们得到的边界框是是这样的，注意有一些边界框可以超出所在格子的高度和宽度（编号 1 所示）。接下来你抛弃概率很低的预测，去掉这些连神经网络都说，这里很可能什么都没有，所以你需要抛弃这些（编号 2 所示）。

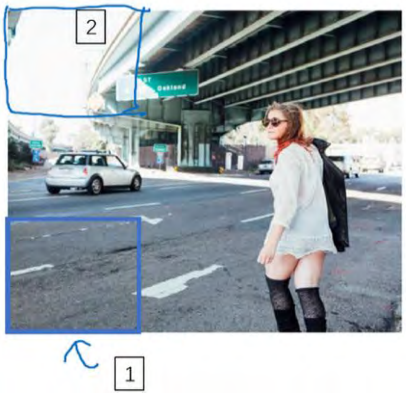
最后，如果你有三个对象检测类别，你希望检测行人，汽车和摩托车，那么你要做的是，对于每个类别单独运行非极大值抑制，处理预测结果所属类别的边界框，用非极大值抑制来处理行人类别，用非极大值抑制处理车子类别，然后对摩托车类别进行非极大值抑制，运行三次来得到最终的预测结果。所以算法的输出最好能够检测出图像里所有的车子，还有所有的行人（编号 3 所示）。

这就是 **YOLO** 对象检测算法，这实际上是最有效的对象检测算法之一，包含了整个计算机视觉对象检测领域文献中很多最精妙的思路。你可以在本周的编程作业中尝试现实这个算法，所以我希望你喜欢本周的编程练习，这里还有一个可选的视频，你们可以看，也可以不看，总之，我们下周见。

3.10 候选区域（选修）（Region proposals (Optional)）

如果你们阅读一下对象检测的文献，可能会看到一组概念，所谓的候选区域，这在计算机视觉领域是非常有影响力的概念。我把这个视频定为可选视频是因为我用到候选区域这一系列算法的频率没有那么高，但当然了，这些工作是很有影响力的，你们在工作中也可能会碰到，我们来看看。

你们还记得滑动窗法吧，你使用训练过的分类器，在这些窗口中全部运行一遍，然后运行一个检测器，看看里面是否有车辆，行人和摩托车。现在你也可以运行一下卷积算法，这个算法的其中一个缺点是，它在显然没有任何对象的区域浪费时间，对吧。



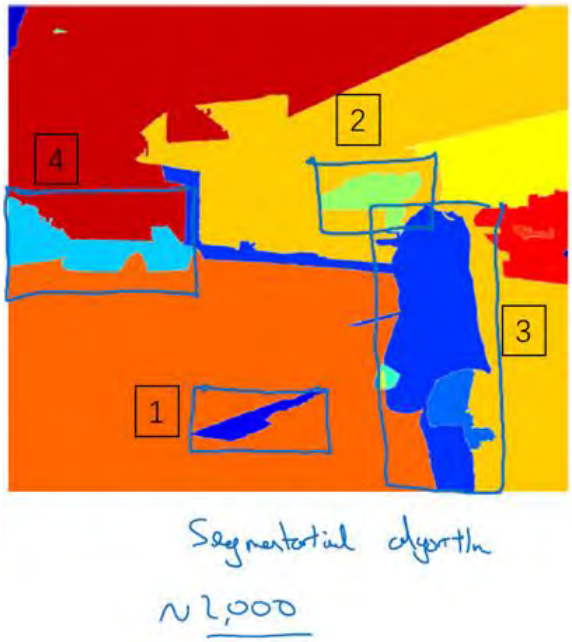
所以这里这个矩形区域（编号 1）基本是空的，显然没有什么需要分类的东西。也许算法会在这个矩形上（编号 2）运行，而你知道上面没有什么有趣的东西。

[Girshik et. al, 2013, Rich feature hierarchies for accurate object detection and semantic segmentation]

所以 **Ross Girshick**，**Jeff Donahue**，**Trevor Darrell**，**Jitendra Malik**，在本幻灯片底部引用到的论文中提出一种叫做 R-CNN 的算法，意思是带区域的卷积网络，或者说带区域的 **CNN**。这个算法尝试选出一些区域，在这些区域上运行卷积网络分类器是有意义的，所以这里不再针对每个滑动窗运行检测算法，而是只选择一些窗口，在少数窗口上运行卷积网络分类器。

选出候选区域的方法是运行图像分割算法，分割的结果是下边的图像，为了找出可能存在对象的区域。比如说，分割算法在这里得到一个色块，所以你可能会选择这样的边界框（编号 1），然后在这个色块上运行分类器，就像这个绿色的东西（编号 2），在这里找到一个

色块，接下来我们还会在那个矩形上（编号 2）运行一次分类器，看看有没有东西。在这种情况下，如果在蓝色色块上（编号 3）运行分类器，希望你能检测出一个行人，如果你在青色色块(编号 4)上运行算法，也许你可以发现一辆车，我也不确定。



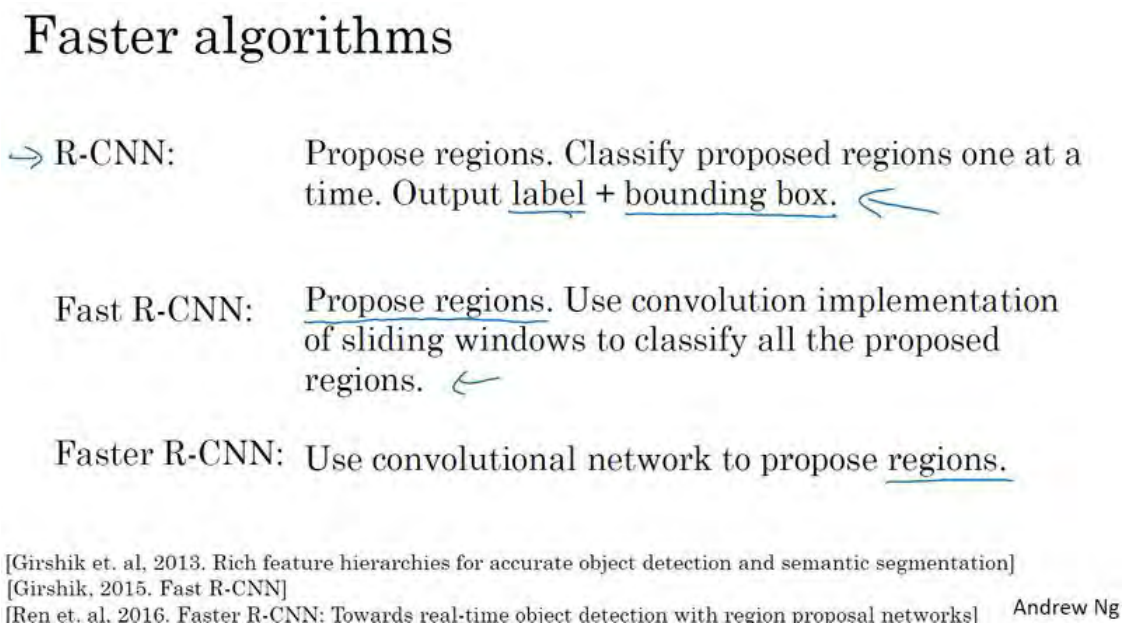
所以这个细节就是所谓的分割算法，你先找出可能 2000 多个色块，然后在这 2000 个色块上放置边界框，然后在这 2000 个色块上运行分类器，这样需要处理的位置可能要少的多，可以减少卷积网络分类器运行时间，比在图像所有位置运行一遍分类器要快。特别是这种情况，现在不仅是在方形区域（编号 5）中运行卷积网络，我们还会在高高瘦瘦（编号 6）的区域运行，尝试检测出行人，然后我们在很宽很胖的区域（编号 7）运行，尝试检测出车辆，同时在各种尺度运行分类器。

这就是 **R-CNN** 或者**区域 CNN** 的特色概念，现在看来 **R-CNN** 算法还是很慢的。所以有一系列的研究工作去改进这个算法，所以基本的 **R-CNN** 算法是使用某种算法求出候选区域，然后对每个候选区域运行一下分类器，每个区域会输出一个标签，有没有车子？有没有行人？有没有摩托车？并输出一个边界框，这样你就能在确实存在对象的区域得到一个精确的边界框。

澄清一下，**R-CNN** 算法不会直接信任输入的边界框，它也会输出一个边界框b_x，b_y，b_h和b_w，这样得到的边界框比较精确，比单纯使用图像分割算法给出的色块边界要好，所以它可以得到相当精确的边界框。

现在 **R-CNN** 算法的一个缺点是太慢了，所以这些年来有一些对 **R-CNN** 算法的改进工作，**Ross Girshik** 提出了快速的 **R-CNN** 算法，它基本上是 **R-CNN** 算法，不过用卷积实现了滑动窗

法。最初的算法是逐一对区域分类的，所以快速 **R-CNN** 用的是滑动窗法的一个卷积实现，这和你在本周第四个视频（3.4 卷积的滑动窗口实现）中看到的大致相似，这显著提升了 **R-CNN** 的速度。



事实证明，快速 **R-CNN** 算法的其中一个问题是得到候选区域的聚类步骤仍然非常缓慢，所以另一个研究组，**任少卿**（**Shaoqing Ren**）、**何凯明**（**Kaiming He**）、**Ross Girshick** 和**孙剑**（**Jiangxi Sun**）提出了更快的 **R-CNN** 算法，使用的是卷积神经网络，而不是更传统的分割算法来获得候选区域色块，结果比快速 **R-CNN** 算法快得多。不过我认为大多数更快 **R-CNN** 的算法实现还是比 **YOLO** 算法慢很多。

候选区域的概念在计算机视觉领域的影响力相当大，所以我希望你们能了解一下这些算法，因为你可以看到还有人在用这些概念。对我个人来说，这是我的个人看法而不是整个计算机视觉研究界的看法，我觉得候选区域是一个有趣的想法，但这个方法需要两步，首先得到候选区域，然后再分类，相比之下，能够一步做完，类似于 **YOLO** 或者你只看一次（**You only look once**）这个算法，在我看来，是长远而言更有希望的方向。但这是我的个人看法，而不是整个计算机视觉研究界的看法，所以你们最好批判接受。但我想这个 **R-CNN** 概念，你可能会想到，或者碰到其他人在用，所以这也是值得了解的，这样你可以更好地理解别人的算法。

现在我们就讲完这周对象检测的材料了，我希望你们喜欢本周的编程练习，我们下周见。

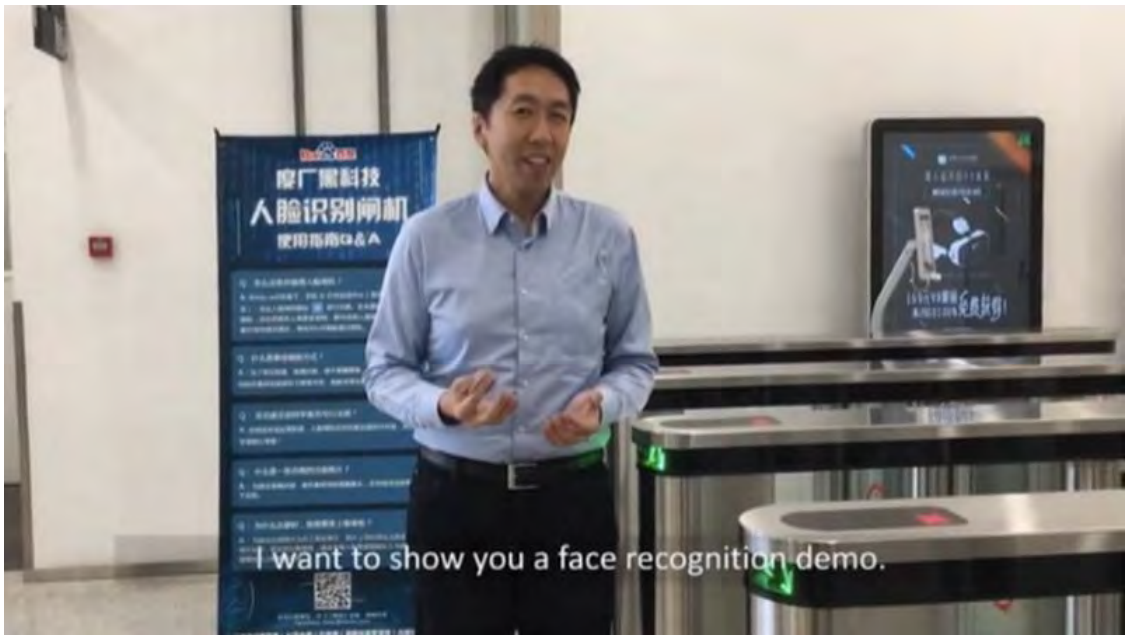
第四周 特殊应用：人脸识别和神经风格转换（Special applications: Face recognition &Neural style transfer）

4.1 什么是人脸识别？（What is face recognition?）

欢迎来到第四周，即这门课卷积神经网络课程的最后一周。到目前为止，你学了很多卷积神经网络的知识。我这周准备向你展示一些重要的卷积神经网络的特殊应用，我们将从人脸识别开始，之后讲神经风格迁移，你将有机会在编程作业中实现这部分内容，创造自己的艺术作品。

让我们先从人脸识别开始，我这里有一个有意思的演示。我在领导百度 **AI** 团队的时候，其中一个小组由林元庆带领的，做过一个人脸识别系统，这个系统非常棒，让我们来看一下。

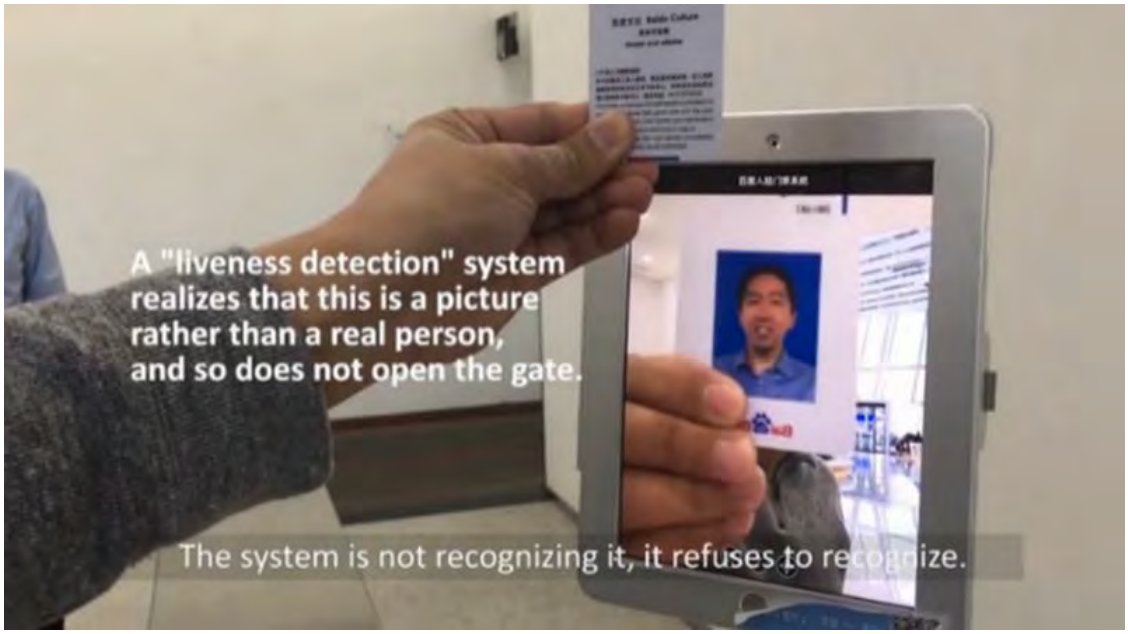
（以下内容为演示视频内容）



视频开始：

我想演示一个人脸识别系统，我现在在百度的中国总部，很多公司要求进入公司的时候要刷工卡，但是在这里我们并不需要它，使用人脸识别，看看我能做什么。当我走近的时候，它会识别我的脸，然后说欢迎我（**Andrew NG**），不需要工卡，我就能通过了。

让我们看看另一种情况，在旁边的是林元庆，**IDL**（百度深度学习实验室）的主管，他领导开发了这个人脸识别系统，我把我的工卡给他，上面有我的头像，他会试着用我的头像照片，而不是真人来通过。



（林元庆语：我将尝试用 **Andrew** 的工卡骗过机器，看看发生什么，系统不会识别，系统拒绝识别。现在我要用我自己的脸，（系统语音：“欢迎您”）（林元庆顺利通过））

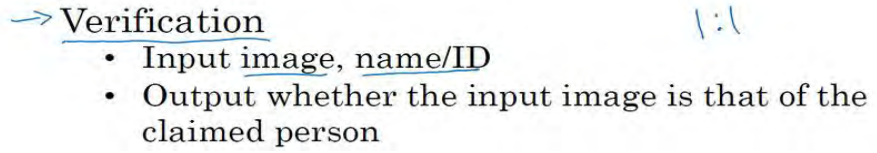
类似于这样的人脸识别系统在中国发展很快，我希望这个技术也可以在其他的国家使用。

#视频结束

挺厉害的吧，你刚看到的这个视频展示了人脸识别和活体检测，后一项技术确认你是一个活人。事实上，活体检测可以使用监督学习来实现，去预测是不是一个真人，这个方面我就不多说了。我主要想讲的是，如何构造这个系统中的人脸识别这一部分。

首先，让我们了解一下人脸识别的一些术语。

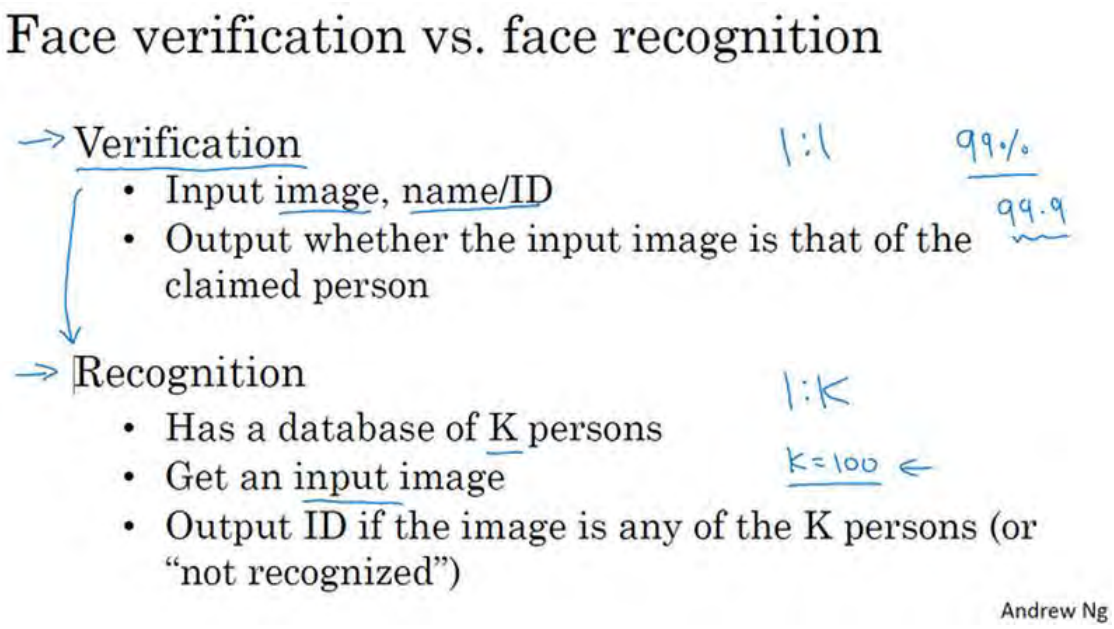
在人脸识别的相关文献中，人们经常提到人脸验证（**face verification**）和人脸识别（**face recognition**）。



这是人脸验证问题，如果你有一张输入图片，以及某人的 **ID** 或者是名字，这个系统要做的是，验证输入图片是否是这个人。有时候也被称作 1 对 1 问题，只需要弄明白这个人是否和他声称的身份相符。

而人脸识别问题比人脸验证问题难很多（整理者注：1 对多问题（$1:K$）），为什么呢？假设你有一个验证系统，准确率是 99%，还可以。但是现在，假设在识别系统中，$K = 100$，如果你把这个验证系统应用在 100 个人身上，人脸识别上，你犯错的机会就是 100 倍了。如

果每个人犯错的概率是 1%，如果你有一个上百人的数据库，如果你想得到一个可接受的识别误差，你要构造一个验证系统，其准确率为 99.9%或者更高，然后才可以在 100 人的数据库上运行，而保证有很大几率不出错。事实上，如果我们有一个 100 人的数据库，正确率可能需要远大于 99%，才能得到很好的效果。

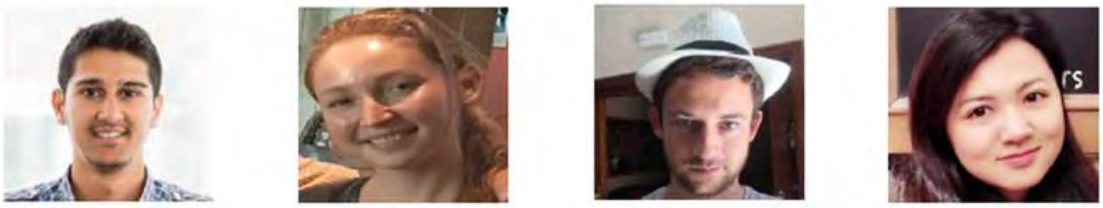


在之后的几个视频中，我们主要讲构造一个人脸验证，作为基本模块，如果准确率够高，你就可以把它用在识别系统上。

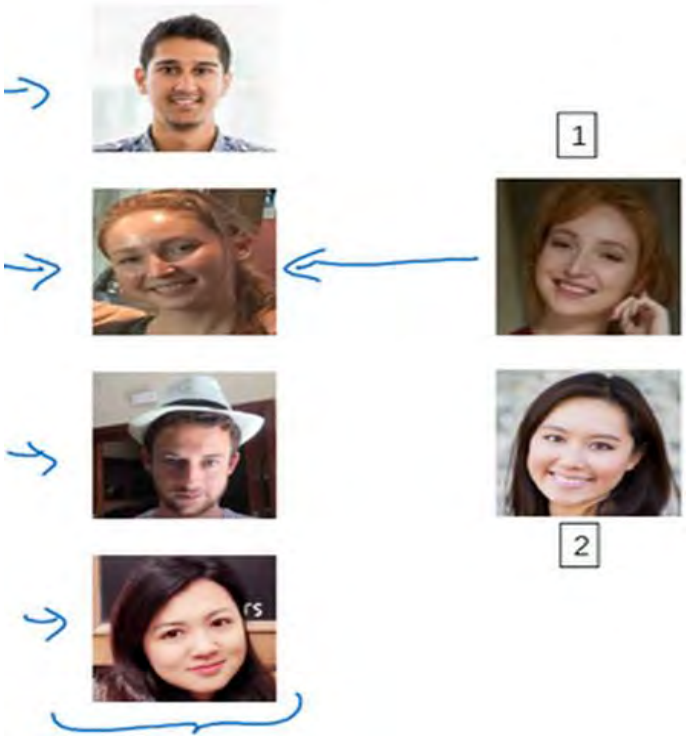
下一个视频中，我们将开始讨论如何构造人脸验证系统，人脸验证之所以难，原因之一在于要解决“一次学”（**one-shot learning problem**）问题。让我们看下一个视频，什么是一次学习问题。

4.2 One-Shot 学习（One-shot learning）

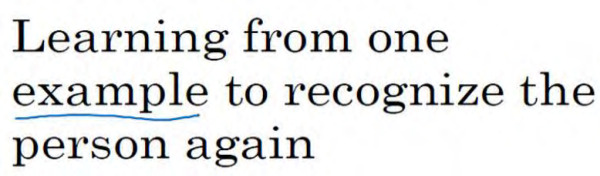
人脸识别所面临的一个挑战就是你需要解决一次学习问题，这意味着在大多数人脸识别应用中，你需要通过单单一张图片或者单单一个人脸样例就能去识别这个人。而历史上，当深度学习只有一个训练样例时，它的表现并不好，让我们看一个直观的例子，并讨论如何去解决这个问题。



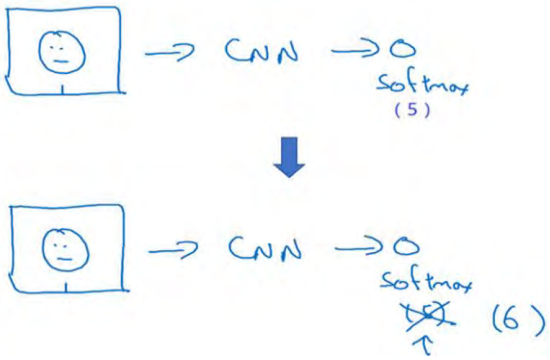
假设你的数据库里有 4 张你们公司的员工照片，实际上他们确实是我们 **deeplearning.ai** 的员工，分别是 **Kian**，**Danielle**，**Younes** 和 **Tian**。现在假设有个人（编号 1 所示）来到办公室，并且她想通过带有人脸识别系统的栅门，现在系统需要做的就是，仅仅通过一张已有的 Danielle 照片，来识别前面这个人确实是她。相反，如果机器看到一个不在数据库里的人（编号 2 所示），机器应该能分辨出她不是数据库中四个人之一。



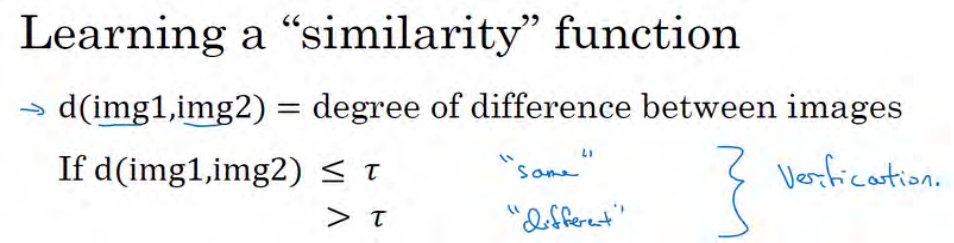
所以在一次学习问题中，只能通过一个样本进行学习，以能够认出同一个人。大多数人脸识别系统都需要解决这个问题，因为在你的数据库中每个雇员或者组员可能都只有一张照片。



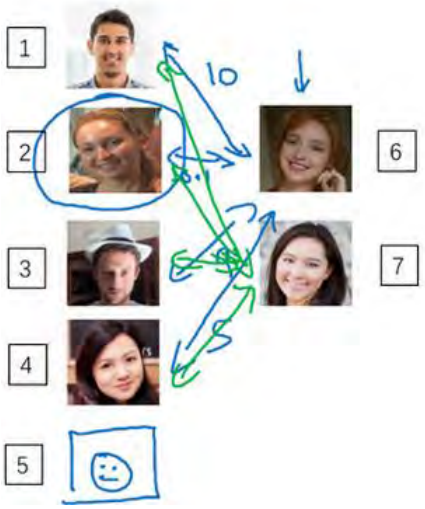
有一种办法是，将人的照片放进卷积神经网络中，使用 **softmax** 单元来输出 4 种，或者说 5 种标签，分别对应这 4 个人，或者 4 个都不是，所以 **softmax** 里我们会有 5 种输出。但实际上这样效果并不好，因为如此小的训练集不足以去训练一个稳健的神经网络。

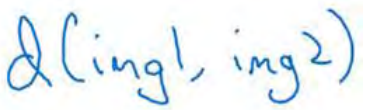


而且，假如有新人加入你的团队，你现在将会有 5 个组员需要识别，所以输出就变成了 6 种，这时你要重新训练你的神经网络吗？这听起来实在不像一个好办法。



所以要让人脸识别能够做到一次学习，为了能有更好的效果，你现在要做的应该是学习 Similarity 函数。详细地说，你想要神经网络学习这样一个用d表示地函数，$d(img1, img2) = degree\ of\ difference\ between\ images$，它以两张图片作为输入，然后输出这两张图片的差异值。如果你放进同一个人的两张照片，你希望它能输出一个很小的值，如果放进两个长相差别很大的人的照片，它就输出一个很大的值。所以在识别过程中，如果这两张图片的差异值小于某个阈值τ，它是一个超参数，那么这时就能预测这两张图片是同一个人，如果差异值大于 τ，就能预测这是不同的两个人，这就是解决人脸验证问题的一个可行办法。





要将它应用于识别任务，你要做的是拿这张新图片（编号 6），然后用 d 函数去比较这两张图片（编号 1 和编号 6），这样可能会输出一个非常大的数字，在该例中，比如说这个数字是 10。之后你再让它和数据库中第二张图（编号 2）片比较，因为这两张照片是同一个人，所以我们希望会输出一个很小的数。然后你再用它与数据库中的其他图片（编号 3、4）进行比较，通过这样的计算，最终你能够知道，这个人确实是 **Danielle**。

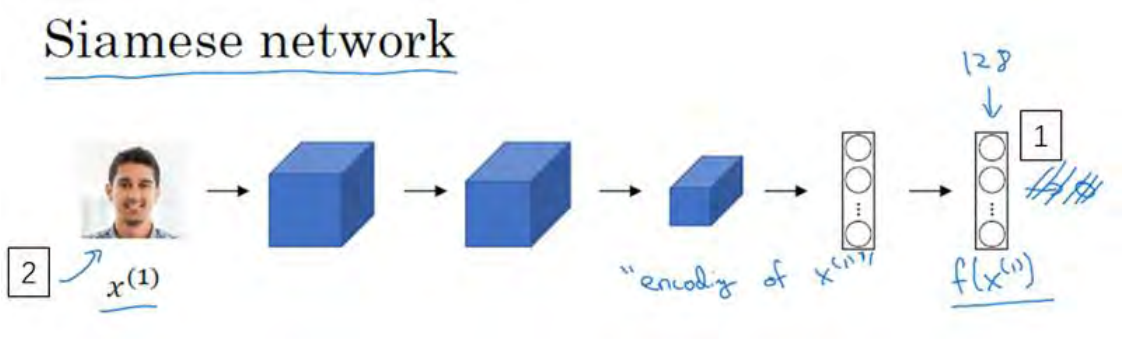
对应的，如果某个人（编号 7）不在你的数据库中，你通过函数 d 将他们的照片两两进行比较，最后我们希望d会对所有的比较都输出一个很大的值，这就证明这个人并不是数据库中 4 个人的其中一个。

要注意在这过程中你是如何解决一次学习问题的，只要你能学习这个函数d，通过输入一对图片，它将会告诉你这两张图片是否是同一个人。如果之后有新人加入了你的团队（编号 5），你只需将他的照片加入你的数据库，系统依然能照常工作。

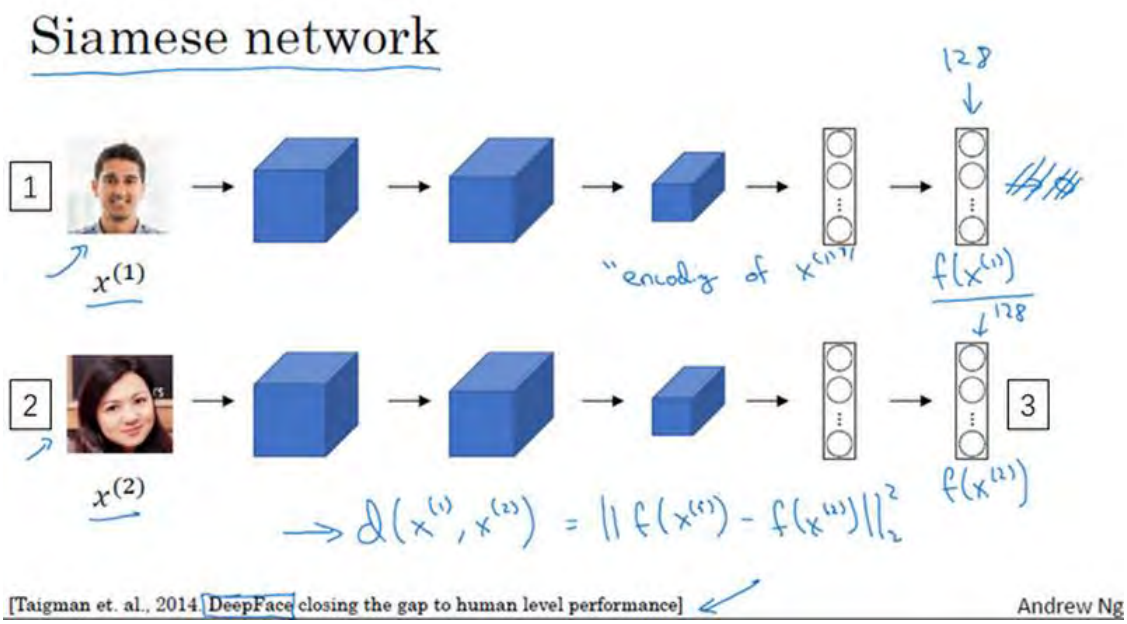
现在你已经知道函数 d 是如何工作的，通过输入两张照片，它将让你能够解决一次学习问题。那么，下节视频中，我们将会学习如何训练你的神经网络学会这个函数 d。

4.3 Siamese 网络（Siamese network）

上个视频中你学到的函数 d 的作用就是输入两张人脸，然后告诉你它们的相似度。实现这个功能的一个方式就是用 **Siamese** 网络，我们看一下。



你经常看到这样的卷积网络，输入图片$x^{(1)}$，然后通过一系列卷积，池化和全连接层，最终得到这样的特征向量（编号 1）。有时这个会被送进 **softmax** 单元来做分类，但在这个视频里我们不会这么做。我们关注的重点是这个向量（编号 1），加如它有 128 个数，它是由网络深层的全连接层计算出来的，我要给这 128 个数命个名字，把它叫做$f(x^{(1)})$。你可以把$f(x^{(1)})$看作是输入图像$x^{(1)}$的编码，取这个输入图像（编号 2），在这里是 **Kian** 的图片，然后表示成 128 维的向量。

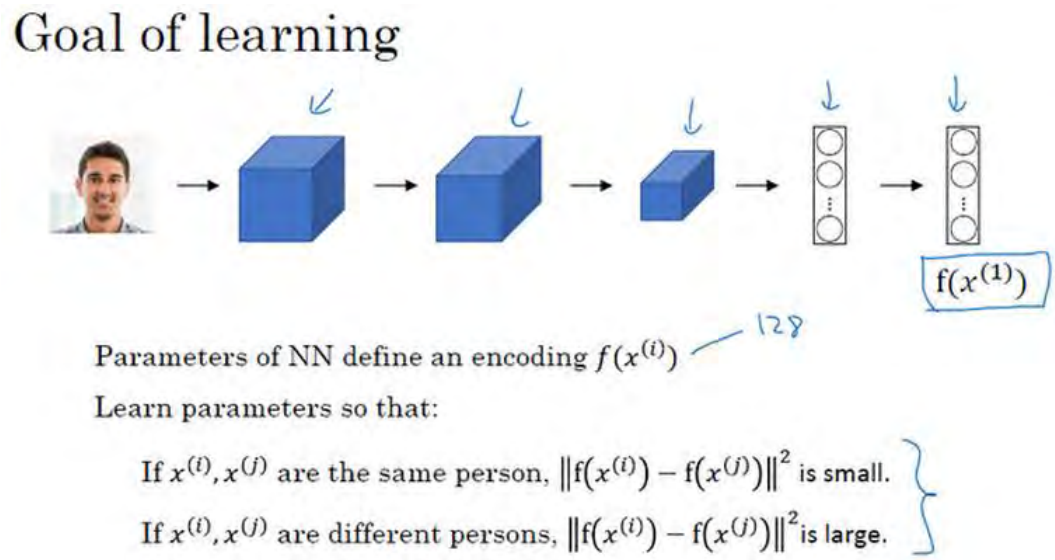


建立一个人脸识别系统的方法就是，如果你要比较两个图片的话，例如这里的第一张（编号 1）和第二张图片（编号 2），你要做的就是把第二张图片喂给有同样参数的同样的神经网络，然后得到一个不同的 128 维的向量（编号 3），这个向量代表或者编码第二个图片，我要把第二张图片的编码叫做$f(x^{(2)})$。这里我用$x^{(1)}$和$x^{(2)}$仅仅代表两个输入图片，他们没

必要非是第一个和第二个训练样本，可以是任意两个图片。

最后如果你相信这些编码很好地代表了这两个图片，你要做的就是定义d，将$x^{(1)}$和$x^{(2)}$的距离定义为这两幅图片的编码之差的范数，$d(x^{(1)},x^{(2)}) = ||f(x^{(1)}) - f(x^{(2)})||_2^{(2)}$。

对于两个不同的输入，运行相同的卷积神经网络，然后比较它们，这一般叫做 **Siamese** 网络架构。这里提到的很多观点，都来自于 **Yaniv Taigman**，**Ming Yang**，**Marc' Aurelio Ranzato**，**Lior Wolf** 的这篇论文，他们开发的系统叫做 **DeepFace**。



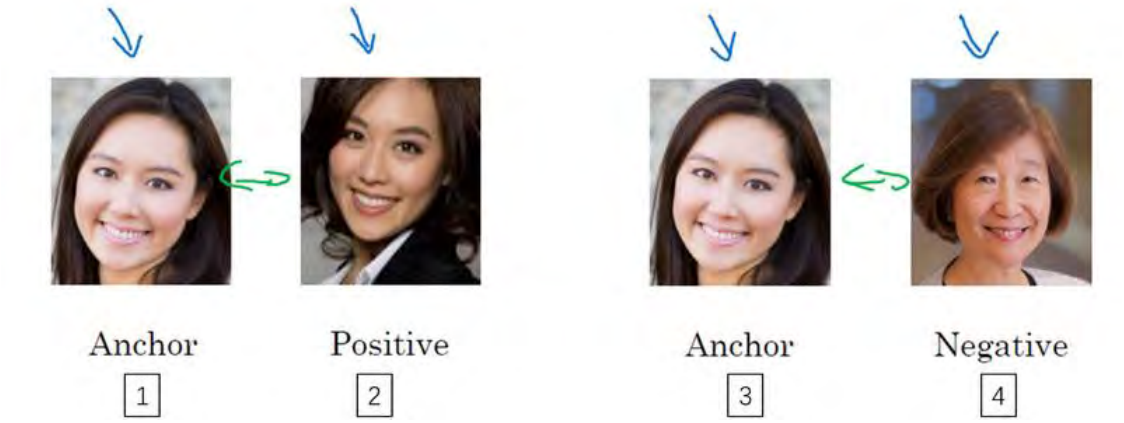
怎么训练这个 **Siamese** 神经网络呢？不要忘了这两个网络有相同的参数，所以你实际要做的就是训练一个网络，它计算得到的编码可以用于函数d，它可以告诉你两张图片是否是同一个人。更准确地说，神经网络的参数定义了一个编码函数$f(x^{(i)})$，如果给定输入图像$x^{(i)}$，这个网络会输出$x^{(i)}$的 128 维的编码。你要做的就是学习参数，使得如果两个图片$x^{(i)}$和$x^{(j)}$是同一个人，那么你得到的两个编码的距离就小。前面几个幻灯片我都用的是$x^{(1)}$和$x^{(2)}$，其实训练集里任意一对$x^{(i)}$和$x^{(j)}$都可以。相反，如果$x^{(i)}$和$x^{(j)}$是不同的人，那么你会想让它们之间的编码距离大一点。

如果你改变这个网络所有层的参数，你会得到不同的编码结果，你要做的就是用反向传播来改变这些所有的参数，以确保满足这些条件。

你已经了解了 **Siamese** 网络架构，并且知道你想要网络输出什么，即什么是好的编码。但是如何定义实际的目标函数，能够让你的神经网络学习并做到我们刚才讨论的内容呢？在下一个视频里，我们会看到如何用三元组损失函数达到这个目的。

4.4 Triplet 损失（Triplet 损失）

要想通过学习神经网络的参数来得到优质的人脸图片编码，方法之一就是定义三元组损失函数然后应用梯度下降。



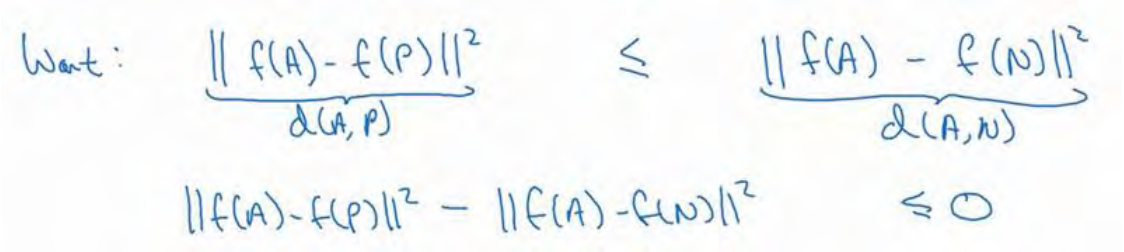
我们看下这是什么意思，为了应用三元组损失函数，你需要比较成对的图像，比如这个图片，为了学习网络的参数，你需要同时看几幅图片，比如这对图片（编号 1 和编号 2），你想要它们的编码相似，因为这是同一个人。然而假如是这对图片（编号 3 和编号 4），你会想要它们的编码差异大一些，因为这是不同的人。

用三元组损失的术语来说，你要做的通常是看一个 **anchor** 图片，你想让 **anchor** 图片和 **positive** 图片（**positive** 意味着是同一个人）的距离很接近。然而，当 **anchor** 图片与 **negative** 图片（**negative** 意味着是非同一个人）对比时，你会想让他们的距离离得更远一点。

这就是为什么叫做三元组损失，它代表你通常会同时看三张图片，你需要看 **anchor** 图片、**postive** 图片，还有 **negative** 图片，我要把 **anchor** 图片、**positive** 图片和 **negative** 图片简写成A、P、N。

把这些写成公式的话，你想要的是网络的参数或者编码能够满足以下特性，也就是说你想要$||f(A)-f(P)||^2$，你希望这个数值很小，准确地说，你想让它小于等$f(A)$和$f(N)$之间的

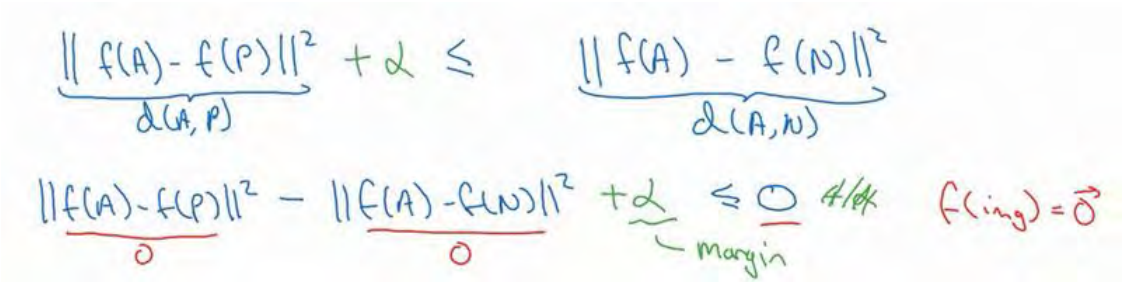
距离，或者说是它们的范数的平方（即：$||f(A)-f(P)||^2 \leq ||f(A)-f(N)||^2$）。（$||f(A)-f(P)||^2$）当然这就是$d(A,P)$，（$||f(A)-f(N)||^2$）这是$d(A,N)$，你可以把$d$ 看作是距离(**distance**)函数，这也是为什么我们把它命名为d。



现在如果我把方程右边项移到左边，最终就得到：

$$||f(A)-f(P)||^2 \leq ||f(A)-f(N)||^2$$

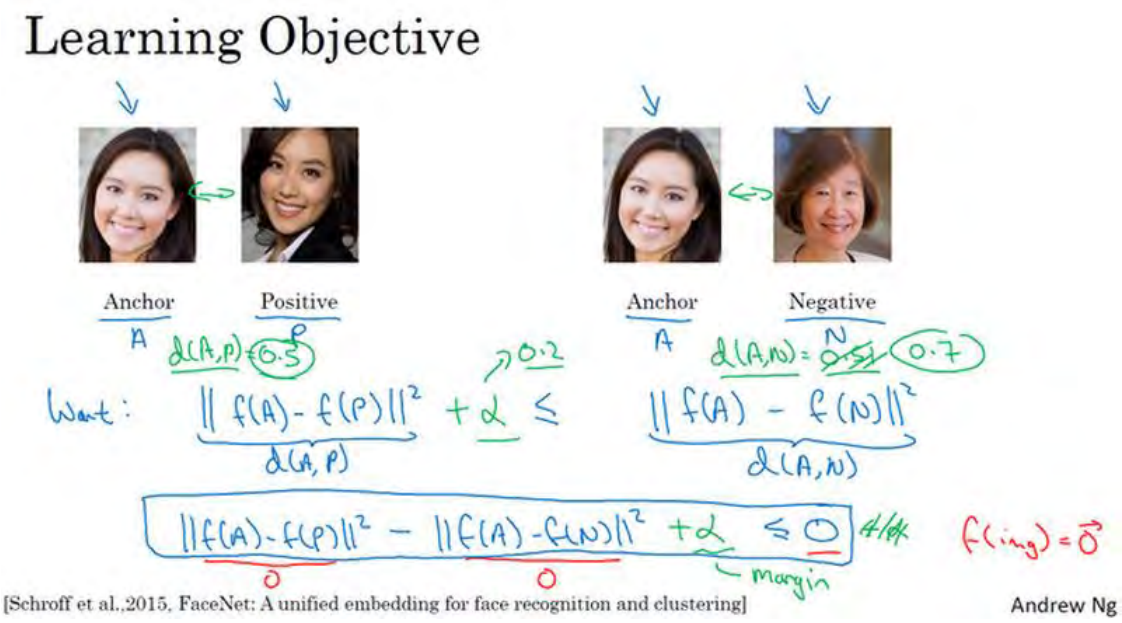
现在我要对这个表达式做一些小的改变，有一种情况满足这个表达式，但是没有用处，就是把所有的东西都学成 0，如果 f 总是输出 0，即 0-0≤0，这就是 0 减去 0 还等于 0，如果所有图像的 f 都是一个零向量，那么总能满足这个方程。所以为了确保网络对于所有的编码不会总是输出 0，也为了确保它不会把所有的编码都设成互相相等的。另一种方法能让网络得到这种没用的输出，就是如果每个图片的编码和其他图片一样，这种情况，你还是得到 0-0。



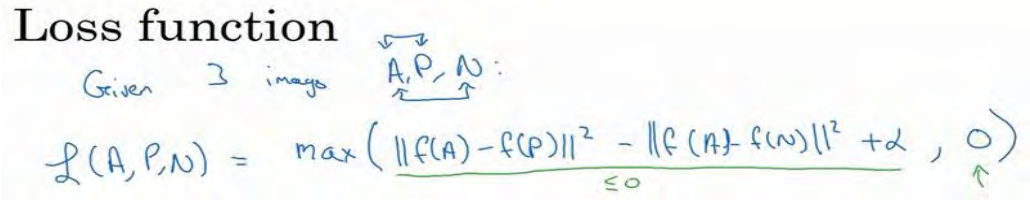
为了阻止网络出现这种情况，我们需要修改这个目标，也就是，这个不能是刚好小于等于 0，应该是比 0 还要小，所以这个应该小于一个-α 值（即$||f(A)-f(P)||^2 - ||f(A)-f(N)||^2 \leq -a$），这里的$a$是另一个超参数，这个就可以阻止网络输出无用的结果。按照惯例，我们习惯写$+a$（即$||f(A)-f(P)||^2 - ||f(A)-f(N)||^2 + a \leq 0$），而不是把$-a$写在后面，它也叫做间隔(**margin**)，这个术语你会很熟悉，如果你看过关于支持向量机 (**SVM**) 的文献，没看过也不用担心。我们可以把上面这个方程（$||f(A)-f(P)||^2||f(A)-f(N)||^2$）也修改一下，加上这个间隔参数。

举个例子，假如间隔设置成 0.2，如果在这个例子中，$d(A,P)=0.5$，如果 **anchor** 和 **negative** 图片的d，即$d(A,N)$只大一点，比如说 0.51，条件就不能满足。虽然 0.51 也是大于

0.5 的，但还是不够好，我们想要$d(A,N)$比$d(A,P)$大很多，你会想让这个值（$d(A,N)$）至少是 0.7 或者更高，或者为了使这个间隔，或者间距至少达到 0.2，你可以把这项调大或者这个调小，这样这个间隔a，超参数a 至少是 0.2，在$d(A,P)$和$d(A,N)$之间至少相差 0.2，这就是间隔参数a的作用。它拉大了 **anchor** 和 **positive** 图片对和 **anchor** 与 **negative** 图片对之间的差距。取下面的这个方框圈起来的方程式，在下个幻灯片里，我们会更公式化表示，然后定义三元组损失函数。



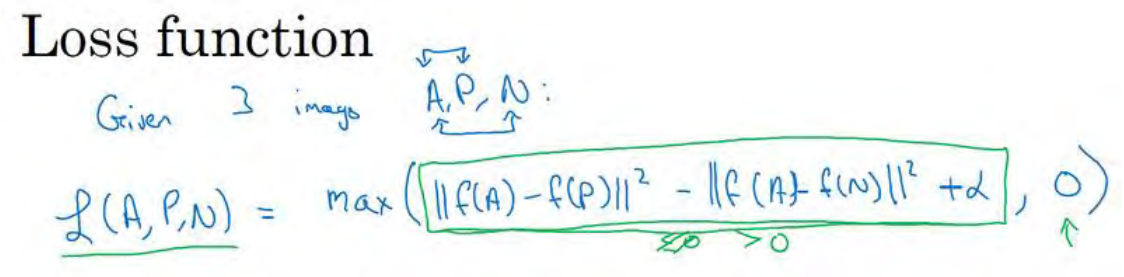
三元组损失函数的定义基于三张图片，假如三张图片A、P、N，即 **anchor** 样本、**positive** 样本和 **negative** 样本，其中 **positive** 图片和 **anchor** 图片是同一个人，但是 **negative** 图片和 **anchor** 不是同一个人。



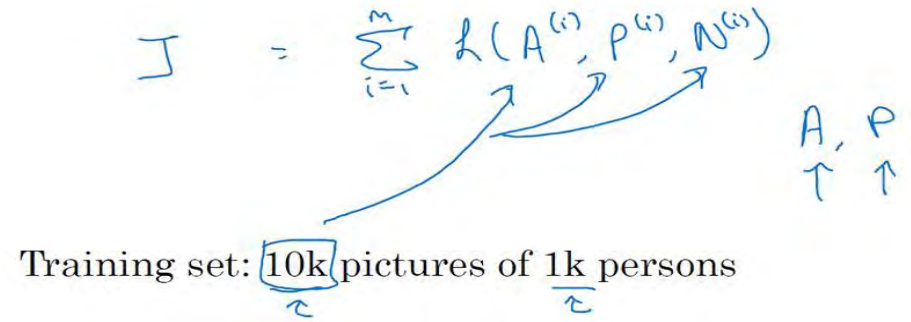
接下来我们定义损失函数，这个例子的损失函数，它的定义基于三元图片组，我先从前一张幻灯片复制过来一些式子，就是$||f(A)-f(P)||^2-||f(A)-f(N)||^2+a\leq 0$。所以为了定义这个损失函数，我们取这个和 0 的最大值：

$$L(A,P,N)=max(||f(A)-f(P)||^2-||f(A)-f(N)||^2+a,0)$$

这个 max 函数的作用就是，只要这个$||f(A)-f(P)||^2-||f(A)-f(N)||^2+a\leq 0$，那么损失函数就是 0。只要你能使画绿色下划线部分小于等于 0，只要你能达到这个目标，那么这个例子的损失就是 0。

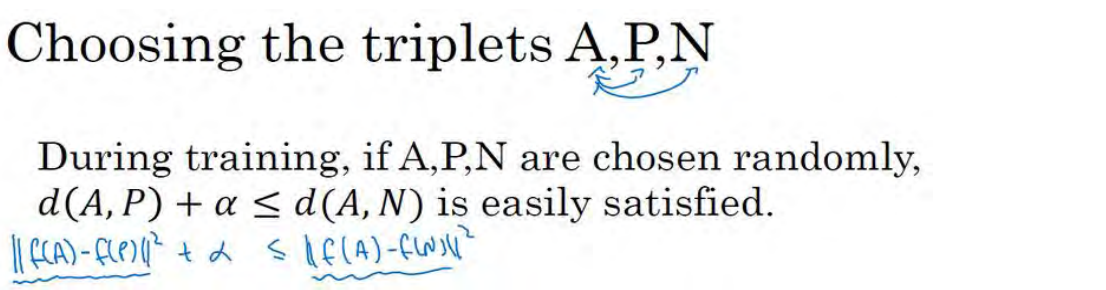


另一方面如果这个$||f(A)-f(P)||^2 - ||f(A)-f(N)||^2 + a \leq 0$，然后你取它们的最大值，最终你会得到绿色下划线部分（即$||f(A)-f(P)||^2 - ||f(A)-f(N)||^2 + a$）是最大值，这样你会得到一个正的损失值。通过最小化这个损失函数达到的效果就是使这部分$||f(A)-f(P)||^2 - ||f(A)-f(N)||^2 + a$成为 0，或者小于等于 0。只要这个损失函数小于等于 0，网络不会关心它负值有多大。

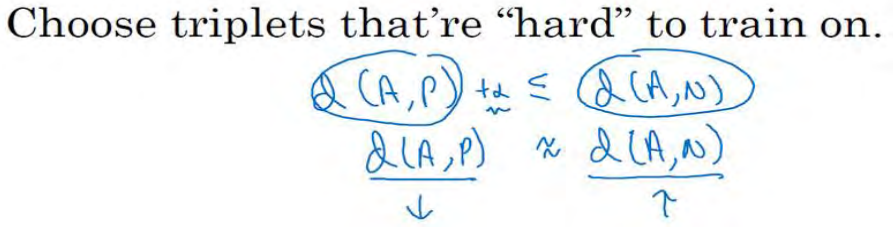


这是一个三元组定义的损失，整个网络的代价函数应该是训练集中这些单个三元组损失的总和。假如你有一个 10000 个图片的训练集，里面是 1000 个不同的人的照片，你要做的就是取这 10000 个图片，然后生成这样的三元组，然后训练你的学习算法，对这种代价函数用梯度下降，这个代价函数就是定义在你数据集里的这样的三元组图片上。

注意，为了定义三元组的数据集你需要成对的A和P，即同一个人的成对的图片，为了训练你的系统你确实需要一个数据集，里面有同一个人的多个照片。这是为什么在这个例子中，我说假设你有 1000 个不同的人的 10000 张照片，也许是这 1000 个人平均每个人 10 张照片，组成了你整个数据集。如果你只有每个人一张照片，那么根本没法训练这个系统。当然，训练完这个系统之后，你可以应用到你的一次学习问题上，对于你的人脸识别系统，可能你只有想要识别的某个人的一张照片。但对于训练集，你需要确保有同一个人的多个图片，至少是你训练集里的一部分人，这样就有成对的 **anchor** 和 **positive** 图片了。



现在我们来看，你如何选择这些三元组来形成训练集。一个问题是如果你从训练集中，随机地选择A、P和N，遵守A和P是同一个人，而A和N是不同的人这一原则。有个问题就是，如果随机的选择它们，那么这个约束条件（$d(A,P)+a\leq d(A,N)$）很容易达到，因为随机选择的图片，A和N比A和P差别很大的概率很大。我希望你还记得这个符号 $d(A,P)$就是前几个幻灯片里写的$||f(A)-f(P)||^2$，$d(A,N)$就是$||f(A)-f(N)||^2$，$d(A,P)+a\leq d(A,N)$即$||f(A)-f(P)||^2+a\leq||f(A)-f(N)||^2$。但是如果$A$和$N$是随机选择的不同的人，有很大的可能性$||f(A)-f(N)||^2$会比左边这项$||f(A)-f(P)||^2$大，而且差距远大于$a$，这样网络并不能从中学到什么。



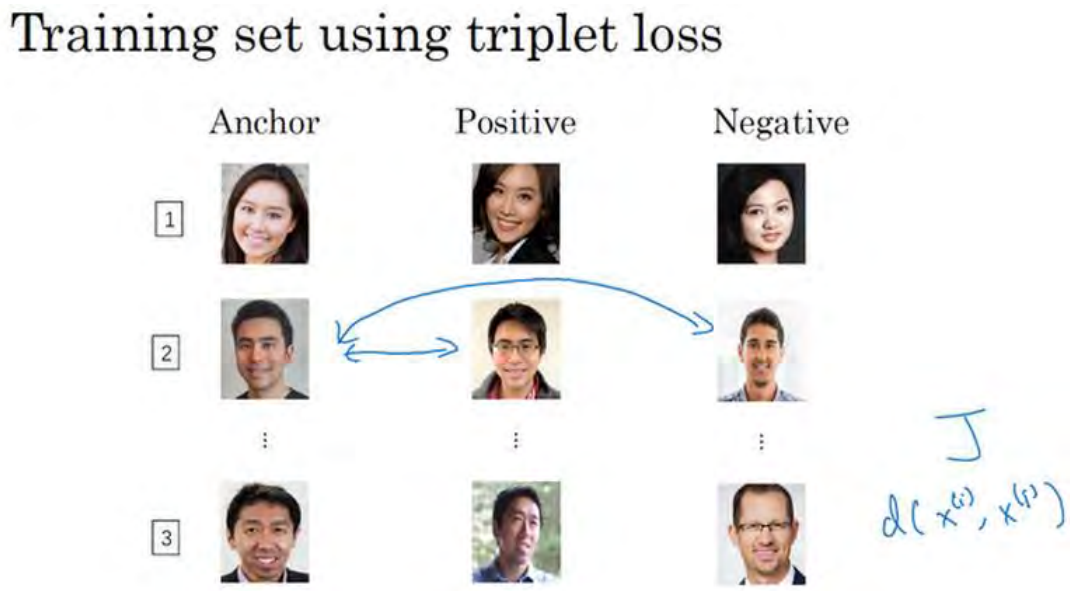
所以为了构建一个数据集，你要做的就是尽可能选择难训练的三元组A、P和N。具体而言，你想要所有的三元组都满足这个条件（$d(A,P)+a\ leqd(A,N)$），难训练的三元组就是，你的A、P和N的选择使得$d(A,P)$很接近$d(A,N)$，即$d(A,P)\approx d(A,N)$，这样 你的学习算法会竭尽全力使右边这个式子变大（$d(A,N)$），或者使左边这个式子（$d(A,P)$）变小，这样左右两边至少有一个a的间隔。并且选择这样的三元组还可以增加你的学习算法的计算效率，如果随机的选择这些三元组，其中有太多会很简单，梯度算法不会有什么效果，因为网络总是很轻松就能得到正确的结果，只有选择难的三元组梯度下降法才能发挥作用，使得这两边离得尽可能远。

如果你对此感兴趣的话，这篇论文中有更多细节，作者是 **Florian Schroff**，**Dmitry Kalenichenko**，**James Philbin**，他们建立了这个叫做 **FaceNet** 的系统，我视频里许多的观点都是来自于他们的工作。

[Schroff et al.,2015, FaceNet: A unified embedding for face recognition and clustering]

顺便说一下，这有一个有趣的事实，关于在深度学习领域，算法是如何命名的。如果你研究一个特定的领域，假如说“某某”领域，通常会将系统命名为“某某”网络或者深度“某某”，我们一直讨论人脸识别，所以这篇论文叫做 **FaceNet**(人脸网络)，上个视频里你看到过 **DeepFace**(深度人脸)。“某某”网络或者深度“某某”，是深度学习领域流行的命名算法的方式，你可以看一下这篇论文，如果你想要了解更多的关于通过选择最有用的三元组训练来加速算法的细节，这是一个很棒的论文。

总结一下，训练这个三元组损失你需要取你的训练集，然后把它做成很多三元组，这就是一个三元组（编号 1），有一个 **anchor** 图片和 **positive** 图片，这两个（**anchor** 和 **positive**）是同一个人，还有一张另一个人的 **negative** 图片。这是另一组（编号 2），其中 **anchor** 和 **positive** 图片是同一个人，但是 **anchor** 和 **negative** 不是同一个人，等等。



定义了这些包括A、P和N图片的数据集之后，你还需要做的就是用梯度下降最小化我们之前定义的代价函数 J，这样做的效果就是反向传播到网络中的所有参数来学习到一种编码，使得如果两个图片是同一个人，那么它们的d就会很小，如果两个图片不是同一个人，它们的d 就会很大。

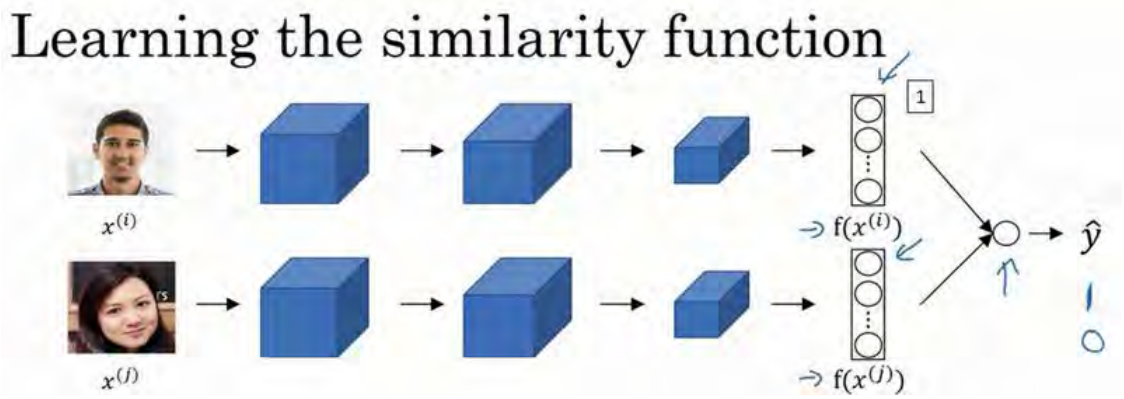
这就是三元组损失，并且如何用它来训练网络输出一个好的编码用于人脸识别。现在的人脸识别系统，尤其是大规模的商业人脸识别系统都是在很大的数据集上训练，超过百万图片的数据集并不罕见，一些公司用千万级的图片，还有一些用上亿的图片来训练这些系统。这些是很大的数据集，即使按照现在的标准，这些数据集并不容易获得。幸运的是，一些公司已经训练了这些大型的网络并且上传了模型参数。所以相比于从头训练这些网络，在这一领域，由于这些数据集太大，这一领域的一个实用操作就是下载别人的预训练模型，而不是

一切都要从头开始。但是即使你下载了别人的预训练模型，我认为了解怎么训练这些算法也是有用的，以防针对一些应用你需要从头实现这些想法。

这就是三元组损失，下个视频中，我会给你展示 **Siamese** 网络的一些其他变体，以及如何训练这些网络，让我们进入下个视频吧。

4.5 面部验证与二分类（Face verification and binary classification）

Triplet loss 是一个学习人脸识别卷积网络参数的好方法，还有其他学习参数的方法，让我们看看如何将人脸识别当成一个二分类问题。



另一个训练神经网络的方法是选取一对神经网络，选取 **Siamese** 网络，使其同时计算这些嵌入，比如说 128 维的嵌入（编号 1），或者更高维，然后将其输入到逻辑回归单元，然后进行预测，如果是相同的人，那么输出是 1，若是不同的人，输出是 0。这就把人脸识别问题转换为一个二分类问题，训练这种系统时可以替换 **triplet loss** 的方法。

最后的逻辑回归单元是怎么处理的？输出$\hat{y}$会变成，比如说 **sigmoid** 函数应用到某些特征上，相比起直接放入这些编码（$f(x^{(i)}), f(x^{(j)})$），你可以利用编码之间的不同。

$$y = \sigma(\sum_{k=1}^{128} w_i |f(x^{(i)})_k - f(x^{(j)})_k| + b)$$

我解释一下，符号$f(x^{(i)})_k$代表图片$x^{(i)}$的编码，下标k代表选择这个向量中的第k个元素，$|f(x^{(i)})_k - f(x^{(j)})_k|$对这两个编码取元素差的绝对值。你可能想，把这 128 个元素当作特征，然后把他们放入逻辑回归中，最后的逻辑回归可以增加参数w_i和b，就像普通的逻辑回归一样。你将在这 128 个单元上训练合适的权重，用来预测两张图片是否是一个人，这是一个很合理的方法来学习预测 0 或者 1，即是否是同一个人。

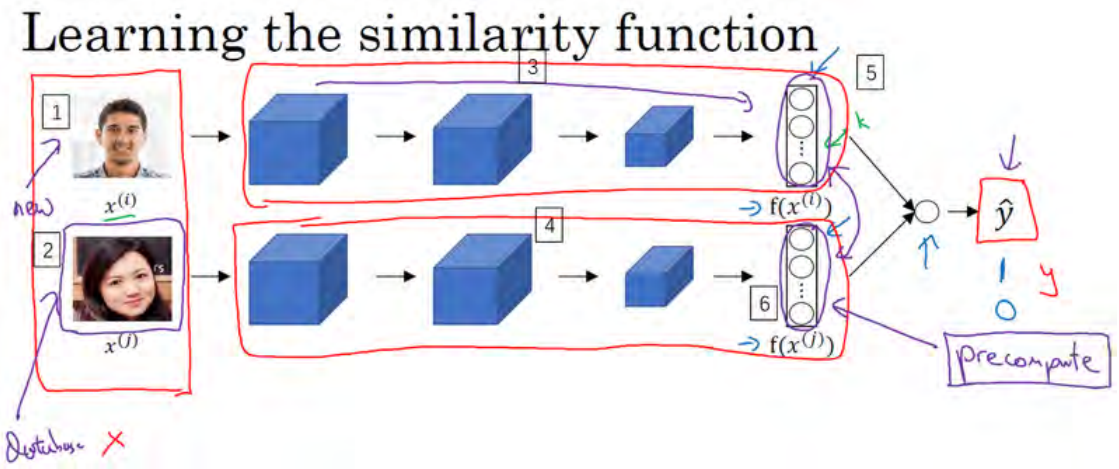
$$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_i \frac{|f(x^{(i)})_k - f(x^{(j)})_k|}{\frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k}} + b\right)$$

χ^2

还有其他不同的形式来计算绿色标记的这部分公式（$|f(x^{(i)})_k - f(x^{(j)})_k|$），比如说，公式可以是$\frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k}$，这个公式也被叫做$\chi^2$公式，是一个希腊字母$\chi$，也被称为$\chi$平方相似度。

[Taigman et. al., 2014. DeepFace closing the gap to human level performance]

这些公式及其变形在这篇 **DeepFace** 论文中有讨论，我之前也引用过。

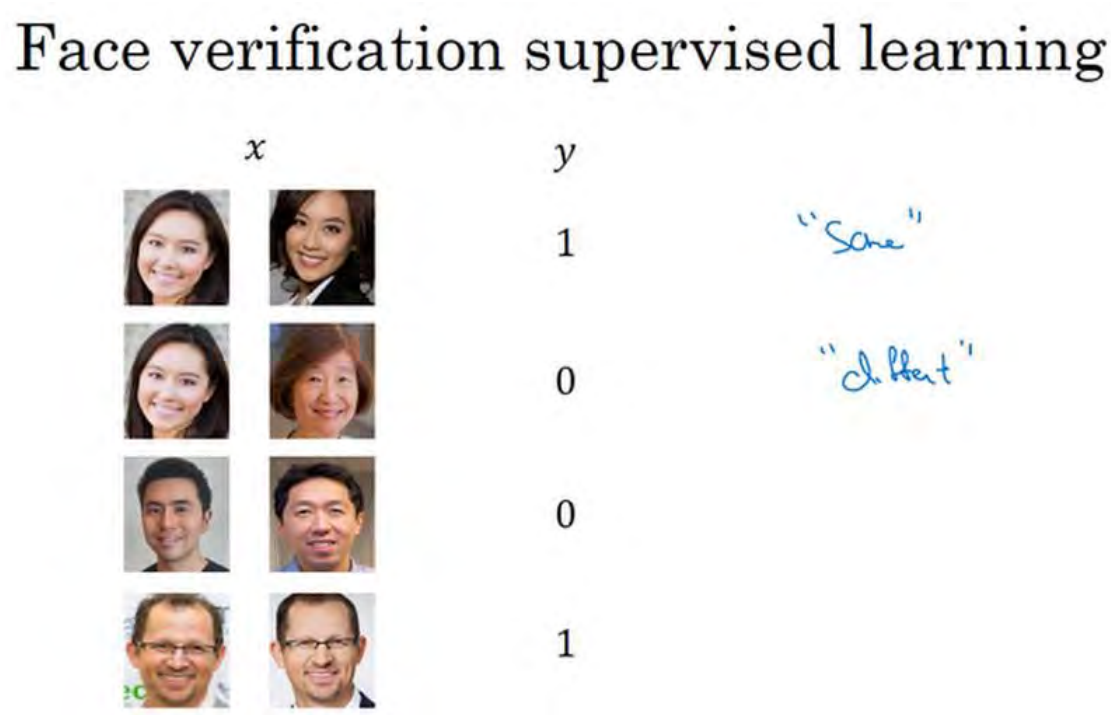


但是在这个学习公式中，输入是一对图片，这是你的训练输入x（编号 1、2），输出y是 0 或者 1，取决于你的输入是相似图片还是非相似图片。与之前类似，你正在训练一个 **Siamese** 网络，意味着上面这个神经网络拥有的参数和下面神经网络的相同（编号 3 和 4 所示的网络），两组参数是绑定的，这样的系统效果很好。

之前提到一个计算技巧可以帮你显著提高部署效果，如果这是一张新图片（编号 1），当员工走进门时，希望门可以自动为他们打开，这个（编号 2）是在数据库中的图片，不需要每次都计算这些特征（编号 6），不需要每次都计算这个嵌入，你可以提前计算好，那么当一个新员工走近时，你可以使用上方的卷积网络来计算这些编码（编号 5），然后使用它，和预先计算好的编码进行比较，然后输出预测值$\hat{y}$。

因为不需要存储原始图像，如果你有一个很大的员工数据库，你不需要为每个员工每次都计算这些编码。这个预先计算的思想，可以节省大量的计算，这个预训练的工作可以用在 **Siamese** 网路结构中，将人脸识别当作一个二分类问题，也可以用在学习和使用 **triplet** 损失函数上，我在之前的视频中描述过。

总结一下，把人脸验证当作一个监督学习，创建一个只有成对图片的训练集，不是三个一组，而是成对的图片，目标标签是 1 表示一对图片是一个人，目标标签是 0 表示图片中是不同的人。利用不同的成对图片，使用反向传播算法去训练神经网络，训练 **Siamese** 神经网络。

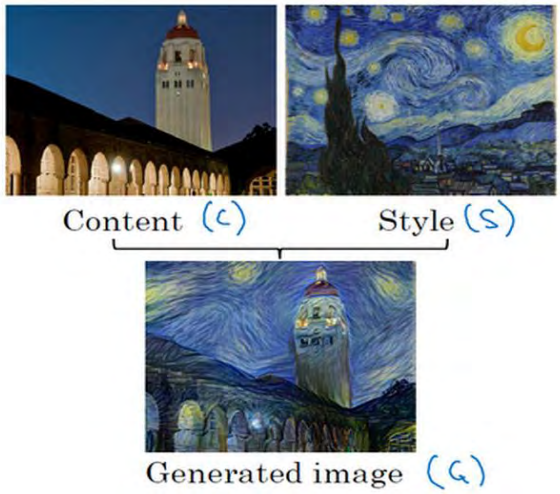


这个你看到的版本，处理人脸验证和人脸识别扩展为二分类问题，这样的效果也很好。我希望你知道，在一次学习时，你需要什么来训练人脸验证，或者人脸识别系统。

4.6 什么是神经风格转换？（What is neural style transfer?）

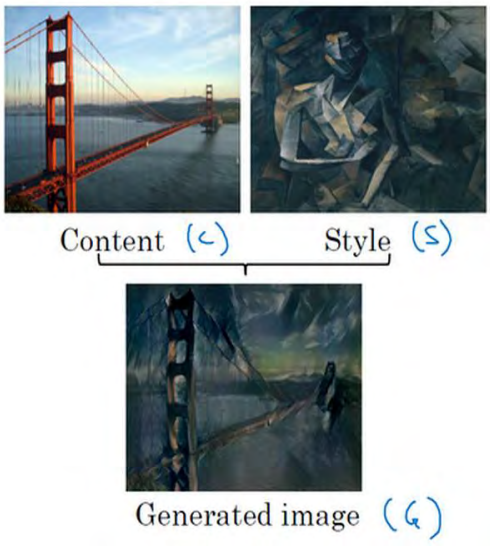
最近，卷积神经网络最有趣的应用是神经风格迁移，在编程作业中，你将自己实现这部分并创造出你的艺术作品。

什么是神经风格迁移？让我们来看几个例子，比如这张照片，照片是在斯坦福大学拍摄的，离我的办公室不远，你想利用右边照片的风格来重新创造原本的照片，右边的是梵高的星空，神经风格迁移可以帮你生成下面这张照片。



这仍是斯坦福大学的照片，但是用右边图像的风格画出来。

为了描述如何实现神经网络迁移，我将使用C来表示内容图像，S表示风格图像，G表示生成的图像。



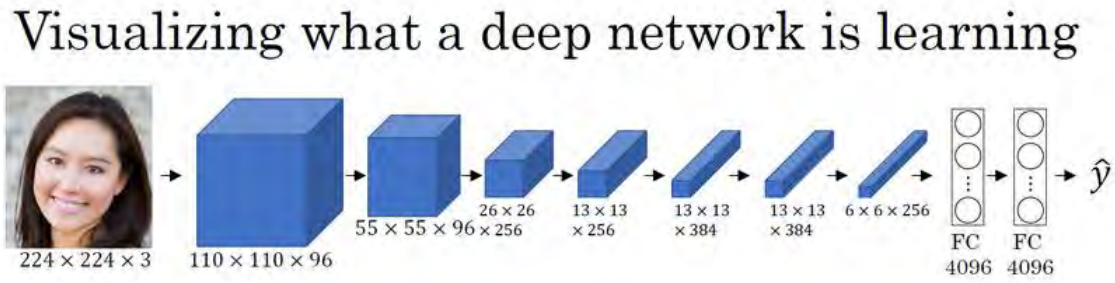
另一个例子，比如，这张图片，C代表在旧金山的金门大桥，还有这张风格图片，是毕加索的风格，然后把两张照片结合起来，得到G这张毕加索风格的的金门大桥。

这页中展示的例子，是由 **Justin Johnson** 制作，在下面几个视频中你将学到如何自己生成这样的图片。

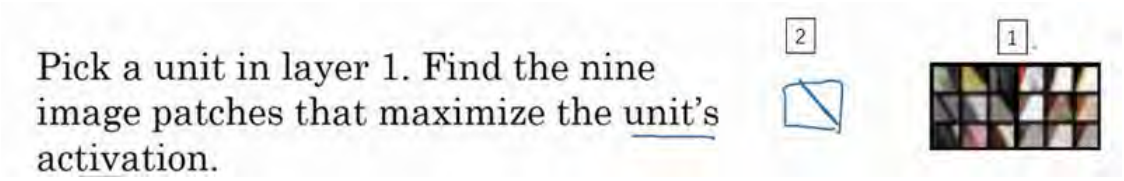
为了实现神经风格迁移，你需要知道卷积网络提取的特征，在不同的神经网络，深层的、浅层的。在深入了解如何实现神经风格迁移之前，我将在下一个视频中直观地介绍卷积神经网络不同层之间的具体运算，让我们来看下一个视频。

4.7 什么是深度卷积网络？（What are deep ConvNets learning?）

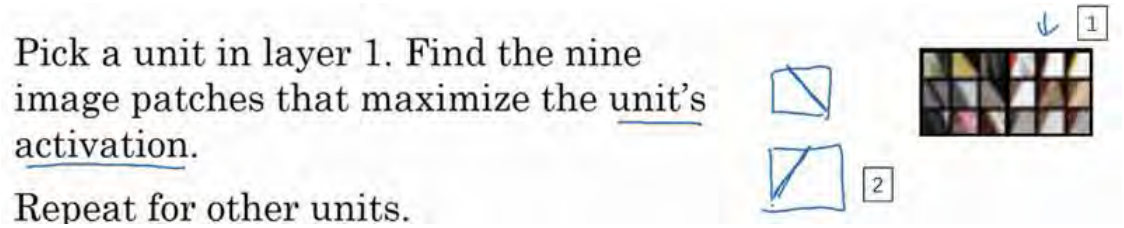
深度卷积网络到底在学什么？在这个视频中我将展示一些可视化的例子，可以帮助你理解卷积网络中深度较大的层真正在做什么，这样有助于理解如何实现神经风格迁移。



来看一个例子，假如你训练了一个卷积神经网络，是一个 **Alex** 网络，轻量级网络，你希望将看到不同层之间隐藏单元的计算结果。

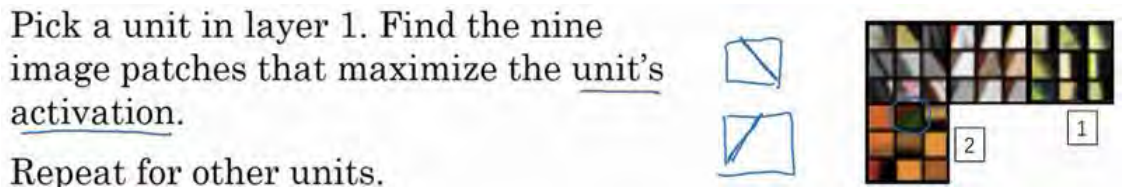


你可以这样做，从第一层的隐藏单元开始，假设你历遍了训练集，然后找到那些使得单元激活最大化的一些图片，或者是图片块。换句话说，将你的训练集经过神经网络，然后弄明白哪一张图片最大限度地激活特定的单元。注意在第一层的隐藏单元，只能看到小部分卷积神经，如果要画出来哪些激活了激活单元，只有一小块图片块是有意义的，因为这就是特定单元所能看到的全部。你选择一个隐藏单元，发现有 9 个图片最大化了单元激活，你可能找到这样的 9 个图片块（编号 1），似乎是图片浅层区域显示了隐藏单元所看到的，找到了像这样的边缘或者线（编号 2），这就是那 9 个最大化地激活了隐藏单元激活项的图片块。

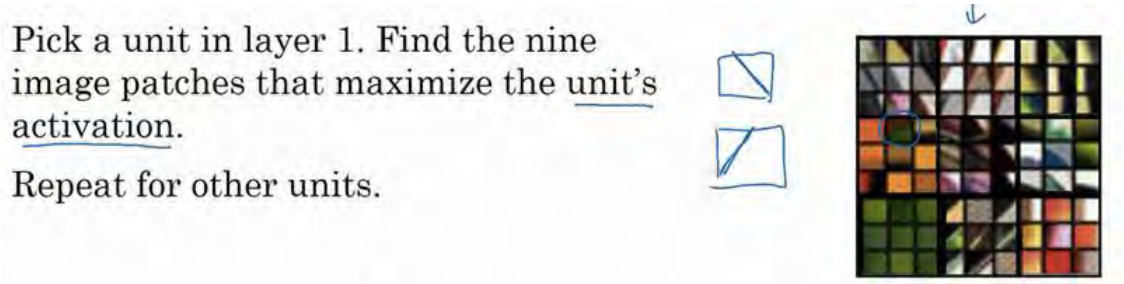


然后你可以选一个另一个第一层的隐藏单元，重复刚才的步骤，这是另一个隐藏单元，似乎第二个由这 9 个图片块（编号 1）组成。看来这个隐藏单元在输入区域，寻找这样的线

条（编号 2），我们也称之为接受域。



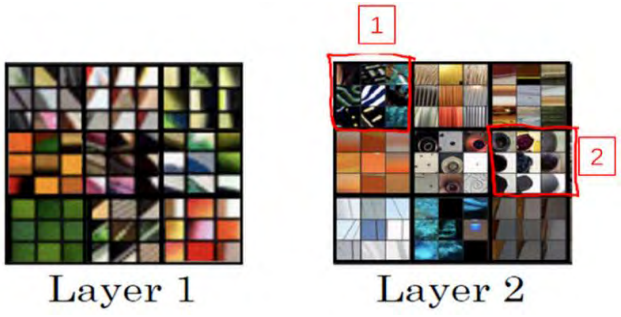
对其他隐藏单元也进行处理，会发现其他隐藏单元趋向于激活类似于这样的图片。这个似乎对垂直明亮边缘左边有绿色的图片块（编号 1）感兴趣，这一个隐藏单元倾向于橘色，这是一个有趣的图片块（编号 2），红色和绿色混合成褐色或者棕橙色，但是神经元仍可以激活它。



以此类推，这是 9 个不同的代表性神经元，每一个不同的图片块都最大化地激活了。你可以这样理解，第一层的隐藏单元通常会找一些简单的特征，比如说边缘或者颜色阴影。[Zeiler and Fergus., 2013, Visualizing and understanding convolutional networks]

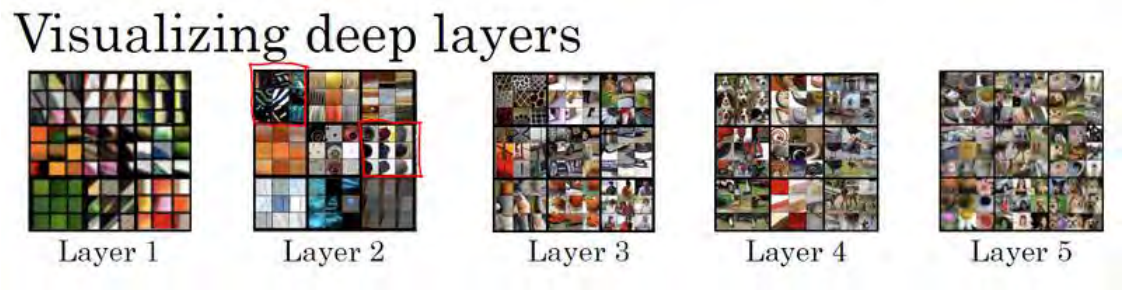
我在这个视频中使用的所有例子来自于 **Matthew Zener** 和 **Rob Fergus** 的这篇论文，题目是《可视化理解卷积神经网络》，我会使用一种更简单的方法来可视化神经网络隐藏单元的计算内容。如果你读过他们的论文，他们提出了一些更复杂的方式来可视化卷积神经网络的计算。

你已经在第一层的 9 个隐藏单元重复了这个过程好几遍，如果在深层的隐藏单元中进行这样的计算呢？卷积神经网络的深层部分学到了什么？在深层部分，一个隐藏单元会看到一张图片更大的部分，在极端的情况下，可以假设每一个像素都会影响到神经网络更深层的输出，靠后的隐藏单元可以看到更大的图片块，我还会画出和这页中的大小相同的图片块。

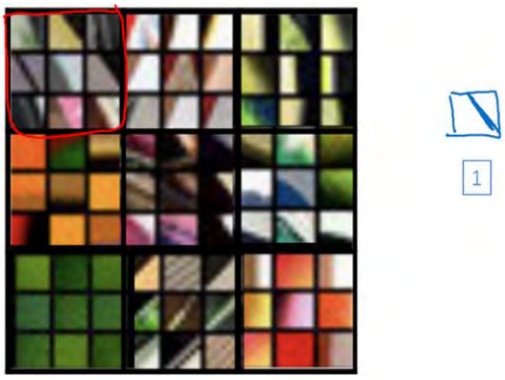


但如果我们重复这一过程，这（**Layer 1** 所示图片）是之前第一层得到的，这个（**Layer**

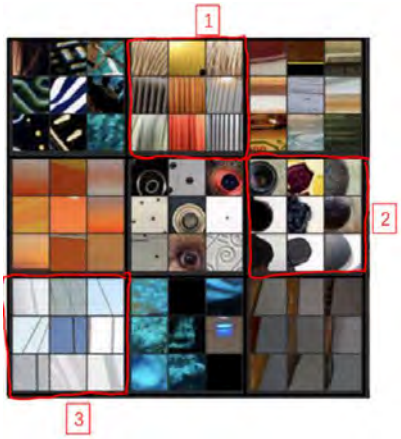
2 所示图片）是可视化的第 2 层中最大程度激活的 9 个隐藏单元。我想解释一下这个可视化，这是（编号 2 所示）使一个隐藏单元最大激活的 9 个图片块，每一个组合，这是另一组（编号 2），使得一个隐藏单元被激活的 9 个图片块，这个可视化展示了第二层的 9 个隐藏单元，每一个又有 9 个图片块使得隐藏单元有较大的输出或是较大的激活。



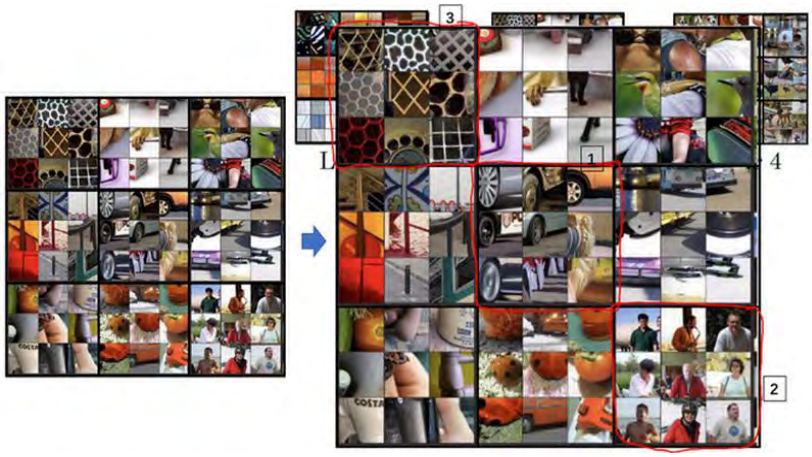
在更深的层上，你可以重复这个过程。



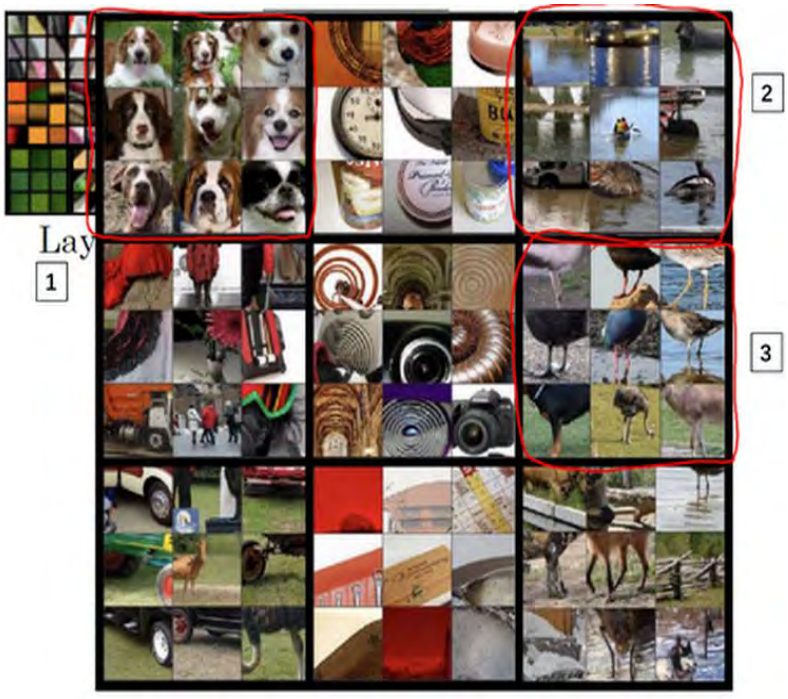
在这页里很难看清楚，这些微小的浅层图片块，让我们放大一些，这是第一层，这是第一个被高度激活的单元，你能在输入图片的区域看到，大概是这个角度的边缘（编号 1）放大第二层的可视化图像。



有意思了，第二层似乎检测到更复杂的形状和模式，比如说这个隐藏单元（编号 1），它会找到有很多垂线的垂直图案，这个隐藏单元（编号 2）似乎在左侧有圆形图案时会被高度激活，这个的特征（编号 3）是很细的垂线，以此类推，第二层检测的特征变得更加复杂。



看看第三层我们将其放大，放得更大一点，看得更清楚一点，这些东西激活了第三层。再放大一点，这又很有趣了，这个隐藏单元（编号 1）似乎对图像左下角的圆形很敏感，所以检测到很多车。这一个（编号 2）似乎开始检测到人类，这个（编号 3）似乎检测特定的图案，蜂窝形状或者方形，类似这样规律的图案。有些很难看出来，需要手动弄明白检测到什么，但是第三层明显，检测到更复杂的模式。



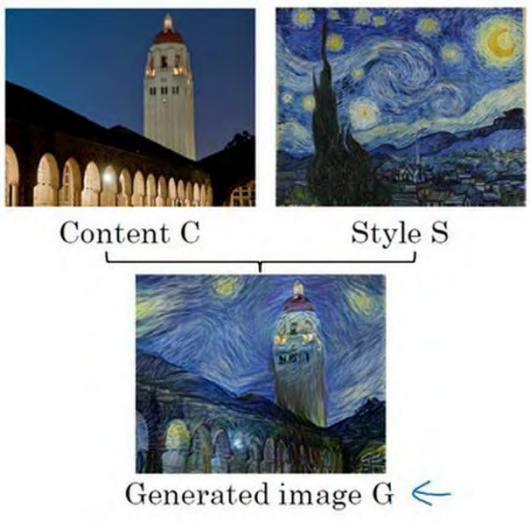
下一层呢？这是第四层，检测到的模式和特征更加复杂，这个（编号 1）学习成了一个狗的检测器，但是这些狗看起来都很类似，我并不知道这些狗的种类，但是你知道这些都是狗，他们看起来也类似。第四层中的这个（编号 2）隐藏单元它检测什么？水吗？这个（编号 3）似乎检测到鸟的脚等等。

第五层检测到更加复杂的事物，注意到这（编号 1）也有一个神经元，似乎是一个狗检测器，但是可以检测到的狗似乎更加多样性。这个（编号 2）可以检测到键盘，或者是键盘质地的物体，可能是有很多点的物体。我认为这个神经元（编号 3）可能检测到文本，但是很难确定，这个（编号 4）检测到花。我们已经有了一些进展，从检测简单的事物，比如说，第一层的边缘，第二层的质地，到深层的复杂物体。

我希望这让你可以更直观地了解卷积神经网络的浅层和深层是如何计算的，接下来让我们使用这些知识开始构造神经风格迁移算法。

4.8 代价函数（Cost function）

要构建一个神经风格迁移系统，让我们为生成的图像定义一个代价函数，你接下看到的是，通过最小化代价函数，你可以生成你想要的任何图像。



记住我们的问题，给你一个内容图像C，给定一个风格图片S，而你的目标是生成一个新图片G。为了实现神经风格迁移，你要做的是定义一个关于G的代价函数J用来评判某个生成图像的好坏，我们将使用梯度下降法去最小化$J(G)$，以便于生成这个图像。

怎么判断生成图像的好坏呢？我们把这个代价函数定义为两个部分。

$$J_{\text{content}}(C,G)$$

第一部分被称作内容代价，这是一个关于内容图片和生成图片的函数，它是用来度量生成图片G的内容与内容图片C的内容有多相似。

$$J_{\text{style}}(S,G)$$

然后我们会把结果加上一个风格代价函数，也就是关于S和G的函数，用来度量图片G的风格和图片S的风格的相似度。

$$J(G)=aJ_{\text{content}}(C,G)+\beta J_{\text{style}}(S,G)$$

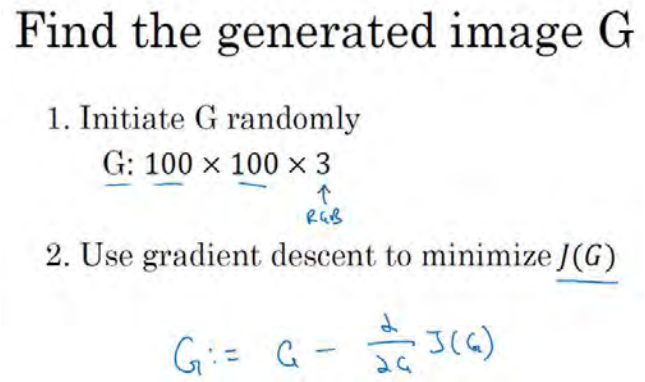
最后我们用两个超参数a和β来来确定内容代价和风格代价，两者之间的权重用两个超参数来确定。两个代价的权重似乎是多余的，我觉得一个超参数似乎就够了，但提出神经风格迁移的原始作者使用了两个不同的超参数，我准备保持一致。

[Gatys et al., 2015. A neural algorithm of artistic style. Images on slide generated by Justin Johnson]

关于神经风格迁移算法我将在接下来几段视频中展示的，是基于 **Leon Gatys**， **Alexandra Ecker** 和 **Matthias Bethge** 的这篇论文。这篇论文并不是很难读懂，如果你愿意，看完这些视

频，我也非常推荐你去看看他们的论文。

算法的运行是这样的，对于代价函数$J(G)$，为了生成一个新图像，你接下来要做的是随机初始化生成图像G，，它可能是 100×100×3，可能是 500×500×3，又或者是任何你想要的尺寸。



然后使用在之前的幻灯片上定义的代价函数$J(G)$，你现在可以做的是使用梯度下降的方法将其最小化，更新$G:=G-\frac{\partial}{\partial G}J(G)$。在这个步骤中，你实际上更新的是图像$G$的像素值，也就是 100×100×3，比如 **RGB** 通道的图片。

这里有个例子，假设你从这张内容图片（编号 1）和风格（编号 2）图片开始，这是另一张公开的毕加索画作，当你随机初始化G，你随机初始化的生成图像就是这张随机选取像素的白噪声图（编号 3）。接下来运行梯度下降算法，最小化代价函数$J(G)$，逐步处理像素，这样慢慢得到一个生成图片（编号 4、5、6），越来越像用风格图片的风格画出来的内容图片。

在这段视频中你看到了神经风格迁移算法的概要，定义一个生成图片G的代价函数，并将其最小化。接下来我们需要了解怎么去定义内容代价函数和风格代价函数，让我们从下一个视频开始学习这部分内容吧。

4.9 内容代价函数（Content cost function）

风格迁移网络的代价函数有一个内容代价部分，还有一个风格代价部分。

$$J(G) = \alpha J_{\text{content}}(C, G) + \beta J_{\text{style}}(S, G)$$

我们先定义内容代价部分，不要忘了这就是我们整个风格迁移网络的代价函数，我们看看内容代价函数应该是什么。

- Say you use hidden layer l to compute content cost.
- Use pre-trained ConvNet. (E.g., VGG network)

假如说，你用隐含层l来计算内容代价，如果l是个很小的数，比如用隐含层 1，这个代价函数就会使你的生成图片像素上非常接近你的内容图片。然而如果你用很深的层，那么那就会问，内容图片里是否有狗，然后它就会确保生成图片里有一个狗。所以在实际中，这个层 l 在网络中既不会选的太浅也不会选的太深。因为你要自己做这周结束的编程练习，我会让你获得一些直觉，在编程练习中的具体例子里通常 l 会选择在网络的中间层，既不太浅也不很深，然后用一个预训练的卷积模型，可以是 **VGG 网络**或者其他的网络也可以。

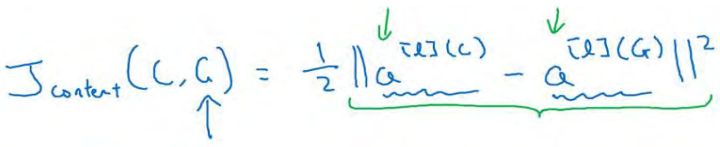
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

现在你需要衡量假如有一个内容图片和一个生成图片他们在内容上的相似度，我们令这个$a^{[l][C]}$和$a^{[l][G]}$，代表这两个图片C和G的l层的激活函数值。如果这两个激活值相似，那么就意味着两个图片的内容相似。

我们定义这个

$$J_{\text{content}}(C, G) = \frac{1}{2}||a^{[l][C]} - a^{[l][G]}||^2$$

为两个激活值不同或者相似的程度，我们取 l 层的隐含单元的激活值，按元素相减，内容图片的激活值与生成图片相比较，然后取平方，也可以在前面加上归一化或者不加，比如$\frac{1}{2}$或者其他的，都影响不大,因为这都可以由这个超参数 α 来调整（$J(G) = \alpha J_{\text{content}}(C, G) + \beta J_{\text{style}}(S, G)$）。

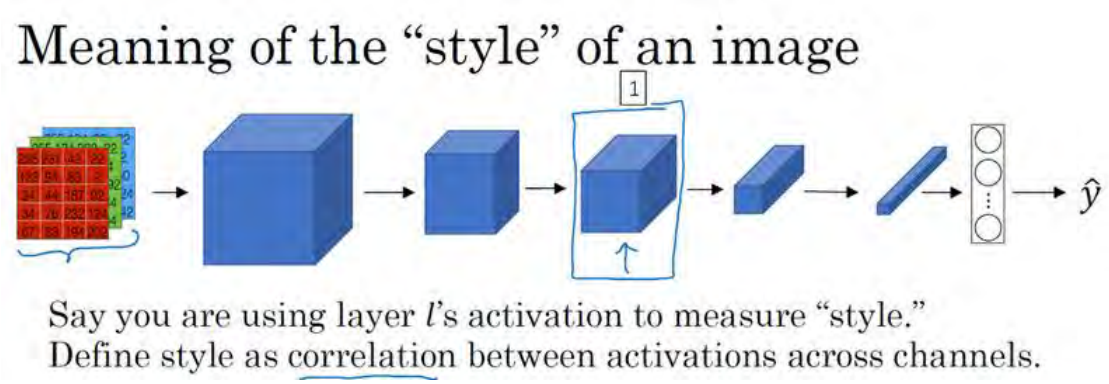


要清楚我这里用的符号都是展成向量形式的，这个就变成了这一项（$a^{[l][C]}$）减这一项（$a^{[l][C]}$）的$L2$范数的平方，在把他们展成向量后。这就是两个激活值间的差值平方和，这就是两个图片之间l层激活值差值的平方和。后面如果对$J(G)$做梯度下降来找G的值时，整个代价函数会激励这个算法来找到图像G，使得隐含层的激活值和你内容图像的相似。

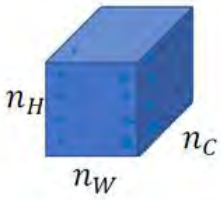
这就是如何定义风格迁移网络的内容代价函数，接下来让我们学习风格代价函数。

4.10 风格代价函数（Style cost function）

在上节视频中，我们学习了如何为神经风格迁移定义内容代价函数，这节课我们来了解风格代价函数。那么图片的风格到底是什么意思呢？



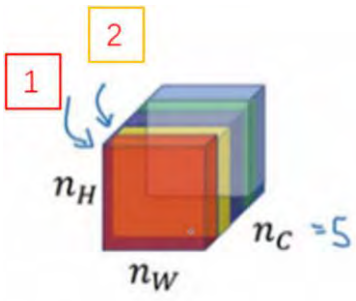
这么说吧，比如你有这样一张图片，你可能已经对这个计算很熟悉了，它能算出这里是否含有不同隐藏层。现在你选择了某一层l（编号 1），比如这一层去为图片的风格定义一个深度测量，现在我们要做的就是将图片的风格定义为l层中各个通道之间激活项的相关系数。



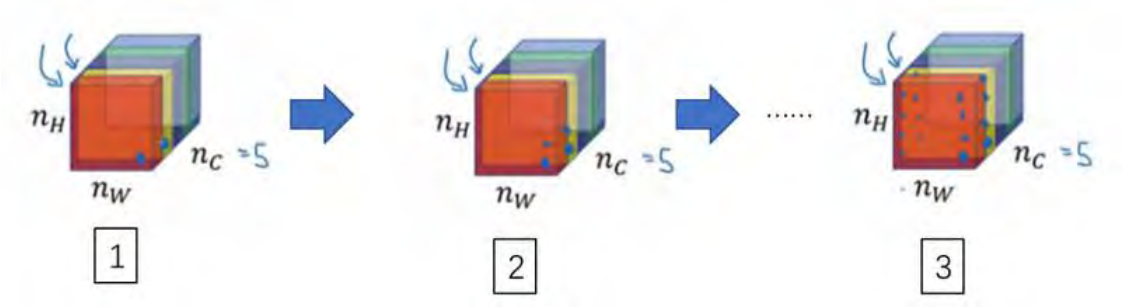
How correlated are the activations across different channels?

我来详细解释一下，现在你将 l 层的激活项取出，这是个$n_H \times n_W \times n_C$的激活项，它是一个三维的数据块。现在问题来了，如何知道这些不同通道之间激活项的相关系数呢？

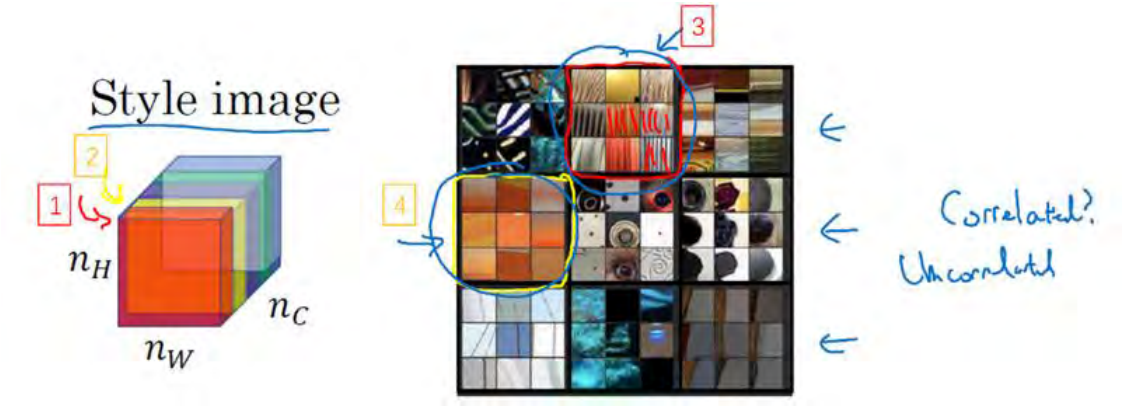
为了解释这些听起来很含糊不清的词语，现在注意这个激活块，我把它的不同通道渲染成不同的颜色。在这个例子中，假如我们有 5 个通道为了方便讲解，我将它们染成了五种颜色。一般情况下，我们在神经网络中会有许多通道，但这里只用 5 个通道，会更方便我们理解。



为了能捕捉图片的风格，你需要进行下面这些操作，首先，先看前两个通道，前两个通道（编号 1、2）分别是图中的红色和黄色部分，那我们该如何计算这两个通道间激活项的相关系数呢？



举个例子，在视频的左下角在第一个通道中含有某个激活项，第二个通道也含有某个激活项，于是它们组成了一对数字（编号 1 所示）。然后我们再看看这个激活项块中其他位置的激活项，它们也分别组成了很多对数字（编号 2，3 所示），分别来自第一个通道，也就是红色通道和第二个通道，也就是黄色通道。现在我们得到了很多个数字对，当我们取得这两个$n_H \times n_W$的通道中所有的数字对后，现在该如何计算它们的相关系数呢？它是如何决定图片风格的呢？

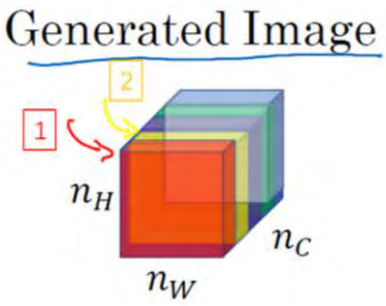


我们来看一个例子，这是之前视频中的一个可视化例子，它来自一篇论文，作者是 **Matthew Zeile** 和 **Rob Fergus** 我之前有提到过。我们知道，这个红色的通道（编号 1）对应的是这个神经元，它能找出图片中的特定位置是否含有这些垂直的纹理（编号 3），而第二个通道也就是黄色的通道（编号 2），对应这个神经元（编号 4），它可以粗略地找出橙色的区域。什么时候两个通道拥有高度相关性呢？如果它们有高度相关性，那么这幅图片中出现垂直纹理的地方（编号 2），那么这块地方（编号 4）很大概率是橙色的。如果说它们是不相关的，又是什么意思呢？显然，这意味着图片中有垂直纹理的地方很大概率不是橙色的。而相关系数描述的就是当图片某处出现这种垂直纹理时，该处又同时是橙色的可能性。

相关系数这个概念为你提供了一种去测量这些不同的特征的方法，比如这些垂直纹理，

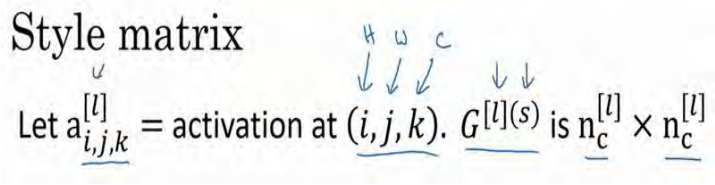
这些橙色或是其他的特征去测量它们在图片中的各个位置同时出现或不同时出现的频率。

如果我们在通道之间使用相关系数来描述通道的风格，你能做的就是测量你的生成图像中第一个通道（编号 1）是否与第二个通道（编号 2）相关，通过测量，你能得知在生成的图像中垂直纹理和橙色同时出现或者不同时出现的频率，这样你将能够测量生成的图像的风格与输入的风格图像的相似程度。



现在我们来证实这种说法，对于这两个图像，也就是风格图像与生成图像，你需要计算一个风格矩阵，说得更具体一点就是用l层来测量风格。

----------（多余内容，整理者认为）----------



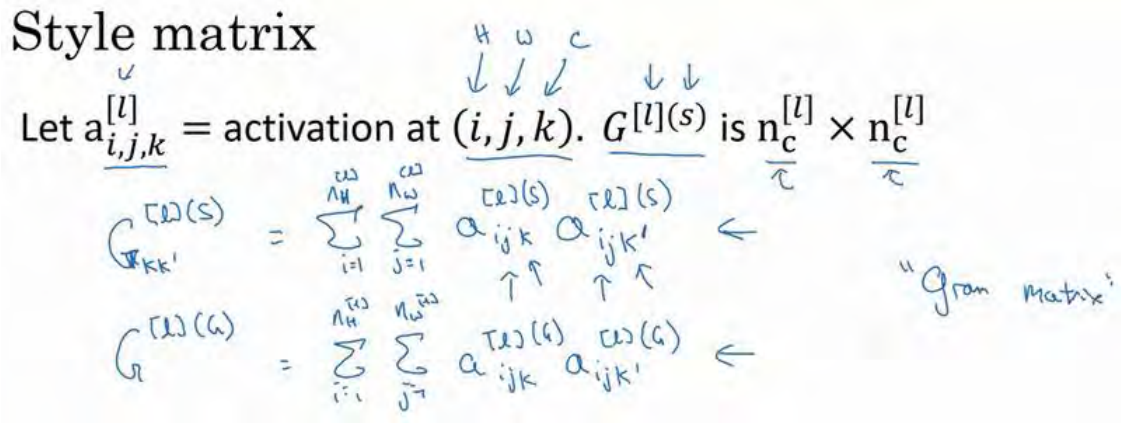
我们设$a^{[l]}_{i,j,k}$，设它为隐藏层l中(i,j,k)位置的激活项，i，j，k分别代表该位置的高度、宽度以及对应的通道数。现在你要做的就是去计算一个关于l层和风格图像的矩阵，即$G^{[l](S)}$（l表示层数，S表示风格图像），这（$G^{[l](S)}$）是一个$n_c \times n_c$的矩阵，同样地，我们也对生成的图像进行这个操作。

但是现在我们先来定义风格图像，设这个关于l层和风格图像的，G是一个矩阵，这个矩阵的高度和宽度都是l层的通道数。在这个矩阵中k和k'元素被用来描述k通道和k'通道之间的相关系数。具体地：

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](S)}_{i,j,k} a^{[l](S)}_{i,j,k'}$$

用符号i，j表示下界，对i，j，k位置的激活项$a^{[l]}_{i,j,k}$，乘以同样位置的激活项，也就是i,j,k'位置的激活项，即$a^{[l]}_{i,j,k'}$，将它们两个相乘。然后i和j分别加到 I 层的高度和宽度，即$n_H^{[l]}$和$n_W^{[l]}$，将这些不同位置的激活项都加起来。(i,j,k)和(i,j,k')中x坐标和y坐标分别对应高度

和宽度，将k通道和k'通道上这些位置的激活项都进行相乘。我一直以来用的这个公式，严格来说，它是一种非标准的互相关函数，因为我们没有减去平均数，而是将它们直接相乘。



这就是输入的风格图像所构成的风格矩阵，然后，我们再对生成图像做同样的操作。

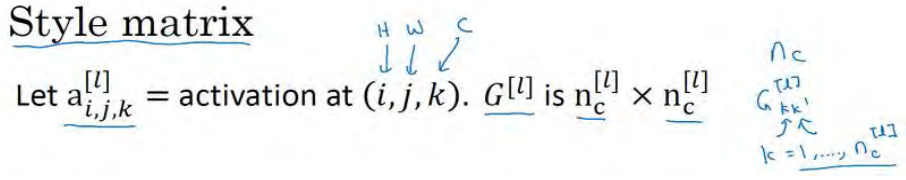
$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](G)}_{i,j,k} a^{[l](G)}_{i,j,k'}$$

$a^{[l](S)}_{i,j,k}$和$a^{[l](G)}_{i,j,k}$中的上标(S)和(G)分别表示在风格图像 S 中的激活项和在生成图像G的激活项。我们之所以用大写字母G来代表这些风格矩阵，是因为在线性代数中这种矩阵有时也叫 **Gram** 矩阵，但在这里我只把它们叫做风格矩阵。

所以你要做的就是计算出这张图像的风格矩阵，以便能够测量出刚才所说的这些相关系数。更正规地来表示，我们用$a^{[l]}_{i,j,k}$来记录相应位置的激活项，也就是l层中的i,j,k位置，所以i代表高度，j代表宽度，k代表着l中的不同通道。之前说过，我们有 5 个通道，所以k就代表这五个不同的通道。

----------（结束）----------

对于这个风格矩阵，你要做的就是计算这个矩阵也就是$G^{[l]}$矩阵，它是个$n_c \times n_c$的矩阵，也就是一个方阵。记住，因为这里有n_c个通道，所以矩阵的大小是$n_c \times n_c$。以便计算每一对激活项的相关系数，所以$G^{[l]}_{kk'}$可以用来测量k通道与k'通道中的激活项之间的相关系数，k和k'会在 1 到n_c之间取值，n_c就是l层中通道的总数量。

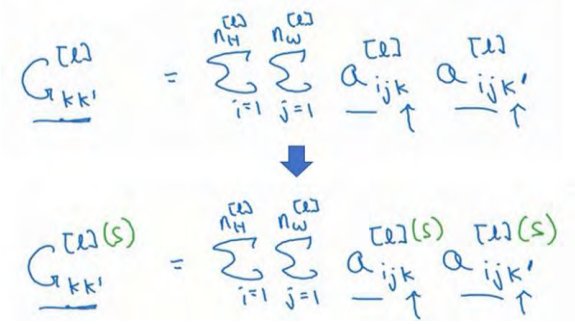


当在计算$G^{[l]}$时，我写下的这个符号（下标kk'）只代表一种元素，所以我要在右下角标

明是kk'元素，和之前一样i，j从一开始往上加，对应(i,j,k)位置的激活项与对应(i,j,k')位置的激活项相乘。记住，这个i和j是激活块中对应位置的坐标，也就是该激活项所在的高和宽，所以i会从 1 加到$n_H^{[l]}$，j 会从 1 加到$n_W^{[l]}$，k和k'则表示对应的通道，所以k和k'值的范围是从 1 开始到这个神经网络中该层的通道数量$n_C^{[l]}$。这个式子就是把图中各个高度和宽度的激活项都遍历一遍，并将k和k'通道中对应位置的激活项都进行相乘，这就是$G_{kk'}^{[l]}$的定义。通过对k和k'通道中所有的数值进行计算就得到了G矩阵，也就是风格矩阵。

$$G_{kk'}^{[l]} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{i,j,k}^{[l]} a_{i,j,k'}^{[l]}$$

要注意，如果两个通道中的激活项数值都很大，那么$G_{kk'}^{[l]}$也会变得很大，对应地，如果他们不相关那么$G_{kk'}^{[l]}$就会很小。严格来讲，我一直使用这个公式来表达直觉想法，但它其实是一种非标准的互协方差，因为我们并没有减去均值而只是把这些元素直接相乘，这就是计算图像风格的方法。

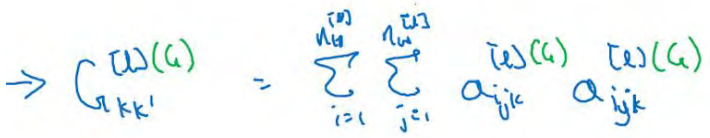


$$G_{kk'}^{[l](S)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{i,j,k}^{[l](S)} a_{i,j,k'}^{[l](S)}$$

你要同时对风格图像S和生成图像G都进行这个运算，为了区分它们，我们在它的右上角加一个(S)，表明它是风格图像S，这些都是风格图像S中的激活项，之后你需要对生成图像也做相同的运算。

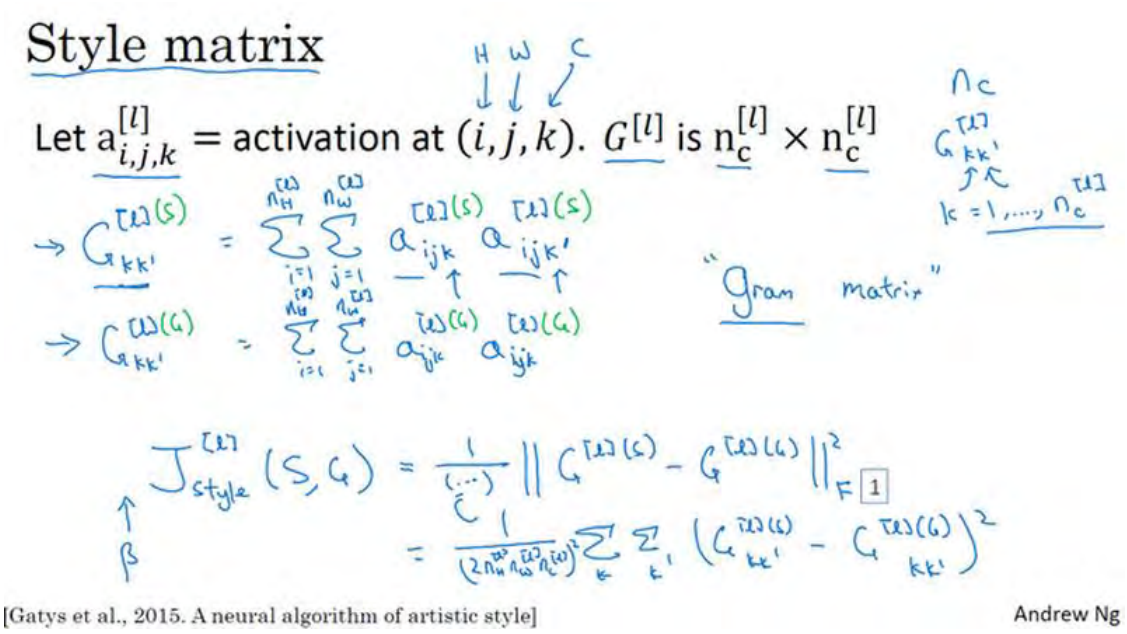
$$G_{kk'}^{[l](G)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{i,j,k}^{[l](G)} a_{i,j,k'}^{[l](G)}$$

和之前一样，再把公式都写一遍，把这些都加起来，为了区分它是生成图像，在这里放一个(G)。

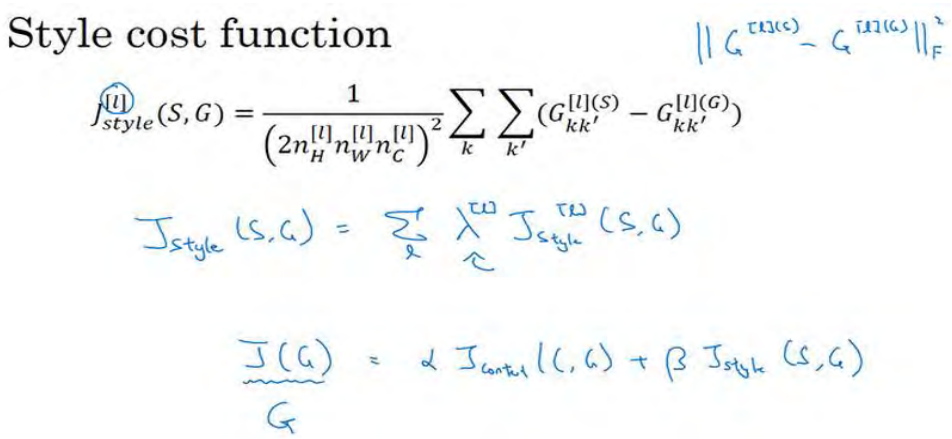


现在，我们有 2 个矩阵，分别从风格图像S和生成图像G。

再提醒一下，我们一直使用大写字母G来表示矩阵，是因为在线性代数中，这种矩阵被称为 **Gram** 矩阵，但在本视频中我把它叫做风格矩阵，我们取了 **Gram** 矩阵的首字母G来表示这些风格矩阵。



最后，如果我们将S和G代入到风格代价函数中去计算，这将得到这两个矩阵之间的误差，因为它们是矩阵，所以在这里加一个F（**Frobenius** 范数，编号 1 所示），这实际上是计算两个矩阵对应元素相减的平方的和，我们把这个式子展开，从k和k'开始作它们的差，把对应的式子写下来，然后把得到的结果都加起来，作者在这里使用了一个归一化常数，也就是$\frac{1}{2n_H^{[l]} n_W^{[l]} n_C^{[l]}}$，再在外面加一个平方，但是一般情况下你不用写这么多，一般我们只要将它乘以一个超参数β就行。



最后，这是对l层定义的风格代价函数，和之前你见到的一样，这是两个矩阵间一个基本的 **Frobenius** 范数，也就是S图像和G图像之间的范数再乘上一个归一化常数，不过这不是很重要。实际上，如果你对各层都使用风格代价函数，会让结果变得更好。如果要对各层都使用风格代价函数，你可以这么定义代价函数，把各个层的结果（各层的风格代价函数）都加起来，这样就能定义它们全体了。我们还需要对每个层定义权重，也就是一些额外的超参数，我们用$\lambda^{[l]}$来表示，这样将使你能够在神经网络中使用不同的层，包括之前的一些可以测量类似边缘这样的低级特征的层，以及之后的一些能测量高级特征的层，使得我们的神经网络在计算风格时能够同时考虑到这些低级和高级特征的相关系数。这样，在基础的训练中你在定义超参数时，可以尽可能的得到更合理的选择。

为了把这些东西封装起来，你现在可以定义一个全体代价函数：

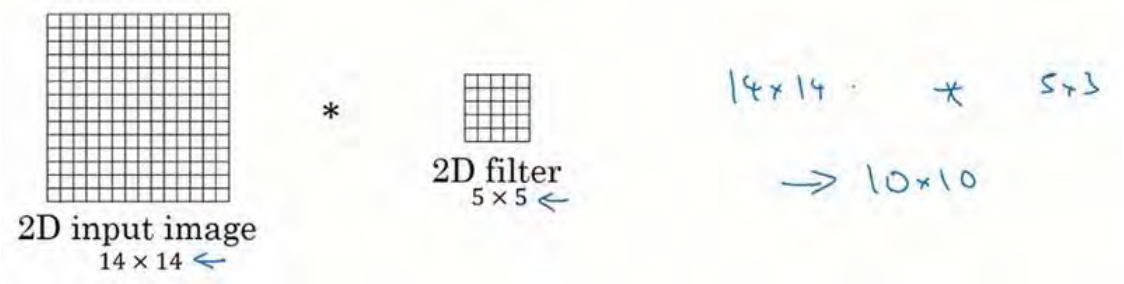
$$J(G) = aJ_{\text{content}(C,G)} + \beta J_{style}(S,G)$$

之后用梯度下降法，或者更复杂的优化算法来找到一个合适的图像G，并计算$J(G)$的最小值，这样的话，你将能够得到非常好看的结果，你将能够得到非常漂亮的结果。

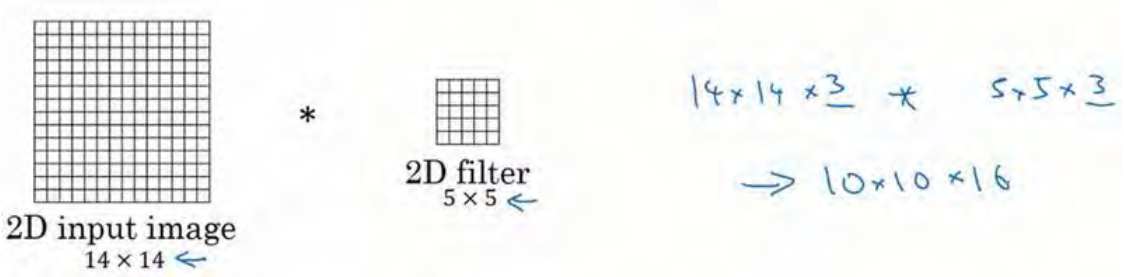
这节神经风格迁移的内容就讲到这里，希望你能愉快地在本周的基础训练中进行实践。在本周结束之前，还有最后一节内容想告诉你们，就是如何对 **1D** 和 **3D** 的数据进行卷积，之前我们处理的都是 **2D** 图片，我们下节视频再见。

4.11 一维到三维推广（1D and 3D generalizations of models）

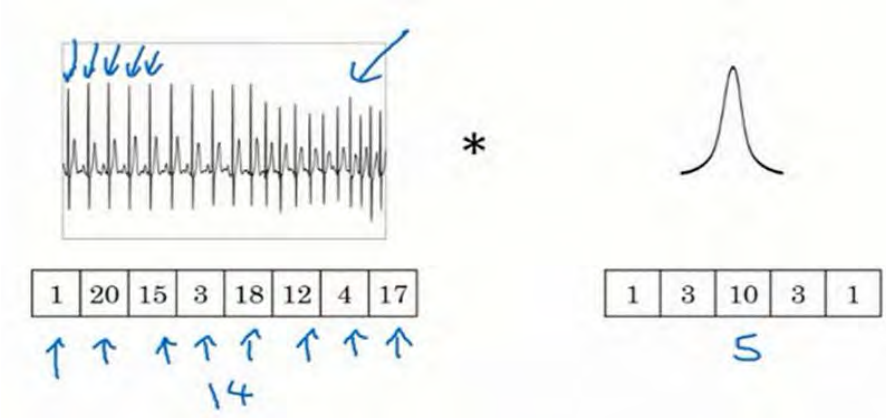
你已经学习了许多关于卷积神经网络（**ConvNets**）的知识，从卷积神经网络框架，到如何使用它进行图像识别、对象检测、人脸识别与神经网络转换。即使我们大部分讨论的图像数据，某种意义上而言都是 2D 数据，考虑到图像如此普遍，许多你所掌握的思想不仅局限于 2D 图像，甚至可以延伸至 1D，乃至 3D 数据。



让我们回头看看在第一周课程中你所学习关于 **2D** 卷积，你可能会输入一个 14×14 的图像，并使用一个 5×5 的过滤器进行卷积，接下来你看到了 14×14 图像是如何与 5×5 的过滤器进行卷积的，通过这个操作你会得到 10×10 的输出。



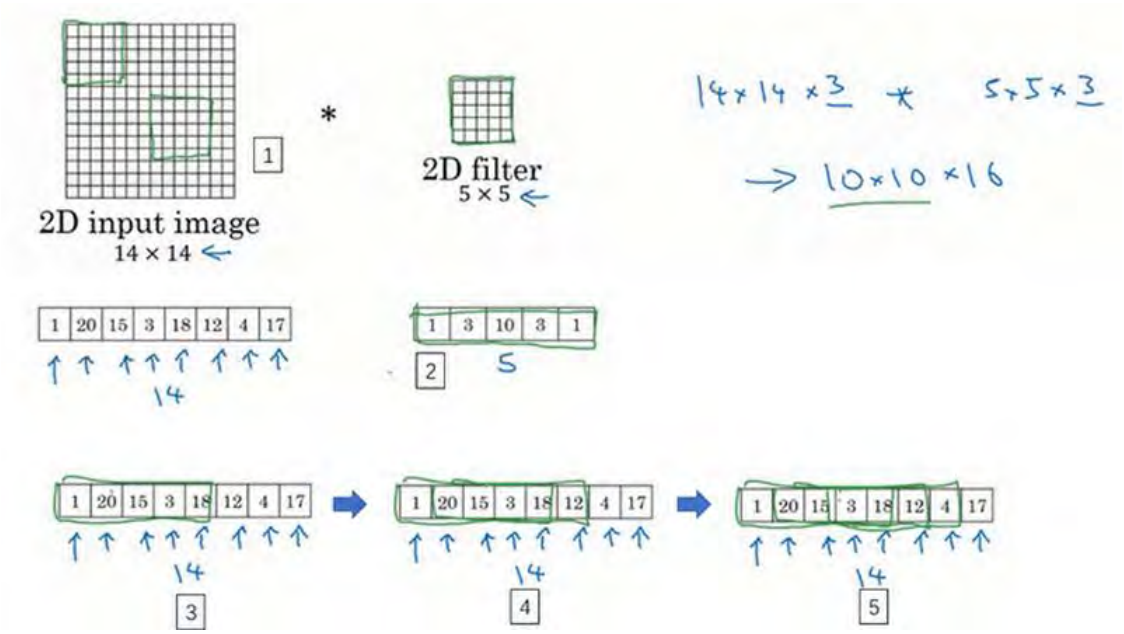
如果你使用了多通道，比如 14×14×3，那么相匹配的过滤器可能是 5×5×3，如果你使用了多重过滤，比如 16，最终你得到的是 10×10×16。



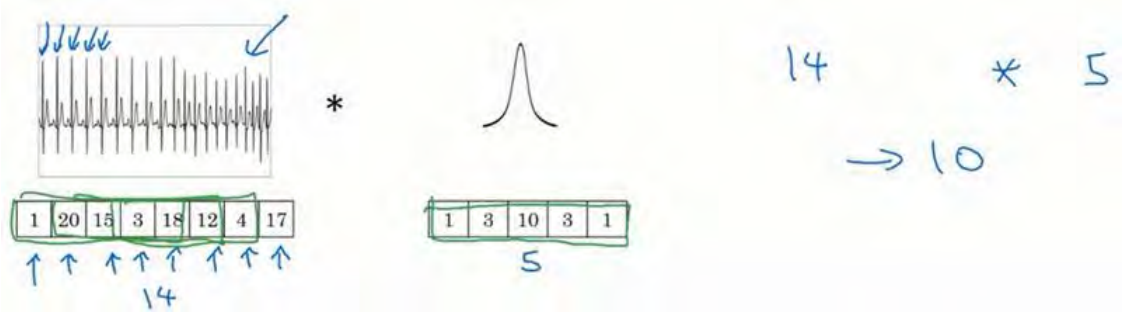
事实证明早期想法也同样可以用于 1 维数据，举个例子，左边是一个 **EKG** 信号，或者说是心电图，当你在你的胸部放置一个电极，电极透过胸部测量心跳带来的微弱电流，正因

为心脏跳动，产生的微弱电波能被一组电极测量，这就是人心跳产生的 **EKG**，每一个峰值都对应着一次心跳。

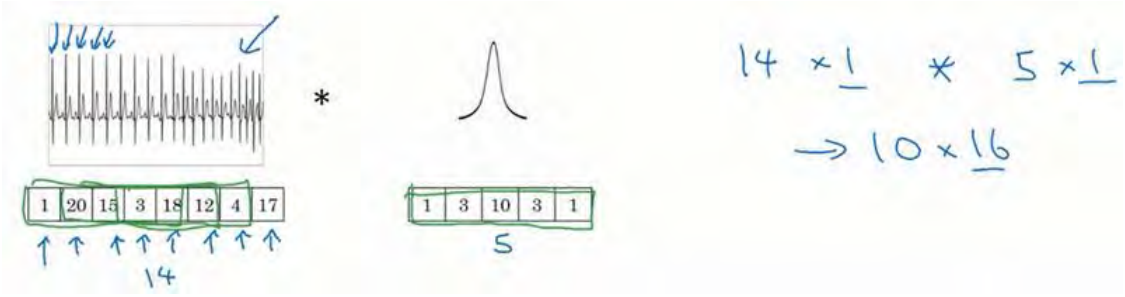
如果你想使用 **EKG** 信号，比如医学诊断，那么你将处理 1 维数据，因为 **EKG** 数据是由时间序列对应的每个瞬间的电压组成，这次不是一个 14×14 的尺寸输入，你可能只有一个 14 尺寸输入，在这种情况下你可能需要使用一个 1 维过滤进行卷积，你只需要一个 1×5 的过滤器，而不是一个 5×5 的。



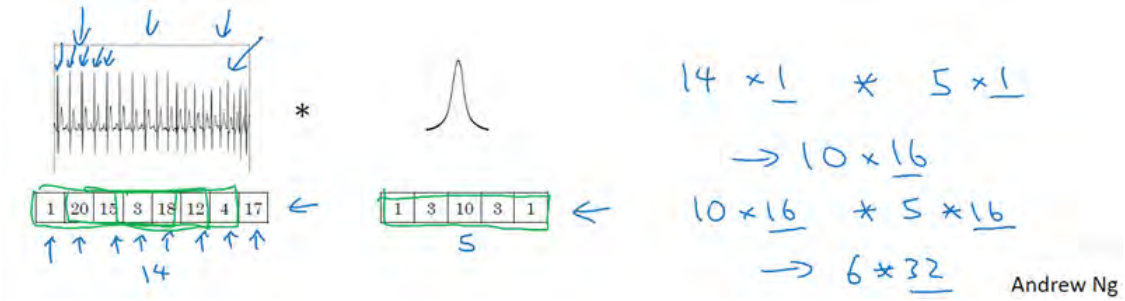
二维数据的卷积是将同一个 5×5 特征检测器应用于图像中不同的位置（编号 1 所示），你最后会得到 10×10 的输出结果。1 维过滤器可以取代你的 5 维过滤器（编号 2 所示），可在不同的位置中应用类似的方法（编号 3，4，5 所示）。



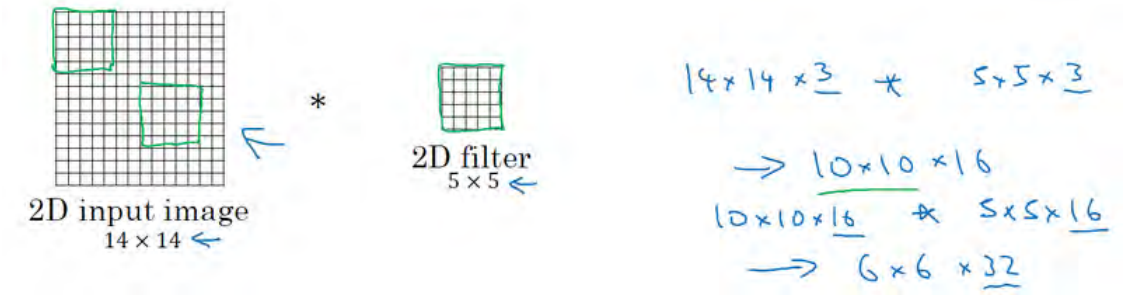
当你对这个 1 维信号使用卷积，你将发现一个 14 维的数据与 5 维数据进行卷积，并产生一个 10 维输出。



再一次如果你使用多通道，在这种场景下可能会获得一个 14×1 的通道。如果你使用一个 **EKG**，就是 5×1 的，如果你有 16 个过滤器，可能你最后会获得一个 10×16 的数据，这可能会是你卷积网络中的某一层。



对于卷积网络的下一层，如果输入一个 10×16 数据，你也可以使用一个 5 维过滤器进行卷积，这需要 16 个通道进行匹配，如果你有 32 个过滤器，另一层的输出结果就是 6×32，如果你使用了 32 个过滤器的话。



对于 2D 数据而言，当你处理 10×10×16 的数据时也是类似的，你可以使用 5×5×16 进行卷积，其中两个通道数 16 要相匹配，你将得到一个 6×6 的输出，如果你用的是 32 过滤器，输出结果就是 6×6×32，这也是 32 的来源。

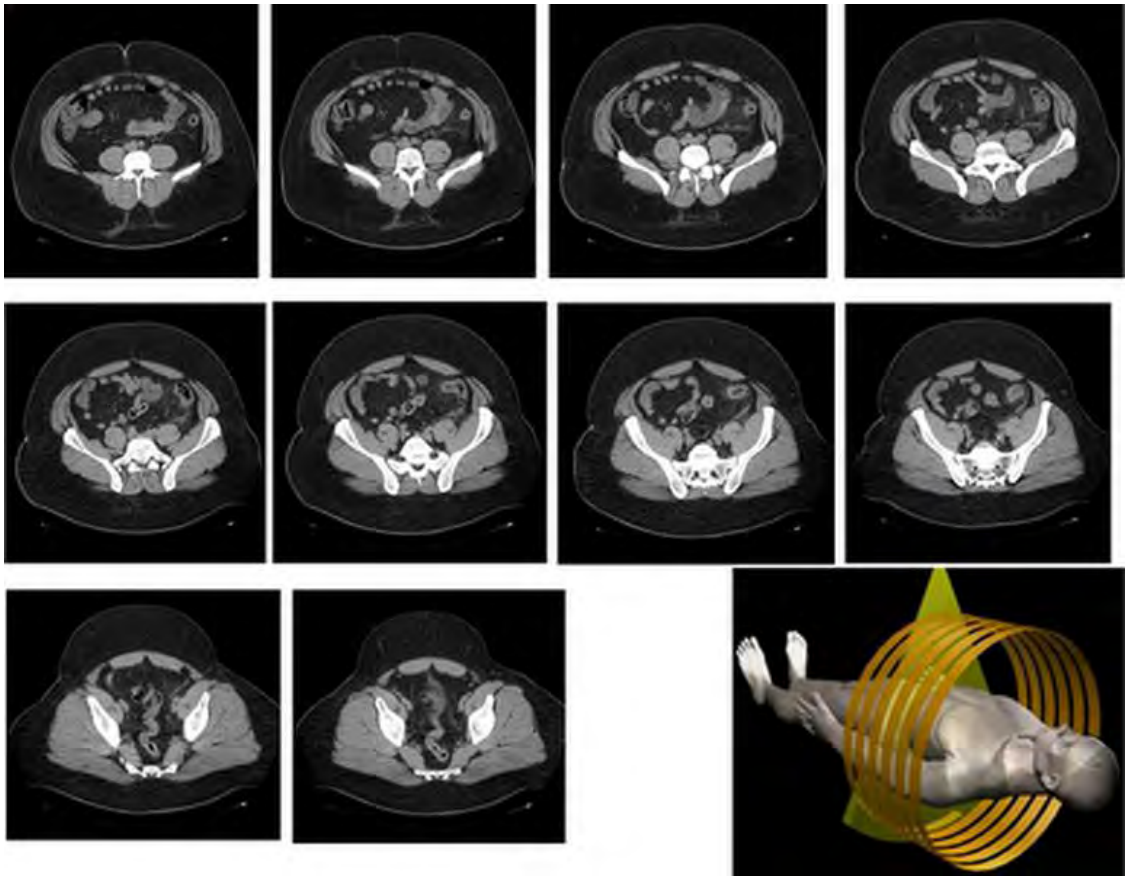
所有这些方法也可以应用于 1 维数据，你可以在不同的位置使用相同的特征检测器，比如说，为了区分 **EKG** 信号中的心跳的差异，你可以在不同的时间轴位置使用同样的特征来检测心跳。

所以卷积网络同样可以被用于 1D 数据，对于许多 1 维数据应用，你实际上会使用递归神经网络进行处理，这个网络你会在下一个课程中学到，但是有些人依旧愿意尝试使用卷积

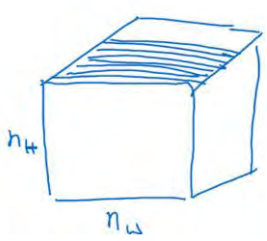
网络解决这些问题。

下一门课将讨论序列模型，包括递归神经网络、**LCM** 与其他类似模型。我们将探讨使用 **1D** 卷积网络的优缺点，对比于其它专门为序列数据而精心设计的模型。

这也是 **2D** 向 **1D** 的进化，对于 3D 数据来说如何呢？什么是 **3D** 数据？与 **1D** 数列或数字矩阵不同，你现在有了一个 **3D** 块，一个 **3D** 输入数据。以你做 **CT** 扫描为例，这是一种使用 X 光照射，然后输出身体的 **3D** 模型，**CT** 扫描实现的是它可以获取你身体不同片段（图片信息）。



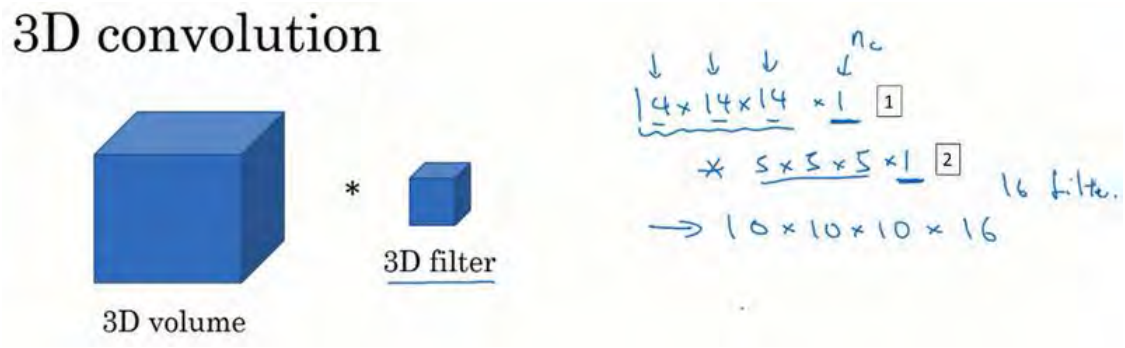
当你进行 **CT** 扫描时，与我现在做的事情一样，你可以看到人体躯干的不同切片（整理者注：图中所示为人体躯干中不同层的切片，附 **CT** 扫描示意图，图片源于互联网），本质上这个数据是 3 维的。



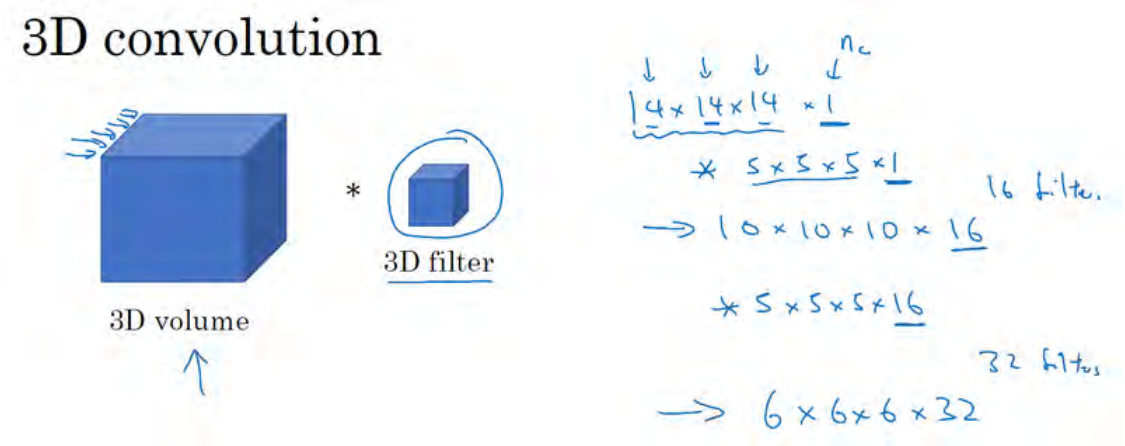
一种对这份数据的理解方式是，假设你的数据现在具备一定长度、宽度与高度，其中每

一个切片都与躯干的切片对应。

如果你想要在 **3D** 扫描或 **CT** 扫描中应用卷积网络进行特征识别，你也可以从第一张幻灯片（**Convolutions in 2D and 1D**）里得到想法，并将其应用到 **3D** 卷积中。为了简单起见，如果你有一个 **3D** 对象，比如说是 14×14×14，这也是输入 CT 扫描的宽度与深度（后两个 14）。再次提醒，正如图像不是必须以矩形呈现，**3D** 对象也不是一定是一个完美立方体，所以长和宽可以不一样，同样 CT 扫描结果的长宽高也可以是不一致的。为了简化讨论，我仅使用 14×14×14 为例。



如果你现在使用 5×5×5 过滤器进行卷积，你的过滤器现在也是 **3D** 的，这将会给你一个 10×10×10 的结果输出，技术上来说你也可以再×1（编号 1 所示），如果这有一个 1 的通道。这仅仅是一个 3D 模块，但是你的数据可以有不同数目的通道，那种情况下也是乘 1（编号 2 所示），因为通道的数目必须与过滤器匹配。如果你使用 16 过滤器处理 5×5×5×1，接下来的输出将是 10×10×10×16，这将成为你 3D 数据卷积网络上的一层。



如果下一层卷积使用 5×5×5×16 维度的过滤器再次卷积，通道数目也与往常一样匹配，如果你有 32 个过滤器，操作也与之前相同，最终你得到一个 6×6×6×32 的输出。

某种程度上 **3D** 数据也可以使用 **3D** 卷积网络学习，这些过滤器实现的功能正是通过你的 3D 数据进行特征检测。**CT** 医疗扫描是 **3D** 数据的一个实例，另一个数据处理的例子是你

可以将电影中随时间变化的不同视频切片看作是 **3D** 数据，你可以将这个技术用于检测动作及人物行为。

总而言之这就是 **1D**、**2D** 及 **3D** 数据处理，图像数据无处不在，以至于大多数卷积网络都是基于图像上的 **2D** 数据，但我希望其他模型同样会对你有帮助。

这是本周最后一次视频，也是最后一次关于卷积神经网络的课程，你已经学习了许多关于卷积网络的知识，我希望你能够在未来工作中发现许多思想对你有所裨益，祝贺你完成了这些视频学习，我希望你能喜欢这周的课后练习，接下来关于顺序模型的课程我们不见不散。

附件

榜样的力量-吴恩达采访人工智能大师实录

吴恩达采访 Geoffery Hinton

Geoffery Hinton 主要观点：要阅读文献，但不要读太多，绝对不要停止编程。

Geoffrey Hinton：谢谢你的邀请

吴恩达：我想你是至今为止整个地球上发明最多深度学习核心理念的人，很多人都称呼你为“深度学习教父”，尽管我是直到和你聊了几分钟之后才发现我是第一个这样称呼你的人，对此我深感荣幸不过我想问的是，许多人把你看作传奇，我更想知道一些传奇背后的私人故事，所以你是怎样在很久之前就投身于人工智能，机器学习以及神经网络之中的呢？

Geoffrey Hinton：当我还在高中时有一个什么都比我强的同学，他是个才华横溢的数学家，有天他来学校并且问我，你知道大脑是用全息图运作的吗？那时应该是 1966 年，我回答他 全息图是个啥？他就解释了一下，在全息图中，你可以切掉它的一半，但依然了解得到全貌，还有大脑中的记忆可能是分布于整个大脑中的，我大概猜到他可能是读过关于 Karl Lashley 的实验，其中讲到切掉老鼠几个小部分的脑子，然后发现很难找到哪一部分存储哪种特别的记忆，那是第一次让我对大脑，怎么储存记忆产生兴趣的时刻，然后当我去上大学的时候，我就开始学习生理学和物理学，当我在剑桥的时候我是唯一一个在学生理学和物理学的本科生，之后我放弃了这个选择并且尝试学哲学，因为我觉得那可能会给我更多的深入了解，但是后来我又觉得缺乏真正能够辨别错误说法的方法，然后我就转去学了心理学而在心理学中有着非常非常过于简单的理论，对我个人来说用来解释大脑的运作看起来无可救药的不充分，之后我花了点时间做一个木匠，然后我又决定想去试试看人工智能，于是就跑去爱丁堡跟 Longuet Higgins 学人工智能，他已经做了很棒的关于神经网络的研究并且刚刚决定放弃于此，转而对 Terry Winograd 的学说表示赞赏，我刚去的时候他觉得我做这个（神经网络）已经过时了，应该开始搞符号主义人工智能，关于这个我们有很多争论，但我还是坚持做自己相信的事情，然后呢？最终我拿到了人工智能博士学位。 但我在英国找不到工作，但我注意到了一个很不错的加州 Sloan 奖学金的广告，我拿到了这个奖学金，我去了加州，那里感觉很不一样。在英国“神经网络”看上去很愚蠢，而在加州 Don Norman 和 David Rumelhart 对于神经网络的观念非常开放，在那里我第一次开始考虑大脑是怎么运作的，和

心理学会有什么联系，看上去是一个积极的方向，这其中有很多乐趣，特别是和 David Rumelhart 一起工作相当棒。

吴恩达：我懂了，很好，1982 年你在 UCSD 和 Rumelhart 在一起工作，最后写出了反向传播的论文，事实上，比这还要复杂点，什么情况呢？

Geoffrey Hinton：大概在 1982 年初 David Rumelhart 和我，还有 Ron Williams 我们开发了反向传播算法，这主要是 David Rumelhart 的想法，我们发现不少人都发明过 David Parker 发明过，可能比我们晚，但发表比我们早，Paul Werbos 发表了好几年的，不过没人注意到还有其他人也做出了类似的算法，但不能清晰地表述出“反向传播”的含义，用链式法则求导并不是很新的想法。

吴恩达：明白，为什么你会认为你们的论文极大地帮助大家理解了“反向传播”呢？似乎你们的论文被认为是让大家去接受这个算法一个里程碑式的影响。

Geoffrey Hinton：最后我们的论文上了《自然》，为了论文被接受，我做了不少人事工作，我想到其中一个审稿人很可能会是 Stuart Sutherland 英国一位很有名的心理学家，我跑去和他聊了很久，跟他解释这到底是怎么一回事，给他留下了很深刻的印象，因为我们给他展示了反向传播法可以学习字元表示，你可以看到这些表示都是一些向量，你可以理解每一个特征的意义，实际上我们训练了三个名字的家族树模型，比如 Mary 的妈妈是 Victoria，你给出前面两个名字，它能预测到最后的名字，训练过后你可以看到每一个名字的所有特征，比如某个人的国籍，是第几代，在家族树中的哪一旁枝等等，这震惊了 Stuart Sutherland，我想，这是论文被通过的原因。

吴恩达：非常早期的放弃词向量，并且你已经在接触在训练算法中出现的能被学习的语义特征了。

Geoffrey Hinton：是的，所以从一个心理学家的角度来说，真正有趣的是它能把两股完全不同的知识概念统一起来曾有一些心理学家认为，知识概念是一大束特征，对此也有很多相关证据，之后又出现了现代 AI 的观点，也就是正式的结构主义观点，意思即是任何一个概念都有许多其他概念与其相关，为了理解一个概念，你会需要一个图形结构或是一个语义网络，然后这个后向传播的例子展示了你可以把信息传给它，信息会变成图形结构，在这个特殊情况是一个族谱，然后输入信息会用这样的方式变成特征，然后它能够使用这些特征来得到新的前后一致的信息，也就是归纳总结化。但是真正重要的是这样来回往复的图形或树形的表现族谱方式，把里面的人变成特征向量这种形式，事实上从图形状的表现手法，你也能够得到特征向量，从特征向量你又可以得到更多的图形陈述，那时候是 1986 年。90 年代

早期 Bengio 就已经展示过，你可以把真实的数据，比如你可以拿来英文文本，然后使用这些技巧得到文本的词语向量，这曾经惊艳了许多人。

吴恩达：最近我们在聊的都是计算机变得有多快，比如使深度学习不断提高的新的 GPU 和超级计算机，我在 90 年代到 1986 年那时还不存在的这些工具，听起来你和 Bengio 那时就已经开始引发了这个潮流。

Geoffrey Hinton：是的，在当时的确是很大的进步。在 1986 年，我曾经用一个速度小于十分之一秒百万次浮点运算的列表机，大概 1993 年左右，人们才逐渐开始见到十秒百万次浮点运算，所以曾经都是 100 的倍数，在那时也许还挺好用的。因为计算机才刚刚开始变快。

吴恩达：原来如此，前几十年的时候，你就已经发明了这么多神经网络和深度学习相关的概念，我其实很好奇，在这么多你发明的东西中，哪些是你到现在为止依然还是保持热情的。

Geoffrey Hinton：我认为最具学术之美的是我和 Terry Sejnowski 做的 Boltzmann 机器，我们发现它能用非常非常简单的学习算法去应用到密度很高的连接起来的网络，这些网络中只有一些节点能被看到，那么隐藏表示方式，能够用非常简单的算法学习，看起来也是一种你应该能够应用大脑的东西，因为只需要知道直接和每一个突触相连接的两个神经元所被传播的信息应该都是一样的，这里有两个不同的时期，我们也可以称为唤醒和睡眠，但是在这两个不同时期传播信息的方法都是一样的，不像在后向传播中有前后向两种，方法不同，因为发送不同种类的信号，这就是我觉得它的魅力所在，很多年以来都看似只是好奇心所向，因为运行速度很慢，不过后来，我去繁为简，开始只在简单些的网络用一个循环，于是就有了受限 Boltzmann 机，实际中反而更有效，在 Netflix 的比赛中，举个例子，受限 Boltzmann 机是第一名所用的算法之一。

吴恩达：事实上，很多最近复苏的神经网络，深度学习，从 2007 年开始，受限 Boltzmann 机和解除限制 Boltzmann 机，你和你实验室做了很多贡献。

Geoffrey Hinton：这是另外让我做得很开心的工作，你能训练受限 Boltamann 机的想法，仅用一层的隐藏特征，只学一层特征，然后你可以重复把特征当成数据，然后再把新的特征当成数据，再重复，直到你想要的次数，实际能够应用，的确很不错，然后 Uy Tay（音）发现这全部过程，可以想成是一个模型，虽然有点奇怪，这个模型顶部是一个受限 Boltzmann 机，往下是个 s 形置信网，这发明很超前，这是个针对模型，并且我们想要，能训练这些受限 Boltzmann 机，能有效地适用于 s 形置信网，那时，已经有人在做神经网络了，在用高密

度连接的网络，但是没有足够好的概率标记，也有人在做图模型，不像我的孩子们能做合适推理，但是也只能用在稀疏连接的网络，我们展示了训练深层置信网络的方法，使大致上的推理变得非常快，只需要一个前向推进，结果就能很美丽，你可以保证，每次学一层新的特征，都有新的带，每次都是这样，每个新的带都会比旧的好。

吴恩达：变分带显示你是否新加了层数，对，我记得那个视频。

Geoffrey Hinton：这也是第二件我始终很感兴趣的东西，第三个是做过的一些变分法，统计学者们也做过类似的东西，不过那时我们并不知道，我们让 EN 变得更有效，通过证明你不需完美的 E 步骤，而只需要个大约的，EN 当时在统计学很有分量，我们证明了它能被一统化，特别是 1993 年 和 Van Camp，我写了篇论文，是第一篇变分贝叶斯的论文，并且证明了可以只用一个版本的贝叶斯学习，更容易处理，因为能用 a 来估算真正的后验概率，也可以用在神经网络中，我对此非常兴奋。

吴恩达：原来如此，哇，真牛。对，我记得提到的所有论文，你和 Hinton（口误）在论文上花了很多时间，我认为一些现在用的算法或大多数人几乎每天用的一些算法，比如 dropout 或来源于你团队的激活函数。

Geoffrey Hinton：对但不完全是，那么其他人可能也做过 ReLU，我们在受限 Boltzmann 机上花功夫证明了 ReLU 几乎完全等同于一叠 logistic 单元，这是其中之一推进 ReLU 前进的力量。

吴恩达：我对此非常好奇，这论文的价值是，用了大量数学证明此函数能被复杂公式估算，你是为了发表论文做的数学，还是真的为了影响 0 和 x 的最大值的发展。

Geoffrey Hinton：并不是为了发论文的情况，数学对推进这个想法，真的很有影响力，我显然已经知道 ReLU 还有 logistic 单元，由于我花了心血在 Boltzmann 机上，全都是用的 logistic 单元，那时候面临的问题是，这个学习算法可能用在 ReLU 吗，证明完 ReLU 几乎等同一叠 logistic 单元后，我们展示了所有的数学证明。

吴恩达：原来如此，它为无数现在使用 ReLU 的人们提供了灵感，也不需要特别懂得每个细节。

Geoffrey Hinton：对，那么其中之一的发现在我到 Google 之后，是 2014 年我在 Google 讲 ReLU 的用法以及怎么用单位矩阵初始化，因为 ReLU 的一大优点是，如果不断复制隐藏层，又用单位矩阵初始化，它会复制下层的模式，我展示了你可以训练一个 300 个隐层的网络，并且用单位矩阵初始化，会很有效率，但是我没有继续研究下去，也很是后悔，我们只发了一篇论文，能证明你可以初始化，可以初始化重复发生，我应该继续研究的，后来就剩

下来了。

吴恩达：很多年以来我都听到你谈论大脑，我听到过你谈论后向传播和大脑的关系，现在你对此是什么想法。

Geoffrey Hinton：我现在正好有论文在准备中，主要想法就是这个，如果后向传播是个好的学习算法，那进化过程肯定会从中干扰，有些细胞会变成眼球或是牙齿，如果细胞能做这些，就一定能应用后向传播了，这样假设会造成极大的选择性压力，所以我觉得，毫不怀疑这种想法的神经科学家有点犯傻，可能有比较微妙的应用，我想大脑可能，不会完全用后向传播，但是足够相似，这么多年以来我想出很多可能的理论，在 1987 年我和 Jay McClelland 做出了循环算法，核心想法是你发送信息并循环，并在循环之中保持，它所包括的信息，最简单的版本是，你有输入和隐藏单元，然后你从输入发信息到隐藏单元，再回到输入，再到隐藏，再回到输入，以此类推，那么你想要 训练一个自动译码器，但是你想绕开后向传播，你只要训练，并且去掉所产生的变化，核心想法是神经元突触的学习方式，通过改变突破前的输入权重比重，同时按比例改变突破后的输入，在再流通时，你需要突破后的输入，旧的比较好，新的较差，这是比较理想的情况，我们在神经科学家发明尖峰时序依赖可塑性之前，发明了这个算法，尖峰时序依赖可塑性，是个相同的算法，但是，相反，需要新的东西较好，旧的较差，所以，用预先设定前景活动的权重比例，乘以新的，再减去旧的，后来我在 2007 年发掘，如果你用一叠，受限 Boltzmann 机来训练，训练完，你会有完全正确的条件，来重建并实现后向传播，如果你关注重建时期，重建时期实际会告诉你偏差表现的导数，2007 年的第一个深度学习专题讨论中，我曾做过相关演讲，那时几乎被完全遗漏的部分，后来 Joshua Bengio 重拾这个想法，下了很多功夫，我自己本人也做了很多相关研究，如果你有一叠自动译码器，就可以通过后向传送活动和重建点定位，得到导数，这个有趣的想法也是大脑工作的原理之一。

吴恩达：另一个据我所知，你正在研究的，怎么在深度学习中解决多个时间技巧，能分享一些你的想法吗？

Geoffrey Hinton：没问题，这要回溯到我研究生第一年，我第一次展示了怎么用快速权重，也就是适应很快，衰退也很快的权重，所以只有短期记忆，我在 1973 年展示了一个很简单的系统，可以用这些权重做真实循环，真实循环的意思是，用到的神经元来表示循环核心的东西用到的权重，实际也是在循环核心被重复利用的知识引导出的新问题是，当你突出循环核心时，怎么记得已经是在过程中了，这个记忆从何而来，因为你用过了重复核心的神经元，答案是你可以把记忆放入快速权重中，这些活动神经元可以从快速权重中复原和

Jimmy Ba 最近的研究中，我们已经有了篇关于，这样来用快速权重复原的论文，这个空缺非常大，1973 年的第一个模型没有被发表，接下来就是 2015 或 2016 年 Jimmy Ba 的模型，前后相隔 40 年。

吴恩达：是个有些年头的概念，五年左右，叫做胶囊，你目前进展如何？

Geoffrey Hinton：我回到之前所在的状态，就是我非常笃定地相信，所有人都不信的东西，我提交了一些论文都会被拒接但是我真的相信这个想法，我也会继续研究，在转移中有一些很重要的概念，其一是如何表示多维个体，你可以用些借壳活动，表示多维个体，只要你知道其中的任何一个，在图片中任何一个区域，你会假设至少有一个特别的特征，之后你会用一堆神经元，以及它们的活动来表示特征的不同方面，比如x, y坐标具体在哪，在哪个方向，移动速度是多快，是什么颜色，什么亮度，类似等等，你可以用一堆神经元，来代表不同维度的同一个东西，假如只有其中一个，这种做法很不一样，相比于普通方式，普通来说在神经网络中就只有一个很宏观的层，所有的单元和作用都在里面，但是你不会想到把它们结合成一个个小组，来用不同的坐标表示相同的东西，我们应该去掉多余的结构，另一个想法是。

吴恩达：在真实的表示方法中，再分段表示，对不同的子集，去表示。

Geoffrey Hinton：我把这些子集叫做胶囊，胶囊是能够表示一个特征的，一个并只有一个的情况，它能表示特征的所有不同性质，胶囊比一般的神经元能表示更多性质，或是一个普通的神经网络，因为只能表示一个度量上的属性。

吴恩达：原来是这样。

Geoffrey Hinton：当你能这么做之后，你还能做到普通神经网络表现很差的是，我称为常规协议，假设你想要分层，然后图片中有一张嘴，一个鼻子或是别的东西，你想知道你该不该尝试组合成一个东西，这个想法就可以用胶囊。比如一张嘴，有关于嘴的参数，还有鼻子的胶囊。也有关于鼻子的参数，再确定该不该把它们拼起来，你会有来决定能不能拼成脸的参数，要是嘴鼻子有对的空间关系，它们就会对应上，在统一层面有两个相应的胶囊，能组成到下一个层面，就可以决定应该组合起来，因为在高维度能对应上很难得，这过滤方法很不同寻常，相比普通情况的神经网络，常规协议会变得至关重要，特别是用有限数据总结概括时，这个观念上的改变会帮助完成分段，我希望统计角度也更有效，对比我们现在的神经网络，如果你想要改变观点，你就得尝试做，并且全都训练一遍。

吴恩达：好的，相比 FIFO 学习，监督学习，还可以做点不一样的。

Geoffrey Hinton：我还是计划做监督学习，但是前向路线会很不一样，不同之处在于里面还有些循环过程，比如你觉得找到个嘴，找到个鼻子，之后用几个循环去决定，它们能不能组成一张脸，用这个循环可以做后向传播，也可以有偏重地试试看，我在多伦多的小组正在研究这个，在多伦多我有一个 Google 小组，是 Brain 组的一部分，现在我对此非常兴奋。

吴恩达：的确很棒，很期待论文的发表。

Geoffrey Hinton：哈哈对，假设能发表的话。

吴恩达：你研究深度学习都几十年了，我很好奇你的想法，关于 AI 有什么改变吗？

Geoffrey Hinton：我用了很多时间，在后向传播上，比如怎么使用，怎么实现它的力量，刚开始，80 年代中时，我们在做偏重学习，结果很不错，然后到 90 年代早期，大多数人类学习都，应该是无监督学习，然后我对无监督学习产生浓厚兴趣，开始研究 Wegstein 算法。

吴恩达：那时候你的探讨对我个人影响也很大，我在领导 Google Brain 团队时，第一个项目就在你的影响下研究了好多无监督学习。

Geoffrey Hinton：是呢，我可能误导你啦，因为长期来说，无监督学习绝对会变得很重要，但是面对现实，近十年以来所做的都是监督学习，侧重学习都有标签，你想预测一个系列的下一个东西，也就是标签，结果惊人的好，我仍相信无监督学习会很重要，当我们真正搞明白一些东西以后，结果会比现在好得多，不过目前还没到。

吴恩达：嗯，深度学习里的高级研究人员，包括我自己对此依然很是激动，因为我们中没有一个人知道该怎么办，可能你知道，但是我不行了。

Geoffrey Hinton：变分法改变代码是你会用到更新参数化的地方，这想法看起来很不错，

生成对抗网络也是很棒的想法，生成对抗网络我认为是深度学习中最新最重要的想法，我希望胶囊也能这么成功，生成对抗网络目前是很大的突破。

吴恩达：稀疏，缓慢的特征上有发生什么吗？其他两个无监督建模的原则是什么。

Geoffrey Hinton：我从未像你一样看重稀疏性，但是慢速特征，我认为，是个错误，不应该说慢，基本想法是对的，但不应该只考虑不变的特征，而是可预测改变的特征，建任何模型的基本原则大概是，先记录测量，对其应用非线性变换，直到状态向量成为表达式，这项活动变得线性时，不能像做普通渗透一样，假设线性应该找一个从观察转换，到潜在变量的转换，线性操作，比如潜在变量的矩阵乘积。即是如此，举个例子：如果你想改变观点，如果你想从另一个观点产生图像，需要从像素转化到坐标，当你有坐标表示后，希望胶囊也能做到如此，你就可以做矩阵乘数来改变观点，再投射到像素中。

吴恩达：这就是为什么这是个非常宏观的原则，也是你做面部人工合成的原因，对吗，即是把脸压缩成低维度向量，再生成其他的脸。

Geoffrey Hinton：我曾有个学生研究这个，我自己本人没怎么做。

吴恩达：你应该常常被问到，如果有人想要入门深度学习，该做什么，你有什么建议吗？之前应该有很多一对一的情况，但对于全球范围都在看这个视频的观众，对于要学深度学习的人们，你有什么样的建议。

Geoffrey Hinton：好，我的建议是多读论文，但别读太多，我从导师那里得到这个建议，很不想大多数人说的，大多数会告诉你尽量多读，然后开始自己的研究，对一些研究人员应该是正确的，但是对有创意的人应该读一少部分，然后发现一点你认为所有人都错了的东西，在这点我一般都逆着来，你看到它，感觉不太对，然后想怎么才能做对，当人们反对你时，要坚持自我，我支持人们坚持自我的原则，是判断直觉的对错，你直觉还不错的话，就该坚持，最后一定会成功，要是你直觉一般的话，做啥都无所谓。

吴恩达：我懂了哈哈，鼓舞人心的建议 就坚持下去。

Geoffrey Hinton：该相信直觉，不相信就没意义了

吴恩达：我懂了，是的。我通常建议人们不要只看，而是要重现发表了的论文，自然而然限制了你做的数量，因为重现结果很耗时间。

Geoffrey Hinton：对，当你重现发表的论文时，会发现所有使之成功的小技巧；另一个建议是，永远不要停止编程，因为如果你给学生布置任务，他们三天打鱼两天晒网，回头就会告诉你看，没做成，没做成的原因，往往是他们所做的小决定，当时不觉得很重要，举个例子，如果你给一个好学生任务，你可以给他们任何人任务，他都会做成，我记得曾经有一

次，我说。诶等等，我们上次讨论时，因为某某原因，是不可能成功呀，学生回答说：“对呀，你说完我就发现了，就假设不是你真的意思”。

吴恩达：哈哈原来这样，那很厉害，还有其他关于 AI 和深度学习的建议吗？

Geoffrey Hinton：我认为基本上，开始锻炼直觉时要读够，然后相信直觉，自己动手，不要担心别人有反对意见。

吴恩达：你也没法证明，他们说的对不对，除非亲自做了，才能知道

Geoffrey Hinton：对，但还有一点，如果你有个绝好的想法，别人都觉得完全荒谬，那你就找对东西了，举个例子，当我刚想出来变分法时，我给之前一个叫 Peter Brown 的学生写了封信，他懂得很多 EN 相关知识，他就拿去给一起工作的人看，名字叫俩兄弟，可能是双胞胎吧，然后他说，俩兄弟说了，你要么是你喝多了，要么是傻，俩兄弟当真认为是荒谬之论，部分原因可能是我用的解释方式，因为我只解释了直觉，但当你有个很不错的想法时，其他人觉得完全是垃圾，就是个好想法的信号了。

吴恩达：好的，研究课题的话，新研究生们应该多研究胶囊，无监督学习，还有其他吗？

Geoffrey Hinton：对新研究生的一个好建议是，找一个和你意见一致的导师，因为如果你做的东西，导师也深深赞同，你会得到很好的建议，要是做你导师不感兴趣的东西，你会得到没啥用的建议。

吴恩达：好的，最后对于学习者的建议，有什么建议给想攻读博士的人，或去顶级公司工作，或顶级研究团队。

Geoffrey Hinton：这问题复杂。我认为现在，没有足够的深度学习学者在大学中教育有求知欲的人，就没有足够的教职人员，不过应该是暂时性的，发生的事是这样，大多数部门很少有，真正懂得这场革命的人，我几乎同意，这并不是二次工业革命，但是规模接近，有如此巨大的改变，基本是因为我们和计算机的关系改变，不再只是编程序，而是让它们有能力自动解决问题，从根本上改变了计算机的用法，计算机科学部门，却是在之前基础上建立起来的，他们暂且不懂，训练计算机会和编程一样重要，部门中一半的人，得实际去试过训练计算机，我所在的部门就拒绝承认，应该放手让很多人去做，他们觉得有几个就够，可不能放太多人去，这种情况下，你就得建议大公司多花点时间训练员工，Google 培养的人们，我们叫做 brain 居民，我很怀疑最后大学们能赶上。

吴恩达：嗯，没错，实际上，能有很多学生都发现了，前 50 的学校超过一半的申请者实际，只想训练计算机而不是真正做编程，是，归因齐宗，深度学习 AI 的微专业课程，据我所知，最初都是，你在 Coursera 上教授的，还得回溯到 2012 年，奇怪的是，也是你发表

RMS 算法的时候，也还挺难的。

Geoffrey Hinton：你邀请我去做，大型开放网课，我可是很怀疑的，但你一直逼我，我很庆幸我那么做了，尽管工作量极大。

吴恩达：是的，感谢你做到了，我还记得你发牢骚，说要做的这么多，你还老是熬夜，但我觉得很多很多的学者都受益于你的课程，我也非常感激你真的做到了。

Geoffrey Hinton：很棒，对。

吴恩达：这些年来，我也目睹你被卷入 AI 界模范的辩论中，以及是否有过 AI 的变更，你可以分享一下你的想法吗？

Geoffrey Hinton：我很乐意，早期时，大概 50 年代，像 Von Neumann 之类的人，都不相信符号化 AI，他们灵感更多来源于大脑，不幸的是，他们都英年早逝，未能使自己的想法面世，AI 早期时，人们完全被说服，智力的表示该是，某种符号表达，比较整洁的逻辑，而不完全是逻辑，但是类似逻辑，智力的本质是推理，现在发生的是，有种完全不同的观点，就是无论想法是什么，都是一个很大的神经活动向量，对比符号化的表示，我认为那些把想法想成是符号表达的人，大错特错，输入是一串字符，输出是一串单词，因此，单词的字符串显然能作为表达式，他们觉得一定是字符串，或其他类似字符串的东西，我却不认为是这样，我觉得想法该是某种语言，简直和，把想法置于空间层面理解，必须得是像素进，像素出，一样傻，要是我们能与点矩阵打印机相连，那像素就会出来，但中间的并不是像素，所以我认为想法该是一个大向量，有因果能力的大向量，能引发出其他大向量，这于 AI 的标准观点，符号化表达完全不同。

吴恩达：好的，AI 是肯定会改变，到新的视角的。

Geoffrey Hinton：起码一部分，大多数人还是认为，AI 是符号式的

吴恩达：非常感谢你愿意接受采访，能听到深度学习进化的全过程很棒，还有你依旧在带领它前进，很感谢你 Geoff。

Geoffrey Hinton：感谢你给我这个机会，谢谢你。

吴恩达采访 Ian Goodfellow

吴恩达：嗨，Ian，感谢你今天接受采访。

Ian：谢谢你邀请我，Andrew，我很高兴来到这里。

吴恩达：今天你也许是世界上最知名的深度学习研究员之一，让我们来听听你的心路历程吧，你是怎样一步步进入这行的呢?

Ian：好，是个好想法，我想我是在遇到你之后才开始对机器学习感兴趣的，我一直从事神经科学研究，我的本科时代导师斯坦福大学的 Jerry Cain 鼓励我去上你的 AI 课。

吴恩达：啊，这我不知道呢。

Ian：好，所以我一直觉得 AI 是个好想法，但在实践中，主要的，我想主要达到实用的是一些比如游戏 AI，有很多人工编码的规则，让游戏中的非玩家角色，在不同的时间点说出不同的脚本对话，然后，当我在上你的 AI 入门课程时，你讲到的话题，线性回归和线性回归，误差的偏差和方差分解，我开始意识到这可以是真正的科学，我实际上可以，在 AI 领域从事科学研究而不是神经科学。

吴恩达：我知道了，很好，那接下来呢?

Ian：然后我就回来当你的课程助教了呀。

吴恩达：原来如此，当我的助教。

Ian：其实我人生的一大转折点，就是在当那门课助教的时候，其中一名学生，我的朋友

Ethan Dreifuss 对 Geoff Hinton 的深度信念网络论文很感兴趣。

吴恩达：我知道了…

Ian：是我们两个最后一起在斯坦福大学搭建了世界第一台 GPU CUDA 机器,专门用来跑玻尔兹曼机,就在那年寒假的业余时间里,我知道了,在那时候,我开始有一个非常强烈的直觉,深度学习才是未来,那时我们接触了很多其他算法,比如支持向量机,它们的渐近线不太靠谱,当你输入更多训练数据时,它们却在变慢,或者对于相同数量的训练数据,改变其他设定并没有改善它们的表现,从那开始,我就专注于深度学习了。

吴恩达：我记得 Richard Reyna 有一篇很老的 GPU 文章,提到你做了很多早期的工作。

Ian：是的，是的，那篇文章用的就是我们搭建的一些机器，是的，我建造的第一台机器就是 Ethan 和我建立的机器，用我们自己的钱在 Ethan 妈妈的房子里搭的，之后，我们用实验室经费在斯坦福实验室搭了两三台。

吴恩达：哇，太神奇了，我还不知道这事，太好了。那么，今天真正以风暴席卷深度学习世界的，是你发明的生成式对抗网络(GAN)，那么你怎么想出来的呢?我一直在研究生成模型很久，所以 GAN 是其中一种生成模型，你有很多训练数据，你希望学会制造更多类似它们的数据，但它们都是虚构的，网络还没见过这种形式的虚构数据，还有几种办法可以做生成模型，在我们想出 GAN 之前还流行了几年，在我读博的时候，我一直在研究其他各种方法，我非常了解所有其他框架的优缺点，玻尔兹曼机器和稀疏编码，还有其他多年来一直很受欢迎的方法，我那时正在寻找某个可以同时避免所有方法缺点的东西，最后，当我在一个酒吧里和朋友争论生成模型的时候，灵感来了，我开始告诉他们，你需要这么做，这么，这么做我保证管用，我的朋友不相信会管用，我本来还在写一本深度学习的教科书。

吴恩达：我知道了..

Ian：但是我强烈相信，这个想法是靠谱的，我马上回家，当天晚上就调试成功了。

吴恩达：所以只花了一晚上就调试成功第一个 GAN 了?

Ian：我大概在午夜做完的，就从我朋友在酒吧的离职派对离开回到家之后。

吴恩达：我知道了..

Ian：而它的第一个版本是有效的，这是非常非常幸运的，我没有搜索超参数或任何东西。

吴恩达：我在某个地方读过一个故事，在那里你有一次涉死体验，让你对 AI 的信仰更坚定了，给我讲讲那个故事。

Ian：我其实没有涉死啦，但有那么一瞬间我觉得要死了，我头很痛很痛，一些医生认为我可能有脑出血，在我等待我的 MRI 结果，看看有没有脑出血时，我意识到，我的大部分想法都是，要确定有其他人继续尝试我当时的研究思路。

吴恩达：我懂了，我懂了。

Ian：回想起来，那些都是非常愚蠢的研究思路。

吴恩达：我懂了..

Ian：但在这一点上，我意识到这实际上是我生活中优先级最高的事，就是做机器学习研究工作。

吴恩达：我懂了，是啊，那很棒，当你以为你快要死的时候，你还是想如何完成研究。

Ian：是的。

吴恩达：是，这真的是信仰。

Ian：是的。

吴恩达：是啊，是啊，所以今天你仍然处于 GAN 研究的风暴中心，就是这个生成性对抗网络，可以告诉我怎么看 GAN 的未来吗？

Ian：现在 GANs 应用在很多场合里，比如半监督学习，生成其他模型的训练数据，甚至模拟科学实验，原则上这些东西都可以用其他生成模型来做，所以我认为 GAN 现在在一个重要的十字路口，现在它们有时候效果很好，但要把它们的潜力真正发挥出来，更像是艺术而不是科学，10 年前人们对深度学习的感觉或多或少也是如此，当时我们正在使用，以波尔兹曼机器为基础的深层信念网络。它们他们非常非常挑剔。随着时间的推移，我们切换到修正线性单元和批量归一化，深入学习变得更加可靠，如果我们可以把 GAN 变得像深度学习一样可靠，那么我想我们会继续看到 GAN，在今天它们的应用领域里获得更大的成功，如果我们弄不清楚如何稳定 GAN，那么我想它对深度学习历史的贡献，就是它向人们展示了如何完成这些涉及到生成模型的全部任务，最终，我们将用其他形式的生成模型来代替它们，所以我花了大约 40%的时间试图稳定 GAN。

吴恩达：我懂了，很酷，我想就像很多人大约 10 年前进入深度学校领域一样，比如你自己，最后变成了领域的先驱者，也许今天加入 GAN 的人，如果它确实管用，那么最后可能也会成为先驱。

Ian：是啊，很多人已经是 GAN 的早期先驱，如果你想描述 GAN 的历史，你真的需要提到 比如 Indico 等其他组织，还有 Facebook 和伯克利，那些小组完成的各种工作。

吴恩达：所以除了你的研究，你还合著了一本关于深度学习的书，可以说说吗?

Ian：没错，我和 Yoshua Bengio 和 Aaron Courville 合著的，他们是我的博士导师，我们写了一本现代版深度学习教科书，一直很受欢迎，英文版和中文版都很受欢迎，我们已经售出了我想两种语言加起来有 70000 本吧，而且我从学生那里得到了很多反馈他们说获益良多，我们有件事做得和其他教材不同，我们一开始就介绍深度学习需要用到的数学知识，我从斯坦福大学课程中得到的一件事是，线性代数和概率论非常重要，人们对机器学习算法感到兴奋，但如果你想成为一名非常优秀的从业者，你必须掌握基本数学，这是整个算法的基础，所以我们确保一开始，集中讨论需要的数学基础，这样，你就不需要重头开始学习线性代数，你可以得到一个短期训练课程，了解对深度学习最有用的线性代数。

吴恩达：所以即使有些学生数学基础不好或者有几年没接触过数学，你就可以从教材的

开始，学到进入深度学习的所有背景知识。

Ian：你需要知道的所有事实都在那里，当然，你肯定需要集中精神去掌握其中一些概念。

吴恩达：是的，是的，很好。

Ian：如果有人真的害怕数学，这经历可能还会有点痛苦，但如果你准备好去学习的话，我相信你一定可以掌握的，你们需要的所有工具都在哪了。

吴恩达：作为在深度学习领域工作了很长时间的研究员，我很好奇，果你回头看看这几年，可以告诉我你的一些想法吗？AI 和深度学习在这些年是如何逐渐演变的。

Ian：十年前，我觉得，作为一个社区，机器学习中最大的挑战就是这样，如何让它可以处理 AI 相关的任务，对于更简单的任务，我们那时有很好的工具，比如我们想要提取特征，识别规律，人类设计师可以做很多事情，他们设计出这些功能，然后交给计算机去做，这种做法对很多任务效果都很好，比如预测用户会不会点开广告，或者不同的基本科学分析，但当我们要处理几百万像素的图片时，就很困难了，或者处理音频波形，其中系统必须从零开始学到所有知识，五年前我们开始跨越了这个障碍，现在我们来到了一个时代。如果你想从事 AI 事业，有太多不同的道路可以走，也许他们面临的最难的问题是应该选择哪条路走下去，你希望让加强学习效果和监督学习一样好吗?你希望让无监督学习效果和监督学习一样好吗?你希望机器学习算法是不偏不倚，不会带上我们的偏见，尽量避免这些偏见，如果你想确保和 AI 有关的社会问题得到妥善解决，确保 AI 可以让所有人获益，而不是造成社会动荡和大规模失业?我想现在，真的可以做到很多不同的东西，可以避开 AI 的所有缺点，同时利用它能提供的所有优点。

吴恩达：今天有很多人想进入 AI 领域，你对这些人有什么建议?

Ian：我想很多想进入 AI 领域的人，一开始想，他们绝对需要获得博士学位或者这样那样的证书，我觉得实际上这已经不是必要条件了，其中一种获得注意的方式是，在 GitHub 上写很好的代码，如果你有一个很好玩的项目，解决了某人在前沿希望解决的问题，一旦他们找到了你的 GitHub 代码，他们会直接找到你，让你到他们那工作，我雇的很多人，去年在 OpenAI 或今年在谷歌招聘的人，我一开始想很他们合作因为，我见到他们在互联网开源论坛上发表的一些代码段，写文章并发表到 arXiv 上也是可以的，很多时候要将一个东西打磨完美成为对科学文献的新贡献是很难的，但在这之前你可能已经能开发出一个有用的软件产品了。

吴恩达：所以读读你的教材，在各种材料上练习然后把代码发布到 GitHub 或者 arXiv

上。

Ian：如果你要用那本教材学习，那么同时开始做一个项目是很重要的，总之要选择某种方式，将机器学习应用到你兴趣的领域，比如，如果你是一名田野生物学家，你想利用深度学习，也许你可以用它来识别鸟类，或者如果你不知道想用机器学习做什么，你可以去做，比如街景门牌号码分类器，这里所有数据集都设立好了，你直接可以用，这样你就可以练习一下，教材介绍到的所有基本技能，或者当你看给你解释概念的 Coursera 视频时，直接去练习。

吴恩达：所以在过去的几年里，我看到你做的一些对抗性样本的工作，告诉我们一下。

Ian：是，我想对抗性样本就是我称为机器学习安全的新领域，过去，我们看到计算机安全问题，攻击者可能会骗计算机跑错误的代码，这就是所谓的应用级安全性，以前有些攻击方式是人类可以骗过一台计算机，让它相信网络上的消息来自某个可信任的人，但其实不是真的，这就是所谓的网络级安全性，现在我们开始看到，你也可以骗到机器学习算法，让它们去做不应该做的事情，即使运行机器学习算法的程序运行的代码完全正确，即使运行机器学习算法的程序知道网络上所有消息的来源，也能骗到，我认为在新技术开发初期，考虑加入安全性非常重要，我们发现把一个系统建立起来以后，再引入安全性是很难的，所以我对现在要研究的想法非常激动，如果我们现在开始预见机器学习的安全问题，就可以从一开始确保这些算法的安全性，而不是过几年再回头打补丁。

吴恩达：谢谢，那太棒了，你的故事有很多神奇的地方，尽管已经认识你很多年了，我实际上并不知道，所以感谢你分享这一切。

Ian：你太客气了，谢谢你邀请我，这是很好的机会。

吴恩达：好的，谢谢。

Ian：非常感谢！

吴恩达采访 Ruslan Salakhutdinov

吴恩达：欢迎你，Rus，很高兴今天你能接受采访。

Rus：谢谢你，谢谢你，Andrew。

吴恩达：现在你是苹果公司的研究主管，你也有一个教职，卡内基梅隆大学的教授，所以我很想听听你的个人故事，你是如何一步步进入深度学习领域工作的呢?

Rus：是的，其实，某种程度上，我进入深度学习领域纯粹是运气，我在多伦多大学读的硕士，然后我休学了一年，我实际上在金融领域工作，很意外吧，那时候，我不太清楚是否要去读一个博士学位，然后发生了一些事情，发生了一些意外的事情，有一天早上我要上班路上遇到了Geoff Hinton，Geoff告诉我，嘿，我有这个好想法，来我办公室，我跟你说，所以我们基本上一起散步，然后他开始给我讲波尔兹曼机器，对比散度法还有其他算法，当时我并没有明白他在说什么，但真的，我真的很激动，太令人兴奋了，我非常兴奋，后来就这样，三个月内，我就跟着Geoff读博了，所以这是开始，因为那是2005，2006年的事了，这是一些原始的深度学习算法，使用受限玻尔兹曼机，无人监督的预训练，这些概念开始成熟，这就是一切的开始，真的是这样，那个特别的早晨，我偶然碰到了Geoff，完全改变了我未来的事业发展方向。

吴恩达：然后你其实是早期一篇关于受限玻尔兹曼机的论文的合著者，真正让神经网络和深度学习的概念涅槃重生，你可以告诉我更多关于那个研讨会的工作吗?

Rus：是的，这其实真的很激动人心，是的，那是第一年，我的PhD生涯的第一年，Geoff和我试图探索使用受限玻尔兹曼机的这些算法，使用预训练等技巧训练多层网络，具体来说 我们集中精力处理自动编码器，我们应该怎么高效的去做PCA的非线性拉伸呢?这是非常令人兴奋的，因为我们的系统能够处理MNIST数字，这是令人兴奋的，但那之后我们走的路，让我们看到这些模型真的可以推广到人脸识别，我还记得那时我们有这个Olivetti人脸数据集然后我们开始在想，是不是可以改善文档压缩，我们开始观察所有这些不同的数据，实值计数，布尔代数，用了整整一年，我还是博士生第一年，所以那是很充实的学习经验，但是 真的在六七个月内，我们已经能够得到非常有趣的结果，我是说非常好的结果，我想我们能够训练这些层次非常深的自动编码器，，这是当时还做不到的事情，用传统的优化技术做不到的事情，但接下来，这就变成了对我们来说非常激动人心的时期，那是超级激动人心的，是的，因为我那时不断在学习新知识，但同时，我们的工作出来的结果，真的很令人印象深刻。

吴恩达：所以在深度学习复兴初期，很多研究都集中在受限玻尔兹曼机上，然后是深玻尔兹曼机，还出来了很多令人兴奋的研究，包括你们组的一些研究，但玻尔兹曼机还有受限玻尔兹曼机器现在怎么样了?

Rus：是的，这是一个很好的问题，我觉得在早期，我们使用受限玻尔兹机器的方式，你可以想象一下训练一堆这些受限玻尔兹曼机，让你能够很有效地一次学习一层，有很好的理论基础，告诉你添加特定的一层之后，你可以在特定条件上证明它是有变分界限之类的，所以是有理论支持的，这些模型能够在预训练这些系统方面达到很好的效果，然后在2009年左右，2010年，计算力量开始出现了，GPU开始很强，我们很多人开始意识到，直接优化这些深层神经网络，可以得到类似的结果，甚至更好的结果。

吴恩达：所以只是标准的反向传播，不带预训练的受限玻尔兹曼机。

Rus：没错，没错，那之后大概过了三四年，大家都变得很兴奋，因为人们觉得，哇，你真的可以用预训练机制这么训练深层模型，然后，随着更多计算力量变强，人们突然发现，你可以直接做标准反向传播，那是2005年或2004年我们绝对做不到的事情，因为CPU计算要几个月的时间，所以这是一个很大的变化，另一件事，我们还没想清楚的是，如何使用玻尔兹曼机和深波尔兹漫机，我相信他们是非常强大的模型，因为你们可以把它们看成是生成性模型，它们试图对数据进行耦合分布建模，但是当我们开始研究学习算法时，现在学习算法，他们需要使用马尔可夫链，蒙特卡罗和变分学习等，它们并不像反向传播算法那样可以轻松扩展，我们还没有想出更有效训练这类模型的方法，还有卷积的使用，这些模型有点难以适应到现在的计算硬件，我还记得你当时有些工作用的是概率最大池化，来构建这些不同对象的生成模型，来构建这些不同对象的生成模型，但同时，训练这些模型还是很困难的。

吴恩达：可行性有多高呢?

Rus：是啊，可行性有多高呢?我们还是要弄清楚怎么办，另一方面，最近使用变分自动编码器

的一些工作，例如，可以看成是可以看作是玻尔兹曼机的交互式版本，我们想出了训练这些模块的办法，是Max Welling和Diederik Kingma的工作，或者使用重新参数化技巧，我们现在可以在随机系统内使用反向传播算法，正在推动着各方面很大的进步，但是波尔兹曼机这边，我们还没想出应该怎么做到这点。

吴恩达：这实际上是一个非常有趣的视角，我实际上并不知道在计算机较慢的早期RBM，预训练真的很重要，只有计算硬件力量的上升才推动了到标准反向传播的转变，在社区对深度学习思想的演变方面，还有其他话题，我知道你花了很多时间思考这个，生成的无监督方法vs 监督的方法，你可以给我们分享一下你的想法是如何随着时间推移演变的吗?

Rus：是的，我觉得这真是，非常重要的话题，特别是如果我们考虑无监督，半监督或生成模型，因为某种程度上我们最近看到的很多成功案例是来自监督学习的，而在早期无监督学习主要被视为无监督预训练，因为我们不知道如何训练这些多层系统，即使在今天，如果你的系统面对的是大量无标签数据和一小部分有标签数据的样本，这些无监督的预训练模型建立这些生成模型可以帮到监督训练，所以我觉我们社区里很多人都抱有这样的信念，当我开始做博士后，做的都是生成模型并试图学习这些堆叠起来的模型，因为那时是训练这些系统的唯一途径，今天在生成建模领域里有很多工作，你们看看生成对抗网络，你们看看变分自动编码器，深度能量模型是我的实验室现在正在研究的，我认为这是非常令人兴奋的研究，但也许我们还没有把它弄清楚，再次，对于正在考虑进入深度学习领域的许多人来说，这个领域我觉得，我认为我们还会取得很大进展，希望在不久的将来。

吴恩达：所以无监督学习。

Rus：无监督的学习，对，或者你可以看成是无监督学习或者半监督学习，其中我们会给一些提示或者例子，说明不同东西的含义，然后丢给你大量的无标签数据。

吴恩达：所以这实际上是一个非常重要的见解，在深度学习的早期，那时计算机要慢一些，必须用到受限玻尔兹曼机和深玻尔兹曼机，初始化神经网络权重，但随着计算机变得更快，直接反向传播开始效果更好了。还有一个话题我知道你花了很多时间去想的，监督学习和生成模型，无监督学习的对比，你的看法呢?可以告诉我们，关于这个争议的看法是怎么随时间演变的?

Rus：我们所有人都相信在那里可以取得进展，就是这些玻尔兹曼机，变分自动编码器，GAN所有工作，你认为这里很多模型都是生成模型，但是我们还没有弄清楚如何真正让它们变得可行，如何利用大量数据，即使对于.. 我在IT业界见到很多，公司有大量的数据，大量的未标记的数据，有很多注释数据的努力，因为现在这是取得进展的唯一途径，我们应该能够利用这些无标签数据，因为实在太多了，我们还没弄清楚如何做到。

吴恩达：所以你提到对于要进入深度学习领域的人，无监督学习是令人兴奋的领域，今天有很多人想进入深度学习做研究或应用工作，对于这个全球性的社区来说，要做研究或应用工作，你会有什么建议?

Rus：是的，我认为最关键的建议之一，要给进入这个领域新人的建议，我会鼓励他们尝试不同的事情，不要害怕尝试新事物，不要害怕尝试创新，我可以给你一个例子，当我是研究生时，我们正在研究神经网络，这些是非常难以优化的非凸系统，我还记得在优化社区的朋友聊天，他们的反馈总是这样，这些问题你们解决不了的，因为这些是非凸的，你们不了解优化，凸优化都那么难做，你们还要做非凸优化?然后令人惊讶的是在我们的实验室里，我们从来不关心这么具体的问题，我们正在考虑如何优化，看看能否获得有趣的结果，这种心态有效地推动了社区的发展，我们没有害怕，也许一定程度上是因为，我们没有认真研究优化背后的理论，但我会鼓励人们尝试，不要害怕，要试试挑战一些困难的问题。

吴恩达：是的，我记得你曾经说过，不要只学编写高层次的深度学习框架，应该实际了解深度学习的底层。

Rus：是的，没错，我认为，当我教深度学习课程时，我一直尝试去做的是，在作业里，我要求人们实际编写，卷积神经网络的反向传播算法，那很痛苦，但如果你做过一次之后，你就真正了解这些系统背后的运作原理以及如何在GPU上高效实现它们，我认为当你进入研究或工业界时，最重要的是你对这些系统实际在做的事情要有很深入的理解，所以我想这很重要。

吴恩达：既然你有学术界当教授的经验，也有企业经验，我很好奇，如果有人想进入深度学习领域读博和进入公司各有什么利弊?

Rus：是的，我认为这其实是个很好的问题，特别是在我的实验室里，我有各种背景的学生，有些学生想去走学术路线，有些学生想走工业路线，现在变得非常有挑战性了，因为在工业里你也能做出一流的研究，你也可以在学术界做出一流的研究，但利弊方面，在学术界，我觉得你有更多的自由来处理长期的问题或者如果你喜欢思考一些疯狂的问题，你想自由自在的工

作，去研究，与此同时，在企业里的研究也很激动人心，因为很多情况下，你的研究可以直接影响到数百万用户，比如开发一个核心的AI技术，显然，在企业里你有更多的计算资源，能够做到非常惊人的事情，所以都有优缺点，这真的取决于你想做什么，现在环境非常有趣，有学术界转向工业界的，还有工业界的转向学术界的，虽然更少一点，但现在是非常令人兴奋的时代。

吴恩达：听起来学术界机器学习不错，企业界机器学习也很棒，最重要的是跳进去，对吧?选一个，跳进去。

Rus：这真的取决于你的喜好，因为你在什么地方都可以做出惊人的研究。

吴恩达：所以你提到无监督学习是一个令人兴奋的研究前沿，你觉得还有其他领域你觉得是令人兴奋的研究前沿吗?

Rus：是的，当然了，我想我现在看到，在现在的社区里看到，特别是在深度学习社区，有几个趋势，我认为有一个特别的趋于特别令人兴奋是深度加强学习领域，因为我们能够弄清楚，我们如何在虚拟世界中训练代理程序，这是过去几年里，我们看到很多的有很多进展，我们是如何把这些系统推广到更大规模上，我们如何开发新的算法，如何让代理程序互相沟通，我觉得这个领域，一般来说，你能和环境交互这些场合是非常激动人心的，我认为还有另一个领域令人兴奋，就是推理和自然语言理解的领域，我们可以建立基于对话的系统吗?

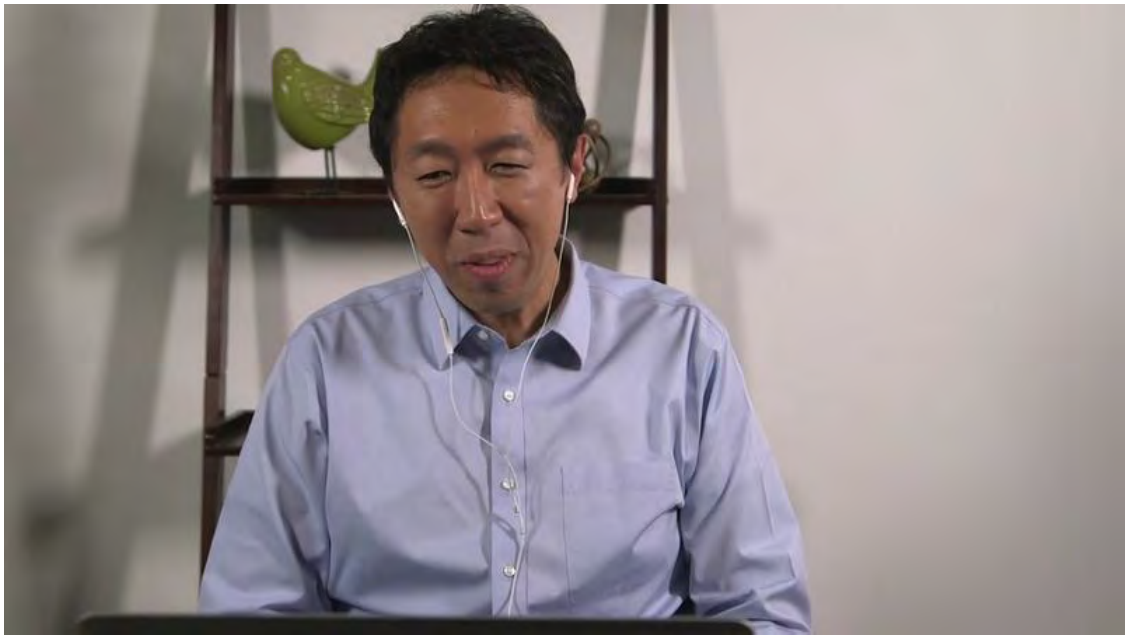
我们可以建立能够推理，能够读懂文本的系统，能够智能回答问题的系统吗?我认为这是现在很多研究的重点，然后还有另一类子领域也是，这个领域可以从很少的几个例子中学到知识，所以通常人们说是一次学习或迁移学习，你从环境习得某种东西，然后我给你一个新的任务，你可以很快地解决这个任务，就像人类一样，而不需要很多很多带标签的样本，所以这个概念是我们社区里很多人都想弄清楚的，我们应该如何做到，如何达到接近人类的学习能力。

吴恩达：谢谢Rus给我们分享所有的评论和见解，更有趣的是，能听到你的早年故事

Rus：谢谢 Andrew，是的，谢谢你的采访。

吴恩达采访 Yoshua Bengio

吴恩达：Yoshua 你好，很高兴您能参加我们的访谈活动。



Yoshua：我也很高兴。

吴恩达：您不仅仅是深度学习方面的研究员，工程师，还是该领域和学术研究界的代表人物之一，我很想知道您是如何入门的，您是如何进入深度学习这个领域 ，并不懈探索的。

Yoshua：小的时候，我读了很多科幻小说和很多同龄人一样，1985 年研究生阶段，我开始阅读神经网络方面的论文，当时特别兴奋，也逐渐对这个领域燃起了热情。

吴恩达：1980 年代中期 1985 年。您还记得当时读到那些论文的情形吗？

Yoshua：当时和专业人士们一起上经典的 AI 课程，我突然发现，这个领域研究的都是人类如何学习，人工智能，如何把人类学习与人工智能和计算机联系起来这样的问题，发现这类文献的时候我异常兴奋，于是开始拜读 Geoff Hinton 等人撰写的，关于联结主义的论文，我研究过循环神经网络，语音识别，HMN，图模型，之后我很快进入了 AT&T 贝尔实验室和麻省理工学院攻读博士后，并发现了训练神经网络的长期依赖问题，之后不久，我受聘来到蒙特利尔大学任教，我的年少岁月多半也都是在蒙特利尔度过的，过去几十年一直投身于此。

吴恩达： 您一定深有感触，谈谈您对深度学习的看法以及神经网络这些年的发展历程吧。

Yoshua：我们从实验，直觉认识入手，而后提出了一些理论，现在我们在认识和理解上

清晰了很多，比如为什么 Backprop（反向传播）如此行之有效，为什么深度如此重要，当时我们对这些概念没有任何可靠的论证依据，2000 年初，我们开始研究深度网络的时候，我们直觉认为神经网络更深才会更加强大，但是我们不知道应该如何深化，如何优化，当然，我们最初进行的相关实验也未能成功。

吴恩达：与 30 年前相比，您认为哪些最关键的设想得到了时间的验证，而哪些又错得最让人意想不到我犯过的一个最大的错误就是和当时 90 年代所有人一样，我也认为执行 backprop 需要光滑非线性算法，因为我觉得，如果当时我们有非线性矫正算法，它有些部分会比较平坦，训练起来就很难了，因为很多地方的导数都是 0，2010 年前后，我们开始在深度网络中尝试 Relu 算法，我当时执着地认为，我们应当注意导数为 0 区域上的神经元会不会太饱和，最终，ReLU 比 sigmoids 函数的效果更好，这出乎我的意料，我们探索这个函数是生物连接的原因，并非我们认为它更容易优化，但结果证明它效果更好，之前我还认为它训练起来会比较难。

吴恩达：请问，深度学习和大脑之间有什么关系，虽然有明确答案，但我更想知道您对此的看法。

Yoshua：最初让我关注神经网络的一种见解是，连接主义者提出信息是分布在被激活的神经元中，而不是由祖母细胞来描述的，他们称之为“符号描述”，它是传统 AI 的观点，我依然相信这是非常重要的信息，甚至近期，人们重新发现它的重要性，它确实是一项重大发现，深度学习是后来才提出的，大约在 2000 年初，但是我 90 年代研究的并不是这些。

吴恩达：是的，我记得，很早以前，您曾搭建过许多相对不深的词向量分布式表达。

Yoshua：是的，没错，那是 90 年代后期，我很感兴趣的内容之一，我和我兄弟 Samy 一起做了尝试就是使用神经网络来解决维数灾难的问题，它是统计学习中的一个核心问题，我们能够以一种非常高效的方式，利用这些分布式表达来表示许多随机变量的联合分布，效果很好，之后我把它扩展到词序列的联合分布，这就是词向量的由来，因为我当时认为，这可以实现对拥有相似语义的单词的泛化。

吴恩达：过去 20 年，您的研究小组完成了多项研究，提出了很多想法，短短几分钟之内无法细数，我好奇的是，小组中的哪项研究或想法，最让您感到自豪。

Yoshua：好的，我前面提到了长期依赖的研究，我想人们依然不能很好理解它，然后是我刚提到的维数灾难，还有近期应用于神经网络的联合分布，它是由 Hugo Larochelle 负责的涵盖了应用于联合分布的，学习词向量的各项工作，然后，我觉得，我们最被关注的研究就是深度学习在自动编码器和 RBMs 堆栈上的应用，还有就是，如何更好地理解用初始化参数解决深度网络训练的难点，还有深度网络中的梯度消失，这项研究及后续实验体现了分段线性激活函数的重要性，其他重要研究还包括无监督学习，降噪自动编码，GANs，这些都是当前非常流行的生成式对抗网络，我们对基于注意力机制的神经网络机器翻译的研究，对翻译工作起到了很重要的作用，现在已经应用到工业系统中，如谷歌翻译，对注意力机制的研究确实改变了我对神经网络的看法，我们曾经认为神经网络只是机器，不过是向量之间的映射，但基于注意力机制，我们现在可以处理各种数据结构，这的确打开了很多有趣领域的大门，生物学联结方面，最近两年我一直在研究的一个课题是，就是我们如何想出像 backprop 这样的概念，而且大脑也可以执行，我们已经发表了几篇论文，神经科学界人士对此很感兴趣，我们将继续对这个课题的研究。

吴恩达：我知道你一直关注的一个话题就是，深度学习与大脑之间的关系，能谈谈这个吗？

Yoshua：我一直在思考两者这间的生物学关联，而且平日里也经常"幻想"，因为我觉得

它就像个谜题，首先通过学习大脑，研究大脑，我们有这么多的证据，如穗时序依赖型可塑性，但另一方面，们又有这么多关于机器学习的概念，比如针对一个目标函数对整个系统进行全局训练，比如 Backprop，，么 Backprop 到底是什么意思，还有 "信用分配"到底是什么意思，当我开始思考大脑如何能像 backprop 一样工作时，就想到，许在 backprop 背后，在着更通用的概念，可以让 backprop 更高效地工作，也许有很多方法可以完成信用分配，这也呼应了增强学习领域的一些问题，所以说这一点很有意思，一个简单的问题开始，你会一步步思考更多的问题，而让你把这么多不同的事物联系起来，像在解一个大谜题，这个过程持续了几年，我要说一点，这些所有的尝试大一定程度上是受了 Jeff Hinton 的启发，他在 2007 年的时候发表过一次演讲，当时是第一个关于深度学习的研讨会，他讲了他对于大脑工作方式的想法，比如怎么利用时间编码，来辅助 backprop 的一些工作，这件事对我近几年的一些探索起到了引导性的作用，说起来这一路走来，已经有 10 多年了。

吴恩达：另外我经常听到你谈到的一个话题就是无监督学习，能说说关于这方面的想法吗？

Yoshua：当然，无监督学习非常重要，目前业内的系统都是基于有监督学习的，这就要求人类先定义出当前问题的一些重要概念，并在数据库中讲这些概念标记出来，目前的玩具问题服务，系统都是基于这个的，但人类本身其实可以做得更好，人类可以探索世界，通过观察发现新的概念并与世界互动，2 岁的小孩，能自己理解直观物理概念，比如他们理解重力，理解压力，理解惯性，理解液体，固体，而且他们的父母并没有给他们解释过这些概念，那他们是怎么理解的呢，这就是无监督学习所要解决的问题，不是简单地在数据库中做不做标识的问题，而是如何构建一个精神结构，从而解决如何通过观察来解释世界，最近，我还在尝试，把无监督学习和增强学习整合在一起，因为我相信，我们在努力尝试解释的重要基础概念，原本是有很强的指示的，但我们可能没有把它们联系起来，也就是说，人类或机器如何通过与世界互动，通过探索世界，尝试不同的事物和控制事物，这些是我认为与无监督学习紧密关联的，所以我 15 年前一开始研究无监督学习时，从不同的机构，到 RBM 等等，最主要关注的是如何实现学习"良好的表现"，当然现在这个问题也仍然重要，但是我们不确定的是到底如何定义“良好的表现”，比如如何确定一个目标函数，过去几年我们试了好多方法，这也是无监督学习研究的一个魅力所在，解决方案的想法有很多，解决问题的方式有很多，也许明年我们就会发现一个完全不同的方案，也许大脑的工作方式跟我们现在所知的完全不一样，目前该领域还没有进入渐进式研究，它本身还是在探索一个完全未知的领域，我们现在还不能清晰地确定什么样的目标函数，能够评估一个系统在无监督学习上是否表现的

很好，当然，这很有挑战性，但同时它也意味着更多未知的可能，而这一点正是研究人员最为着迷的，至少我是这样。

吴恩达：今天，深度学习已经得到了长足的发展，我想目前不管对于谁来讲，都不太可能遍历现今所有关于深度学习的著作了，所以我比较好奇，关于深度学习，你最感兴趣的是什么。

Yoshua：我比较有想法，我感觉现在关于深度学习的研究，已经脱离了我心目中最理想的方向，我感觉现在机器经常出现些小错误，说明它对世界的认知还很表面化，所以最让我兴奋的是，我们现在的研究方向，不是要让我们的系统实现什么功能，而是回到最本原的原理，如何让电脑来观察世界，与世界互动，发现世界，即使世界很简单，就像编一个电脑游戏，我们也没法做得很好，但是这点很好，因为我不用跟谷歌，Facebook，百度他们竞争，因为这就很根本的研究，谁都可以做，而且在家里的车库里就可以了，当然，解决这个问题的方向有很多，但我也看到很多沟通互动功能方面，深度学习和增强学习的融合起着很重要的作用，我非常激动的是，这个方向的研究进度，，在实际应用方面已经有了很大的成就，因为如果看一下我们在实际应用方面面临着巨大的挑战，比如如何应对新的领域或新的范畴，现成的成功的案例很少，但是人类解决这些问题是完全没有问题的，而对于这些迁移学习的问题，如果我们能够让计算机更好地理解世界，处理这些就会效果好很多，更深度的理解，是吧，眼前的世界在发生什么，我看到的是什么，如何通过行为来影响我的所见所闻，这些是我最近最为关心的问题，另外过去 20 年人工智能面临的老问题，深度学习都已经有了变革性的发展，因为深度学习的大多数成功是在认知层面的，那接下来的目标呢，接下来的就是高级条件，就是在抽象层面明白事物的原理，现在我们在研究如何能让机器理解更高级的抽象层面，目前还没有达到，所以我们要继续努力，我们要研究原因，研究信息的时序处理，要研究因果关系，如何让机器自己来发现这些事物，未来可能还需要人类的指导，但主要还是要自主完成。

吴恩达：根据你的描述来看，你是喜欢使用研究方法的，比如你在做的"玩具问题"实验——这里完全没有贬意，是的，但只是对于小问题，而且你相信这个未来可以转到大的问题上。

Yoshua：是的，完全可以，当然我们先要一点点加大规模，从而最终实现，对这些问题的解决，但是之所以先从玩具问题入手，是因为它可以让我们更清楚地明白我们的失败，从而开始对直观物理概念认识的研究，让机器更容易地理解这些概念，算是经典的分步解决方法吧，另外，我觉得有些人可能没想过的是，其实研究周期是可以快很多的，如果几个小时

就能做一个实验，我的进度就可以快很多，如果需要尝试一个较大模型，用来捕捉常识，和普通知识层面的事物，当然这个最后我们是要去做的，只是现有的硬件，每次实验都要花太多时间，所以在硬件运行速度提到上千上百万倍之前，我只能先做玩具问题的实验。

吴恩达：您还说过，深度学习，不仅仅是一门工程学科，还要多下功夫去了解其中的来龙去脉，可否分享一下您的看法。

Yoshua：是的，的确如此，我们目前的工作（方式）有点像盲人走路，我们运气不错，也发现了一些有趣的东西，但是，如果我们能够稍微停一停脚步，试着以一种可以转换的方式理解我们所做的事情，因为我们要归于理论，我所说的理论不一定是数学，当然我喜欢数学，但是我不认为一切事物都要数学化，而是要逻辑化，并不是我可以让别人相信这样有用，可行，这才是最重要的，然后再通过数学来强化和精炼，但更多的是理解，还有做研究，不是要成为下一个基准或标杆，或者打败其他实验室，他公司的人，更多的是，我们应该提出哪些问题来帮助我们，更好地理解我们感兴趣的现象，比如，是什么导致更深度神经网络，或循环神经网络很难训练，我们已经有了一些认识，但还有很多东西我们不了解，所以我们设计实验，可以不以获得更好的算法为目的，而是以更好地理解现有算法为目的，或者研究某算法在什么情况下表现更好并找到原因，为什么才是真正重要的，科学的意义就是提疑解疑。

吴恩达：现如今，有好多人希望进入这个领域，您会给那些想进入 AI 和深度学习领域的人一些什么建议呢？我知道在一对一活动上，您已经回答过很多次了，不过我还是代观看视频的所有网友问一问。

Yoshua：首先大家可以从事的工作和目的各不相同，研究深度学习和应用深度学习来构建产品，所需要具备的东西可能并不一样，在认知水平上，这两种情况是不同的，但是无论哪种情况都需要亲身实践，想真正掌握一门学问，比如深度学习，当然，也需要大量阅读你必须自己动手编程，我经常面试一些使用过软件的学生，现在有太多即插即用的软件，使用者根本不了解自己在做什么，或是只停留在粗浅了解的程度，一旦出现问题，使用者很难解决，也不知道原因，所以大家要亲自实践，即便效率不高，只要知道是怎么回事就好，这很有帮助，尽量亲自动手。

吴恩达：所以不要用那种几行代码就可以解决一切，却不知道其中原理的编程框架。

Yoshua：没错，没错，我还想说的是，如果可以的话，尽量从基本原理入手获取认识，真的很有帮助，当然，平时还要多阅读，多看看别人的代码，自己动手编写代码，多做实验，要真正理解自己做的每一件事，尤其是科学研究层面，多问问自己，我为什么要这么做，人

们为什么要这么做，也许书中就能找到答案，所以要多读书，如果能自己搞清楚当然更好。

吴恩达：很酷的建议，我读了您和 Ian Goodfellow、Aaron Courville 合编的书，各方评价很高。

Yoshua：谢谢， 谢谢，是的 ，销量不错，有点超出预料，我感觉现在读它的人要比能读懂它的人多，呵呵呵，ICLR 大会论文集可以说是，优秀论文最集中的所在，当然，NIPS，ICML 和其他大会也有非常优秀的论文，但如果你想阅读大量优秀论文，去看看最近几年的 ICLR 大会论文集就好，你会对这个领域有一个良好的认识。

吴恩达：酷，有人会问 ，关于深度学习，如何才能做到掌握和精通？对此您有什么看法和建议呢。

Yoshua：这个取决于你的背景，不要畏惧数学，只管发展直觉认识，一旦在你在直觉经验层面得心应手了，数学问题会变得更容易理解，好消息是，不需要攻读五年博士也可以精通深度学习，你可以很快速地上手学习，如果你具备良好的计算机科学和数学功底，几个月时间，你就可以学到足够的应用知识来构建系统，并开始进行研究性实验，接受过良好培训的人可能 6 个月左右吧，或许他们对机器学习一无所知，但是，如果他们精于数学和计算机科学，学起来会非常快，当然，这意味着你要具备良好的数学和计算机科学背景，有时候，计算机科学课程上学到东西还远远不够，你还要继续学习，尤其是数学知识，比如，概率，代数和最优化。

吴恩达：了解，还有微积分。

Yoshua：对，对，还有微积分。

吴恩达：非常感谢 Yoshua 与我们分享这些看法，见解和建议，虽然认识您已久，但很多早期经历的细节我也是今天才知道，谢谢。

Yoshua：也谢谢 Andrew 制作了这样一个特殊的访谈以及现在所致力的事情，希望能够为大家所用。

吴恩达采访 林元庆

吴恩达：欢迎你，元庆，今天你能参与我们的活动，我真的很高兴。

林元庆：没问题。

吴恩达：如今你是 IT 届的领头羊，当中国政府寻求英才，去着手建立国家深度学习实验室时，他们发掘了你，我认为在深度学习领域，你可以说是全国第一人，我想问一些关于你工作的一些问题，但在这之前，我想听听你的个人经历，那你是如何发展到做现在的工作的？

林元庆：好的，事实上，在我读博士之前，我的专业是光学和物理学有很多共同之处，我认为，我的数学基础很坚实，我来美国之后，我就想什么专业能做我的博士课题？我就想，好吧，我可以选择光学或者其他什么的，早在 2000 年的时候，我觉着那时候纳米技术很火，但我当时就想也许我应该着眼于更激动人心的东西，恰巧那时有个好机会，我在宾夕法尼亚大学上课，在那儿，我认识了 Dan Lee，之后，他成了我的博士导师，我就想机器学习是很棒的一件事情，我兴奋异常，然后换了专业，因此我是在宾夕法尼亚大学读的博士，我的专业是机器学习，我在那呆了 5 年，很令人兴奋的时光，我学到了很多东西，很多算法甚至是 PCAs，我以前从不知道那些东西，我感觉每天都会学到新的东西，所以对我来说，那是极其令人激动的经历。

吴恩达：那是你许多新尝试之一，你做了很多工作，尽管在那些时代并不为人所欣赏。

林元庆：对，确实是，所以我认为 NEC 是个神奇的地方（NEC：美国智能图像研究院），刚开始我在那儿的身份是一名研究人员，我也喜欢，学到很多东西的感觉，事实上，在 NEC 的后期，我开始研究计算机视觉方向，说真的，在计算机视觉领域，我起步相对较晚，那时我做的第一件事是参加了 ImageNet 大规模视觉识别挑战赛，那时这个比赛第一年举办，我负责一个团队研究一个项目，很幸运，我们实力很强，我们得了第一，以绝对的优势得了第一。

吴恩达：所以你是 ImageNet 挑战赛冠军的第一位得主？

林元庆：是的，我在那场会议上做了演讲，对我来说，那是一次很棒的经历，那让我接触到了如此大型的计算机视觉技术，从那时起我就开始研究这种大型问题了，当纽约时报头版文章发表后，之后关于 AlexNet 也被发表时，我真的很震惊，我想，哇，深度学习是如此强大，从那以后，我在此方面付出很多努力。

吴恩达：作为中国国家深度学习实验室的主任，你们肯定正研究很多振奋人心的项目，那对正在观看的全球观众而言，关于这个实验室，他们应该了解些什么？

林元庆：国家工程实验室的目标就是建立一个巨大的深度学习的平台，希望是最大的一个或至少在中国是最大的，这个平台会给大家提供深度学习的框架，类似于 PaddlePaddle，我们会提供大规模的计算资源，我们还提供庞大的数据库，如果大家能够能在这个平台上开展研究或开发好的技术，我们会提供巨大的应用空间，比如说，技术在巨大的应用如百度上被证明，技术水平就可以融合进来并对其做出改善，因此，我们认为整合这些资源，我认为会造就一个强大的平台，我各举一个例子，比如说，我们现在发表了一篇文章，另一个人想要重复操作的话，最好的方法就是在某平台处提供代码，之后你就可以把代码下载到电脑上，，你会尝试寻找某处的数据组，然后你大概也需要得到好的计算能力，以便让你的计算资源运转如飞，所以这一切会让你省些功夫，在国家实验室工作将会变得很容易，如果某些人应用此平台做一些工作，写一些文章，实验室能在平台上拥有这些代码，计算架构已经建立起来了，数据也是，基本上你只需要一根线把数据库连接起来，所以，这可以给计算机科学重复性问题的损失带来巨大改善，所以，很简单的，在几秒之内，你就可以开始学习文献上的一些东西，对，这是很强大的，所以，这只是我们工作中的一个例子，以确保我们提供给整个社会和产业一个真正有效的平台。

吴恩达：太神奇了，这确实加速了深度学习研究。

林元庆：没错。

吴恩达：你能透露下中国政府给国家深度学习实验室提供了多少资源吗？

林元庆：我认为，对于国家工程实验室而言，政府可以投资建设一些基础设施，但我认为更重要的是这将会成为中国的一个旗舰机构，引领许多有关深度学习方面的研究，包括像国家项目，法律政策等，事实上这是很有效的，我认为百度，我们很荣幸拥有这个实验室。

吴恩达：你可以说是居于中国深度学习领域的核心地位，中国有很多项目，全世界的观众都还未曾意识到或见到，那国外的人们，应该对中国的深度学习领域有哪些了解呢？

林元庆：我认为在中国，尤其是过去的几年，深度学习充实了一个产品，此领域的确在急速发展，从搜索引擎到，比如说，词组识别，监控，乃至电子商务等许多方面，我认为，他们正在深度学习领域投入巨大的努力，并充分利用技术使这个产业变得更有影响力，总体而言高科技的发展是很重要的，我认为对我来说，能和许多人分享这些，我们相信，这是很重要的，这通常被称为正向循环，举个例子，我们开始构想建立一些技术，这些技术会有一些初始数据，也会有一些初始算法，这些会形成服务的初始产品，接着我们能获得用户的数据，其他人会得到更多的数据，所以，我们就可以研发更好的算法，因为我们看到更多的数据就会找到更好的算法，于是我们就有了更多的数据和更好的算法，我们就可以为产品服务提供更先进的技术，然后我们也绝对希望会吸引到更多的用户，科技变得更加先进，接着，我们就可以得到更多的数据，这是非常好的积极地举措，这也是很特别的，尤其对于 AI 相

关的科技和比如激光等传统科技而言，我以前研究过那些，所以，科技发展的过程会变得很线性，但之前 AI 科技因为有了正向的循环，你可以想象科技肯定会发展的非常快，当我们进行研究时，这也是很重要的，当我们设计 ND 时，我们在快速发展时期的方向研究，，但如果整个产业没能够建立起这个正向循环，如果我们没能够建立起这个强大的正向循环，这很可能行不通，因为有远见的其他人会建立起此循环，他们会比我们更快的到达那个水平，对我们而言，这个逻辑很重要，需要我们注意，比如说，你需要一个公司，那我们该在哪个方向研究，不该在哪个方向研究，，这绝对是需要注意的因素。

吴恩达：如今，无论在中国，还是美国乃至全球有许多人想进军深度学习和 AI 领域，对于那些人你有什么建议？

林元庆：如今，初始进军者肯定拥有开源框架，我认为这是很有效的，当我开始研究深度学习时，当时并没有很多开源资源可用，但今天，在 AI 特别是深度学习领域，是很好的一个社区，那有很多很杰出的人才，就像是 TensorFlow，一个 caffe，他们也称之为 caffe 2 是吧？在中国，我们有很好的 PaddlePaddle 甚至是在网上，他们有很多教学课程，教你怎么利用那些资源，还有，如今也有很多公共可用的标准，人们会看到技术高超，富有经验的先驱，比如，他们是怎么利用那些标准的，总的来说，是接触深度学习的好时机，我认为，这些都是很好的出发点。

吴恩达：你是如何得到这些启发的？

林元庆：事实上，我曾走在南辕北辙的路上，我学习了 PCA LDA 等其它之后，才学习的深度学习，总体而言，我感觉这也是条不错的学习道路，打下了很多基础，我们学习了图模型，这些都是很重要的，尽管现在，深度学习发展的无法想象，但知道一些规则会给你关于深度学习的运作模式很好的直觉感，然后有一天，深度学习和法则之间会产生关联，就像框架和途径那样，我认为这之间存在很多联系，这些法则丰富了深度学习，我的意思是为深度学习提供了多样途径，是的。我认为开放源码是很好的开始，那是很有效的资源，我还会建议你学习一些有关机器学习的基础知识。

吴恩达：谢谢你，听起来很棒，即使我认识你很长时间了，我现在都没想到你思考的很多细节，很感谢你。

林元庆：谢谢让我参与。

吴恩达采访 Pieter Abbeel

吴恩达：谢谢你，Pieter，能够来到这，很多人都认为你是一位有名的机器学习，深度学习，机器人技术的研究者，我想让更多人知道你的故事，你是怎么开始的呢。

Pieter Abbeel：这是个好问题，要是你问 14 岁的我，我的志向在什么，可能就不会是现在的答案，那时候我想当职业篮球运动员，我不认为我能做到。

吴恩达：机器学习侥幸做成了，篮球应该没戏。

Pieter Abbeel：是的，没戏，打篮球很好玩，但是变成职业不太行，在学校我最喜欢物理和数学，所以之后学工程就比较自然，也就是实际应用物理和数学，之后，我本科毕业于电子工程后，我不太确定做啥，因为工程相关的一切都太有趣了，去懂得一个东西如何运作很有趣，建一个东西也是某种意义上，AI 赢了，因为看起来它在某种程度能帮助所有学科，并且它看起来是一切东西的核心，你会思考一个机器怎么思考的，那可能是一切的核心，也就不用选某个特定的学科。

吴恩达：我一直在说 AI 是新的电力，听起来 14 岁时的你，已经有很超前的意识，过去几年你在深度增强学习中贡献很大，现在情况如何，为什么深度增强学习突然变得重要。

Pieter Abbeel：在我研究它之前，我研究了很多增强学习，与你和 Durant 在斯坦福的时候， 当然了，我们做了自主直升机飞行，后来去伯克利和我的学生们，做了个会叠衣服的机器人。可以说是用组合的学习方式来描述，做成一些不学习就无法做成的事，也结合了领

域知识才能完成，这很有趣，因为你需要领域知识，想办法学到很好玩，但同时每次想做成新的应用都会很耗时，需要领域知识和机器学习知识，在 2012 年，通过 Geoff Hinton 多伦多小组对 ImageNet 的研究结果突破 AlexNet 证明监督学习，突然之间大大减少了工程量，AlexNet 中所用的工程量非常之小，让我开始思考，我们该用类似观点回顾增强学习，看我们能否用增强学习做与监督学习同样有趣的事。

吴恩达：听起来你早于，大多数人前，发现深度监督学习的潜力，展望未来，你看到下一件事是什么，你对下一阶段有什么样的预测在深度增强学习中。

Pieter Abbeel：我认为深度增强学习有趣在于，某种程度上比监督学习有更多的问题，在监督学习中问题在于输入，输出，映射；在增强学习中是数据是从何而来的，就是探索问题本身，当你有数据之后，你怎么做信用赋值，你怎样理解早期做什么，才能之后受益以及安全问题，当你有自主收集数据的系统后，在很多情况下其实很危险，想象一家自驾车公司说，我们只会用深度增强学习来运行车辆，听着就像会有很多事故，在真正起作用之前。

吴恩达：你需要反面例子，对吧。

Pieter Abbeel：是需要反面例子，希望也有正面的，我觉得深度增强学习还有很多挑战，

在解决特定问题时及如何做成功，深邃的部分在于表达，增强学习本身还有很多问题，我个人觉得是深度学习的进步，一部分增强学习的谜团能被解开，就是表达的部分，如果有模式，我们可能表达为一个深度网络，并抓取模式，怎么分解模式仍然是增强学习中很大的挑战，我认为大挑战在于，如何让系统在长时间线上保持推理能力，现今很多深度增强学习，都是短时间线上的。还有很多问题是，如果对五秒的反应不错，对整个问题的反应都会不错。五秒和一天的规模相差很大，或是让机器人或软件主体保持一辈子的功能。我认为这里面有很多挑战，安全方面的挑战是如何安全持续地学习，当效果已经很不错时。所以，再举个例子，很多人对自主驾驶车很熟悉，让它们比人开车更好，假设人在每三百万英里，碰到一次很糟的事故，会需要很长时间才会有负面数据，如果和人一样好的话，但你想要更好，那么数据的收集就变得很难，很难得到让系统表现更好的数据，探索其中有许多牵连的挑战，我最兴奋看到的是，我们能不能往后一步，还是能学增强学习的算法，增强是很复杂，信用赋值也是，探索更是，就像对于深度学习中的监督学习，可以做到替代领域知识，可能也能有程序--增强学习的程序也能做到所有，取代我们亲自设计细节。

吴恩达：比如整个程序的回报函数。

Pieter Abbeel：这就需要学习整个增强学习程序，可能是，想象一下，你有这么个程序，不管是什么，你给它一个问题，看多久才能学会，然后发现，嗯，花了挺久，让另一个程序

来修正原本这个，加完这一步，试试看，要是学的快多了，修正的就很好，然后从此想办法继续进步。

吴恩达：我明白了，奠定基础。

Pieter Abbeel：这可能和可用的计算量有关，就像在内环中运行增强学习，现在，运行增强学习是最后一步，越多计算量越有可能运行的了，就像在内环的增强学习是更宏观算法的一部分。

吴恩达：从 14 岁起，你在 AI 工作有二十多年了，你怎么理解 AI 这段时间的进化。

Pieter Abbeel：我试着理解 AI 的时候很有趣，因为其实与我在斯坦福的硕士学位相符，其中有很多领袖我亲自交流过的，比如 John McCarthy，但用的方法与众不同，相比较 2000 年时，大多数人在做的事，然后和 Daphne Koller 的交谈，形成了我对 AI 最初的想法，她的 AI 课程，她的概率图模型课，让我真正产生兴趣，随机的变量如何应用于简单的分布，再被分成子集变量，然后其他的结论，都会让你知道好多，要是能让它计算起来也容易，那就很具挑战了，从那之后，我开始攻读博士，你到斯坦福后，是个很棒的现实提醒，也就是，不该去检验你研究里的度量是否正确，而是去看一种关联，你在做的事情和影响你的东西之间有何关联，你的研究实际改变了什么，而不是具体里面出现的数学。

吴恩达：对，很棒，我没发现之前遗忘了这点。

Pieter Abbeel：是的，这是其中一件，除了大多数人在问的，哪一个 Andrew 给的意见，一直在影响你，是确保你能看到这种关联性。

吴恩达：你已经并且会继续在 AI 领域有很棒的职业前途，对一些在看视频的人，要是他们想入门 AI，你有什么意见。

Pieter Abbeel：我想现在是进入 AI 的大好时机，需求量是如此之高，工作机会如此之多，有大量研究课题，也有大量创业机会等等，所以我很肯定决定入行是很机智的决定，你们中的大多数都能够自学，不管是不是在学校里，有好多好多网课，比如 andrew 的机器学习课，也有，比如说，Andrej Karpathy 的深度学习课，也有线上视频，作为开始很棒，Berkeley 也有增强学习的课程，所有课都在网上，都是很棒的入门方式，很重要的是，你自己要真正着手去做，不是只看看视频，要亲自试，可以用 TensorFlow，Chainer Theano，PyTorch 等等，随便你喜欢哪个，开始是很容易的，进展也会很快。

吴恩达：只要不断练习，对不，实际操作并关注什么成功了，什么没成功。

Pieter Abbeel： Mashable 上周有篇文章，讲一个英国 16 岁，Kaggle 比赛中的领导，里面提到他跑去学习，在网上找到东西，自学了一切，从未上过任何正统课程，一个 16 岁的少年能在 Kaggle 上有这么强的竞争力，说明是真的可能。

吴恩达：我们生在好时代，对人们学习的好时代。

Pieter Abbeel：完全正确。

吴恩达：还有个应该常被问到的问题是，要是有人想进入 AI 机器学习，深度学习的领域，他们该读博还是去大公司工作。

Pieter Abbeel： 很大程度上取决于你能得到的指导，在博士学位里，基本肯定的，是教授的本职工作，也是你的导师，会来主动找你，竭尽他们所能去成就你，帮助你在任何领域，比如 AI，得到成功，有很多用功的人，有时会有两个导师，那是他们的本职工作，也是为什么他们当了教授，大多数教授都能帮助学生做更多事情，但不是说在公司里不是这样，很多公司有很棒的导师并且很爱帮助教育和推进其他人等等。只不过是，可能不会有那样的保障，不会像攻读博士一样，而博士的一大优点，就是你肯定能学到好多并且总有人能帮助你学习。

吴恩达：取决于公司和博士项目本身。

Pieter Abbeel：完全正确，但是我想，主要还是得自己能学很多东西，要是自学的很快，再加上一个更有经验的人，能加速过程，因为他们的职责就是帮助推进你的进步。

吴恩达：你是深度增强学习中很有名的领袖之一，那么是什么样的东西，在深度增强学习中已经很成功了。

Pieter Abbeel：关注深度增强学习中的成功例子，非常非常引人入胜，比如说，通过像素学习玩 Atari，处理像素也就是被处理的数字，被变成一个游戏中的行动，然后，举个例

子，我们在 Berkeley 做过的一些研究，造了会走路也有回馈的模仿型机器人，只是简单的告诉它越往北走越好，对地面影响越小越好，它突然会选择，走路或跑步，即使并没有人告诉它什么是走和跑，或让机器人学讲给儿童的故事，并想法子组在一起，创造更多开放结局等等。

我认为有趣的是，它们能学习，从原始的感觉输入到控制，比如发动机中的扭矩，但是是同时完成的，有趣在于你能用一个单一算法，比如，推力是一瞬间的，你可以学习，可以让机器人学会跑，学会站立，可以有两条腿的机器人，再换成四条腿的，只要用同样的算法，它都会跑，所以增强算法不用改变，非常宏观，Atari 游戏也是如此，每个游戏里 DQN 都是同一个 DQN，当它开始进入，还未曾实现的边界后，它能为每一个任务从头一点点学起，但要是能重复利用，之前学过的东西，更好了，那样学下一个任务就更快了，这是目前未能实现的事情，本质上，它总是从头学起。

吴恩达：你觉得多快，就能看到使用深度增强学习的机器人，出现在我们身边，被广泛应用在全世界。

Pieter Abbeel：我认为真实情况是，从监督学习开始，行为克隆人类的工作，我觉得会有很多业务会建立起来，但总是有人类在幕后指挥工作，比如 Facebook 的信息助手，像这样的助手能被建起，但背后一定有人，做大量工作，机器学习能做人所做的事，并开始为人们提建议，人类会被提供，有限的几个选择，过一阵子，就能变得更好，可以给增强学习，一些实际目的，而不是让人在幕后工作，是实际给它们目标去完成，比如，两者中哪个能更快计划好会议，或多快能定好机票，或是花了多长时间，满意程度如何，不过可能会需要大量克隆人类行为的引导程序，去告诉它们怎么做。

吴恩达：听起来行为克隆就是监督学习，去模仿人在做的事，之后逐渐增强学习会能思考的更长远，这样总结合适吗？

Pieter Abbeel：我觉得是，单因为观察增强学习，从头开始，就很有意思，超级有趣，很少有比观察增强学习机器人从啥都不会到发明创造更好玩的事了，不过非常耗时，而且不总是那么安全。

吴恩达：太谢谢你了，这个采访太棒了，我很荣幸我们有这次聊天机会。

Pieter Abbeel：谢谢你邀请我，我非常感谢。

吴恩达采访 Andrej Karpathy

吴恩达：欢迎 Andrej，今天你能来我很高兴。

Andrej：是的，谢谢你的采访邀请。

吴恩达：人们已经很熟悉你在深度学习领域中的工作了，但不是每个人都知道你的个人故事，不如你就告诉我们，你是怎么一步步完成深度学习的这些工作的?

Andrej：是的，当然好，我第一次接触深度学习时，还是一个在多伦多大学的本科生，那时 Geoff Hinton 就在那里，他带一门深度学习课，那时候用的是受限玻尔兹曼机，在 MNIST 手写数字集上训练，但我真的很喜欢 Geoff 谈到训练网络的方式，他会用网络的智能，这些词，我那时想，这听起来，当我们训练这些数字时会发生奇迹，这是我第一次接触，不过那时候我没有了解很多细节，后来当我在不列颠哥伦比亚大学读硕士时，我上了一门[Nato Defreiter]教的课，还是机器学习，那时我第一次深入了解这些网络和相关知识，我觉得很有趣，当时我对人工智能非常感兴趣，所以我上了很多人工智能课，但是我看到的很多东西没那么令人满意，那时课程介绍了很多深度优先搜索，广度优先搜索，Alpha-Beta 剪枝各种方法，我那时不太懂，我并不满意，当我第一次在机器学习课程中见到神经网络时，我觉得这个词更技术一点，知名度没那么高，人工智能大家都能说上一两句，但机器学习就更小众一点，可以这么说吧，所以我对人工智能不满意，当我看到机器学习，我就想这才是我要花时间研究的 AI 这才是真正有趣的东西，就是这些原因让我开始走这条路，这几乎是全新的计

算范式，可以这么说，因为正常情况下是人类在编程，但是在这种情况下是优化程序自己写代码，所以当你建立了输入输出规范之后，你只需要给它喂一大堆例子，优化程序就自动编程，有时候它写出的程序比人工的还好，所以我觉得这只是一种非常新颖的编程思路，这就是让我感兴趣的地方。

吴恩达：然后通过你的工作，其中一件让你声名远扬的事是你是 ImageNet 分类竞赛的人类基准，这是怎么来的?

Andrej：所以基本上，他们的 ImageNet 比赛可以说成是计算机视觉领域的世界杯，不管人们是否注意这个基准和数字，我们的错误率也会随时间下降，对我来说，人类在这个评分标准上的位置并不清楚，我之前在 CIFAR-10 数据集上做了类似的小规模实验，我在 CIFAR-10 数据集中做的是，我观察了这些 32x32 的图像，我试图自己对他们进行分类，那时候只有十大类，所以很容易构造一个界面去人工分类，那是我自己的错误率大概是 6%，然后根据我实际见到的东西，一个具体任务有多难，我预测出最低能达到的错误率大概会是多少，好吧，我不记得具体数字是多少，我想大概是 10%，现在算法做到了 3%或 2%，或者某个变态的数值，那是我第一次做人类基线实验，非常有趣，我认为这是非常重要的，做这个基线的目的和你的视频里介绍的一样，我的意思是你真的希望这个数字能够表示人类的水平，这样我们就可以和机器学习算法比较，而对于 ImageNet 似乎存在一些分歧，这个基准到底有多重要，应该花多少精力去尽量得到较低的数字，我们甚至不了解人类自己在这个评分系统中的位置，所以我写了这个 JavaScript 界面，我给自己看图，然后 ImageNet 的问题在于，你不只有 10 个分类，你有 1000 个分类，这几乎就像一个用户界面挑战，显然我自己记不住 1000 多个分类，所以我应该怎么设计这个界面，让比赛公平一点，所以我把全部分类列表处理，然后给我自己看看各个分类的例子，所以对于每张图像，我大概浏览了 1000 多个类别，只是想看看，根据我在每个类别里看到的例子判断这个图像可能是什么，而且我认为这个练习本身就是非常有启发性的，我的意思是，我不明白为什么 ImageNet 有三分之一类别都是狗，狗的品种，所以我兴致勃勃地看着那个网络花了大量时间去处理狗，我想它的三分之一性能用来处理狗。我这个小实验做了一两个星期，我把其他所有任务都搁置了，我那时想这练习非常有趣，我最后得到了一个数字，我觉得一个人是不够的，我需要更多人参与到这个实验中来，我试图在实验室内组织人员做同样的事情，我想那时大家都不怎么愿意贡献，花上一两周时间来做这么痛苦的工作，就是坐五个小时，尝试分辨出这只狗的品种是什么，所以在这方面，我无法得到足够多的数据，我们大概估计了一下人类的表现，我觉得很有趣，然后就传开了，那时我还没觉得很明显，我只是想知道这个数字，但这很快变成了一个概念，

大家都很喜欢这个事实，然后就这样了，大家都开玩笑地说，我是那个作为基准的人，当然，我都笑死了，是啊。

吴恩达：当 DeepNet 软件超越了你的表现时，你有没有很惊讶?

Andrej：绝对，是的，绝对的，我的意思是，有时一张图真的很难看出是什么，那图就是小块黑白色，还有一些黑点，到处都是，我没看出来是什么，我只能猜测这属于 20 个类别之间，但网络就直接懂了，我不明白是怎么一回事，所以这里有点点超人类的意思了，但还有，我想网络非常擅长识别这些，地砖图案和纹理的统计规律，我想在这方面，网络比人类表现优秀毫不奇怪，它可以从大量图像中提取精细的统计特征，而在许多情况下，我很惊讶，因为有些图像需要识字，图片有时就是一个瓶子，你看不出来是什么，但它上面有文本，告诉你它是什么，作为人类，我可以阅读文字，这没问题，但网络必须自己学习读取信息来识别物体，因为单看图像并不明显。

吴恩达：还有一件事让你声名远扬，深度学习社区一直很感激你的贡献，就是你教了 CS231n 课程，并把它放到网上，可以告诉我具体的经过吗?

Andrej：是的，当然了，所以我有个强烈的感觉，这种技术是革命性的，很多人都希望能用上，这几乎像一把锤子，我想做的是，我那时能够随意把这把锤子交给很多人，我觉得这种工作很有吸引力，从博士生的角度来看，不太建议做这种事，因为你会把你的研究搁置一边，我的意思是说，这占用了我 120%时间，我必须将所有研究放一边，我是说，这门课我带过两次，每次都要 4 个月时间，所以时间基本上是花在课上，所以从这个角度来看不太

建议，但这基本上是我的博士阶段的亮点，这与研究甚至没有关系，我认为教一门课绝对是我博士生的亮点，只要看到学生，看到他们真的很兴奋，这门课和一般的不同，通常，课程里讲的内容是 19 世纪发现的，这些经典知识，但这样一门课，我可以介绍一周前刚发表的论文，甚至昨天刚发表的，这些都是前沿研究，想本科生，还有其他学生，真的很喜欢这门课贴近前沿的特点，他们发现他们是可以理解到前沿的，这不是核物理或火箭科学，你只需要会微积分，代数，你实际上就能理解所有背后的原理，我想这个事实如此强大，事实上这个领域日新月异，学生们就会觉得每天都处于时代浪潮的前端，我想这就是大家那么喜欢这门课的原因。

吴恩达：而且你真的帮助了很多人，送出去了很多锤子，是啊，作为一个研究深度学习，有些时日的研究员，这个领域还在迅速发展，我想知道，你自己的想法是怎样的，这么多年来你对深度学习的理解有何改变?

Andrej：是的，基本上当年我见到的是受限玻尔兹曼机处理这些手写数字数据，那时我还不知道这种技术会被大规模应用，不知道这个领域有多重要，还有，当我开始研究计算机视觉，卷积网络时，这些概念都已经存在，但它们并不像是计算机视觉界很快就会使用的东西，那时人们的看法是，这些处理小案例不错，但无法推广到更大的图像，这种认识错到极端了。[笑]所以基本上，我很惊讶现在这个技术到处都在用，结果非常好，我说这是最大的惊喜，而且还不仅如此，它在一个地方表现很好，比如 ImageNet，但另一方面，没有人预计到它的趋势，至少我自己没预计到，就是你可以把这些预先训练好的网络迁移到其他领域，你可以在任意其他任务中精细调校网络，因为现在你不只解决了 ImageNet 问题，而且你需要数百万个例子，网络变成了非常普适的特征提取器，而且这是我的第二个想法，我觉得更少人预计到了这个发展，还有这些论文，它们就在这里，人们在计算机视觉里的一切努力，场景分类，动作识别，对象识别，基本属性等等，人们只需要通过微调网络就把每个任务都解决了，所以对我来说是非常意外的。

吴恩达：是的，我想监督学习在媒体上很热门，但是，然而预先训练微调或迁移学习，其实都效果拔群，但这些领域媒体报道更少一些。

Andrej：对的，就是这样，是的，我觉得其中一个进展不大的方向是无监督学习，被寄予了太多希望，我认为这才是真正吸引，在 2007 年左右吸引很多研究人员进入了这个领域的概念，但我觉得那个梦想还没被实现，还有令我意外的一方面是，监督学习竟然效果这么好，而无监督学习，它还是处于很原始的状态，怎么利用它，或者怎么让它达到实用，还不太明显，即使很多人还是对它的未来深信不疑，我说在这个领域，可以用这个词。

吴恩达：所以我知道你是其中一个，一直在思考 AI 远期未来的研究员，你想分享你的想法吗?

Andrej：所以我最后花了差不多一年半，在 OpenAI 这里思考这些话题，在我看来，这个领域会分成两条轨迹，一边是应用 AI 就是搭建这些神经网络，训练它们，主要用监督学习，有可能用无监督学习，然后慢慢提升性能，比如说提高图像识别率之类的；另一个方向是，更一般的人工智能方向，就是如何让神经网络变成一个完全动态的系统，可以思考，有语言能力，可以做人类能做的所有事情，并以这种方式获得智能，我认为一直到很有趣的地方是 例如在计算机视觉中，我们一开始研究的方向，我想是错误的，那时我们试图把它分解成不同的部分，我们就像是，人可以识别人，可以识别场景，人可以识别物体，所以我们就开始研究人类能做的各种事情，一旦做出来了，就分成各种不同的子领域了，一旦我们有了这些独立的系统，我们再开始把它们组装起来，我觉得这种做法是错误的，我们已经见到历史上这么做结果如何，我想这里还有其他类似的事情正在发展，很可能是更高水平的 AI，所以人类会问问题，会做计划，会做实验来了解世界运作的规律，或者和其他人交谈，我们就发明了语言，人们试图通过功能来区分各种能力，然后复制每一种能力，把它们放到一起组成某种机械大脑，我觉得这个方法论是错的，我更感兴趣的领域，不是这种分解的，还原论的手段，而是希望构建一种全面动态的神经网络系统，这样你一直处理的是完整的代理人程序。

那么问题在于，你如何构思目标去优化权重，优化构成大脑的各种权重，才能得到真正的智能行为?所以这是 OpenAI 里我一直在想的很多东西，我认为有很多不同的方式，人们在这个问题上也有很多思考，，例如，在监督学习方向，我在网上发了这篇文章，这不是一篇文章，而是我写的一个小故事，这个小故事尝试构想出一个虚拟的世界，如果我们只通过扩大监督学习规模来逼近这个 AGI，我们知道这是可行的，然后得到像这样的东西，比如亚马逊土耳其机器人，人们可以接触不同机器人，让它们完成各种任务，然后我们在这个基础上训练，把它看成是模仿人类的监督学习数据集，这样的东西会具体是什么样的，所以这里还有其他方向，比如基于算法信息理论的无监督学习，如 AIXI，或者构成人工生命，看起来更像人工进化的东西，所以这就是我花时间思考很多的事情，我已经得到正确答案了，但我不愿意在这里说。

吴恩达：至少我们可以通过阅读你的博文来了解更多信息。

Andrej：是的，当然了。

吴恩达：你今天已经提出了很多建议，还有很多人想进入 AI 和深度学习领域，对于这

些人来说，你有什么建议呢?

Andrej：是的，当然了，我想人们在讨论 CS231n 的时候，为什么他们认为这是一个非常有用的课程，我听到最多的是，人们很喜欢我会一直讲到最底层的技术细节，他们要调用的不是一个库，而是可以看到底层代码，看到一切是怎么实现的，然后，他们自己去实现各大部分，所以你必须接触到最底层，知道一切程序背后的原理，不要随便抽象化，你必须充分了解全栈，了解整个流程，当我学习这些内容时，我发现这样学，学到的东西最多，就是你自己从零开始去实现，这是很重要的，就是这部分学习性价比最高，从理解领域知识方面来看，所以我自己写库，这个库叫 ConvNetJS，它是用 Javascript 写的，可以实现卷积神经网络，那是我学习后向传播的方法，我一直建议别人不要一开始就用 TensorFlow 之类的东西，一旦你自己写出了最底层的代码之后，你可以用，因为你知道所有背后的原理，这样你就很放心，现在就可以使用这样的框架，可以帮你省去一点细节功夫，，但你自己必须知道背后的所有原理，所以这是帮助我最多的东西，这是人们在上 CS231n 课程时最感兴趣的东西，所以我建议很多人这么做。

吴恩达：不是直接跑神经网络，让一切自然发生。

Andrej：是的，在某些层的序列中，我知道当我加入一些 dropout 层，可以让它表现更好，但这不是你想要的在这种情况下， 你会无法有效调试，你不能有效地改进模型。

吴恩达：是的， 这个答案让我想起我们的 deeplearning.ai 课程，一开始先用几周介绍 Python 编程，然后再..。

Andrej：是的 这样很好。

吴恩达：非常感谢你来到这里分享你的见解和建议，在深度学习世界中，你已经是很多人的偶像了，我真的很高兴，非常感谢你今天可以接受采访。

Andrej：是的 谢谢你邀请我。

深度学习符号指南（原课程翻译）

本章讨论深度学习的标准的数学符号。

1).常用的定义:

上标(i)代表第 i 个训练样本， 而上标$[l]$代表第 l 层。

数量:

m ：数据集的样本数

n_x ：输入大小

n_y ：输出大小 (或者类别数)

$n_h^{[l]}$ ：第 l 个隐藏层的数量，在一个 for 循环中，一般这样定义：$n_x = n_h^{[0]}$ ，并且:

n_y=$n_h^{[层数+1]}$

L ：网络的层数

对象：

$X \in \mathbb{R}^{n_x \times m}$ 代表输入的矩阵

$x^{(i)} \in \mathbb{R}^{n_x}$ 代表第 i 个样本的列向量

$Y \in \mathbb{R}^{n_y \times m}$ 是一个矩阵的标签

$y^{(i)} \in \mathbb{R}^{n_y}$ 是第 i 样本的输出标签

$W^{[l]} \in \mathbb{R}^{number\ of\ units\ in\ next\ layer \times number\ of\ units\ in\ the\ previous\ layer}$ 代表权重矩阵

上标 $[l]$ 代表第几层

$b^{[l]} \in \mathbb{R}^{number\ of\ units\ in\ next\ layer}$ 代表第 i 层的误差向量

$\hat{y} \in \mathbb{R}^{n_y}$ 是一个预测的输出向量. 它也可以用 $a^{[L]}$表示，而 L 代表网络的层数。

常用正向传播方程示例：

$a = g^{[l]}\left(W_x x^{(i)} + b_1\right) = g^{[l]}\left(z_1\right)$ ，其中 g[l]代表第 i 层的激活函数。

$\hat{y}^{(i)} = softmax(W_h h + b_2)$

一般激活公式：

$a_j^{[l]} = g^{[l]}\left(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}\right) = g^{[l]}\left(z_j^{[l]}\right)$

$J\left(x, W, b, y\right)$ 或者 $J\left(\hat{y}, y\right)$ 代表代价函数。

代价函数举例：

$J_{CE}\left(\hat{y}, y\right) = -\sum_{i=0}^{m} y^{(i)} \log \hat{y}^{(i)}$

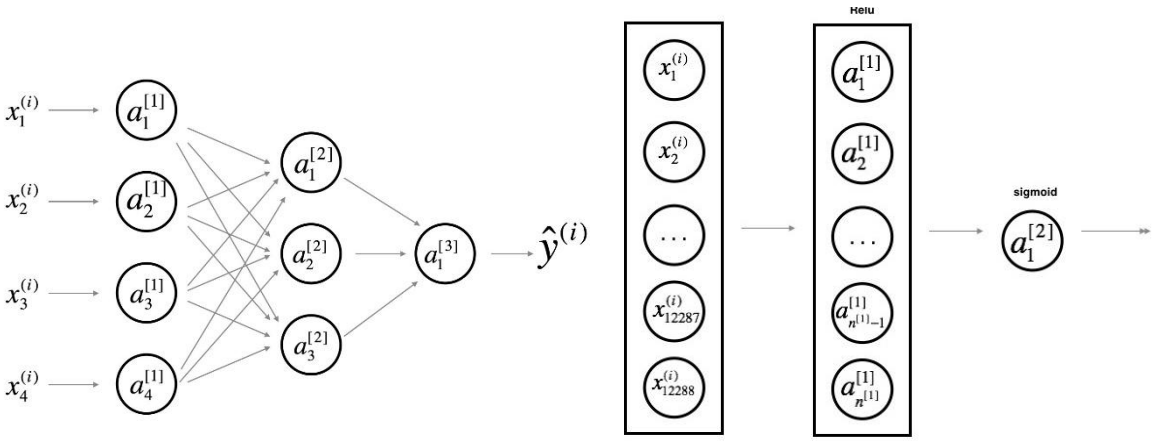
$J_1\left(\hat{y}, y\right) = \sum_{i=0}^{m} \left| y^{(i)} - \hat{y}^{(i)} \right|$

2). 深度学习的表示:

节点：代表输入、激活或者输出。

边：代表权重或者误差

下图是一般标准的深度学习的图例：



左图: 详细的网络：常用于神经网络的表示。 为了更好的审美，我们省略了一些在边上的参数的细节(如 $w_{ij}^{[l]}$ 和 $b_i^{[l]}$ 等)。

右图：简化网络：两层神经网络的更简单的表示。

以上两图都是等效的。

常用的数学公式

一元函数微分学

内容	对应公式、定理、概念
导数和微分的概念左右导数导数的几何意义和物理意义	**1**导数定义：$f'(x_0)=\lim\limits_{\triangle x\to 0}\dfrac{f(x_0+\triangle x)-f(x_0)}{\triangle x}$ （**1**） **或** $f'(x_0)=\lim\limits_{x\to x_0}\dfrac{f(x)-f(x_0)}{x-x_0}$ （**2**） 2 函数 $f(x)$ 在 x_0 处的左、右导数分别定义为： 左导数： $f'_-(x_0)=\lim\limits_{\Delta x\to 0^-}\dfrac{f(x_0+\Delta x)-f(x_0)}{\Delta x}=\lim\limits_{x\to x_0^-}\dfrac{f(x)-f(x_0)}{x-x_0},(x=x_0+\Delta x)$ 右导数： $f'_+(x_0)=\lim\limits_{\Delta x\to 0^+}\dfrac{f(x_0+\Delta x)-f(x_0)}{\Delta x}=\lim\limits_{x\to x_0^+}\dfrac{f(x)-f(x_0)}{x-x_0}$
函数的可导性与连续性之间的关系，平面曲线的切线和法线	Th1: 函数 $f(x)$ 在 x_0 处可微 $\Leftrightarrow$ $f(x)$ 在 x_0 处可导 Th2: 若函数 $y=f(x)$ 在点 x_0 处可导，则 $y=f(x)$ 在点 x_0 处连续，反之则不成立.即函数连续不一定可导. Th3: $f'(x_0)$ 存在 $\Leftrightarrow$ $f'_-(x_0)=f'_+(x_0)$ 设函数$f(x)$在$x=x_0$处可导，则$f(x)$在$M(x_0,y_0)$处的 切线方程：$y-y_0=f'(x_0)(x-x_0)$ 法线方程：$y-y_0=-\dfrac{1}{f'(x_0)}(x-x_0),f'(x_0)\neq 0.$
导数和微分的四则运算，初等函数的导数，	四则运算法则:设函数$u=u(x)$，$v=v(x)$在点x可导则 (1) $(u\pm v)'=u'\pm v'$ $d(u\pm v)=du\pm dv$ (2) $(uv)'=uv'+vu'$ $d(uv)=udv+vdu$ (3) $(\dfrac{u}{v})'=\dfrac{vu'-uv'}{v^2}(v\neq 0)$ $d(\dfrac{u}{v})=\dfrac{vdu-udv}{v^2}$ 基本导数与微分表 (1) $y=c$（常数） $y'=0$ $dy=0$ (2) $y=x^\alpha$（α 为实数） $y'=\alpha x^{\alpha-1}$ $dy=\alpha x^{\alpha-1}dx$ (3) $y=a^x$ $y'=a^x\ln a$ $dy=a^x\ln a dx$ 特例 $(e^x)'=e^x$ $d(e^x)=e^x dx$ (4) $y'=\dfrac{1}{x\ln a}$ $dy=\dfrac{1}{x\ln a}dx$ 特例 $y=\ln x$ $(\ln x)'=\dfrac{1}{x}$ $d(\ln x)=\dfrac{1}{x}dx$ (5) $y=\sin x$ $y'=\cos x$ $d(\sin x)=\cos x dx$ (6) $y=\cos x$ $y'=-\sin x$ $d(\cos x)=-\sin x dx$

	(7) $y=\tan x \quad y'=\frac{1}{\cos^2 x}=\sec^2 x \quad d(\tan x)=\sec^2 x dx$ (8) $y=\cot x \quad y'=-\frac{1}{\sin^2 x}=-\csc^2 x \quad d(\cot x)=-\csc^2 x dx$ (9) $y=\sec x \quad y'=\sec x\tan x \quad d(\sec x)=\sec x\tan x dx$ (10) $y=\csc x \quad y'=-\csc x\cot x \quad d(\csc x)=-\csc x\cot x dx$ (11) $y=\arcsin x \quad y'=\frac{1}{\sqrt{1-x^2}} \quad d(\arcsin x)=\frac{1}{\sqrt{1-x^2}}dx$ (12) $y=\arccos x \quad y'=-\frac{1}{\sqrt{1-x^2}} \quad d(\arccos x)=-\frac{1}{\sqrt{1-x^2}}dx$ (13) $y=\arctan x \quad y'=\frac{1}{1+x^2} \quad d(\arctan x)=\frac{1}{1+x^2}dx$ (14) $y=\operatorname{arc}\cot x \quad y'=-\frac{1}{1+x^2} \quad d(\operatorname{arc}\cot x)=-\frac{1}{1+x^2}dx$ (15) $y=shx \quad y'=chx \quad d(shx)=chxdx$ (16) $y=chx \quad y'=shx \quad d(chx)=shxdx$
复合函数，反函数，隐函数以及参数方程所确定的函数的微分法，	1 反函数的运算法则：设 $y=f(x)$ 在点 x 的某邻域内单调连续，在点 x 处可导且 $f'(x)\neq 0$，则其反函数在点 x 所对应的 y 处可导，并且有 $\frac{dy}{dx}=\frac{1}{\frac{dx}{dy}}$ 2 复合函数的运算法则:若 $\mu=\varphi(x)$ 在点 x 可导,而 $y=f(\mu)$ 在对应点 μ（$\mu=\varphi(x)$）可导,则复合函数 $y=f(\varphi(x))$ 在点 x 可导,且 $y'=f'(\mu)\cdot\varphi'(x)$ 3 隐函数导数 $\frac{dy}{dx}$ 的求法一般有三种方法： (1)方程两边对 X 求导，要记住 y 是 x 的函数，则 y 的函数是 x 的复合函数.例如 $\frac{1}{y}$，y^2，$\ln y$，e^y 等均是 x 的复合函数. 对 X 求导应按复合函数连锁法则做. (2)公式法.由 $F(x,y)=0$ 知 $\frac{dy}{dx}=-\frac{F'_x(x,y)}{F'_y(x,y)}$,其中，$F'_x(x,y)$，$F'_y(x,y)$ 分别表示 $F(x,y)$ 对 x 和 y 的偏导数 (3)利用微分形式不变性
高阶导数，一阶微分形式的不变性，	常用高阶导数公式 （1）$(a^x)^{(n)}=a^x\ln^n a \quad (a>0) \qquad (e^x)^{(n)}=e^x$ （2）$(\sin kx)^{(n)}=k^n\sin(kx+n\cdot\frac{\pi}{2})$ （3）$(\cos kx)^{(n)}=k^n\cos(kx+n\cdot\frac{\pi}{2})$

	（4） $(x^m)^{(n)}=m(m-1)\cdots(m-n+1)x^{m-n}$ （5） $(\ln x)^{(n)}=(-1)^{(n-1)}\frac{(n-1)!}{x^n}$ （6）莱布尼兹公式：若 $u(x),v(x)$ 均 n 阶可导，则 $(uv)^{(n)}=\sum_{i=0}^{n}c_n^i u^{(i)}v^{(n-i)}$ ，其中 $u^{(0)}=u$ ， $v^{(0)}=v$
微分中值定理，必达法则，泰勒公式	Th1(费马定理)若函数 $f(x)$ 满足条件： (1)函数 $f(x)$ 在 X_0 的某邻域内有定义，并且在此邻域内恒有 $f(x)\le f(x_0)$ 或 $f(x)\ge f(x_0)$， (2) $f(x)$ 在 x_0 处可导,则有 $f'(x_0)=0$ Th2 (罗尔定理) 设函数 $f(x)$ 满足条件： (1)在闭区间 $[a,b]$ 上连续； (2)在 (a,b) 内可导，则在 (a,b) 内∃一个 ξ，使 $f'(\xi)=0$ Th3 (拉格朗日中值定理) 设函数 $f(x)$ 满足条件： (1) 在 $[a,b]$ 上连续；(2) 在 (a,b) 内可导；则在 (a,b) 内∃一个 ξ，使 $\frac{f(b)-f(a)}{b-a}=f'(\xi)$ Th4 (柯西中值定理) 设函数 $f(x)$， $g(x)$ 满足条件： (1)在 $[a,b]$ 上连续；(2)在 (a,b) 内可导且 $f'(x)$， $g'(x)$ 均存在，且 $g'(x)\neq 0$ 则在 (a,b) 内∃一个 ξ，使 $\frac{f(b)-f(a)}{g(b)-g(a)}=\frac{f'(\xi)}{g'(\xi)}$ 洛必达法则： 法则Ⅰ ($\frac{0}{0}$型)设函数 $f(x),g(x)$ 满足条件： $\lim_{x\to x_0}f(x)=0,\lim_{x\to x_0}g(x)=0$； $f(x),g(x)$ 在 x_0 的邻域内可导 (在 x_0 处可除外)且 $g'(x)\neq 0$；$\lim_{x\to x_0}\frac{f'(x)}{g'(x)}$ 存在(或 ∞).则 $\lim_{x\to x_0}\frac{f(x)}{g(x)}=\lim_{x\to x_0}\frac{f'(x)}{g'(x)}.$ 法则Ⅰ′ ($\frac{0}{0}$型)设函数 $f(x),g(x)$ 满足条件： $\lim_{x\to\infty}f(x)=0,\lim_{x\to\infty}g(x)=0$；∃一个 $X>0$，当 $\lvert x\rvert>X$ 时， $f(x),g(x)$ 可导,且 $g'(x)\neq 0$；$\lim_{x\to x_0}\frac{f'(x)}{g'(x)}$ 存在(或 ∞).则 $\lim_{x\to x_0}\frac{f(x)}{g(x)}=\lim_{x\to x_0}\frac{f'(x)}{g'(x)}.$ 法则Ⅱ ($\frac{\infty}{\infty}$ 型) 设函数 $f(x),g(x)$ 满足条件：

$\lim_{x\to x_0} f(x)=\infty, \lim_{x\to x_0} g(x)=\infty$； $f(x), g(x)$ 在 x_0 的邻域内可导(在 x_0 处可除外)且 $g'(x)\neq 0$；$\lim_{x\to x_0}\frac{f'(x)}{g'(x)}$ 存在(或 ∞).则 $\lim_{x\to x_0}\frac{f(x)}{g(x)}=\lim_{x\to x_0}\frac{f'(x)}{g'(x)}$.同理法则 II′ ($\frac{\infty}{\infty}$ 型)仿法则 I′ 可写出

泰勒公式: 设函数 $f(x)$ 在点 x_0 处的某邻域内具有 $n+1$ 阶导数，则对该邻域内异于 x_0 的任意点 x，在 x_0 与 x 之间至少∃一个 ξ，使得

$$f(x)=f(x_0)+f'(x_0)(x-x_0)+\frac{1}{2!}f''(x_0)(x-x_0)^2+\cdots$$
$$+\frac{f^{(n)}(x_0)}{n!}(x-x_0)^n+R_n(x)$$

其中 $R_n(x)=\frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}$ 称为 $f(x)$ 在点 x_0 处的 n 阶泰勒余项.令 $x_0=0$，则 n 阶泰勒公式

$$f(x)=f(0)+f'(0)x+\frac{1}{2!}f''(0)x^2+\cdots+\frac{f^{(n)}(0)}{n!}x^n+R_n(x)\cdots\cdots(1)$$

其中 $R_n(x)=\frac{f^{(n+1)}(\xi)}{(n+1)!}x^{n+1}$，$\xi$ 在 0 与 x 之间.(1)式称为麦克劳林公式

常用五种函数在 $x_0=0$ 处的泰勒公式

$$e^x=1+x+\frac{1}{2!}x^2+\cdots+\frac{1}{n!}x^n+\frac{x^{n+1}}{(n+1)!}e^{\xi}$$

或 $$=1+x+\frac{1}{2!}x^2+\cdots+\frac{1}{n!}x^n+o(x^n)$$

$$\sin x=x-\frac{1}{3!}x^3+\cdots+\frac{x^n}{n!}\sin\frac{n\pi}{2}+\frac{x^{n+1}}{(n+1)!}\sin(\xi+\frac{n+1}{2}\pi)$$

或 $$=x-\frac{1}{3!}x^3+\cdots+\frac{x^n}{n!}\sin\frac{n\pi}{2}+o(x^n)$$

$$\cos x=1-\frac{1}{2!}x^2+\cdots+\frac{x^n}{n!}\cos\frac{n\pi}{2}+\frac{x^{n+1}}{(n+1)!}\cos(\xi+\frac{n+1}{2}\pi)$$

或 $$=1-\frac{1}{2!}x^2+\cdots+\frac{x^n}{n!}\cos\frac{n\pi}{2}+o(x^n)$$

$$\ln(1+x)=x-\frac{1}{2}x^2+\frac{1}{3}x^3-\cdots+(-1)^{n-1}\frac{x^n}{n}+\frac{(-1)^n x^{n+1}}{(n+1)(1+\xi)^{n+1}}$$

或 $$=x-\frac{1}{2}x^2+\frac{1}{3}x^3-\cdots+(-1)^{n-1}\frac{x^n}{n}+o(x^n)$$

$$(1+x)^m=1+mx+\frac{m(m-1)}{2!}x^2+\cdots+\frac{m(m-1)\cdots(m-n+1)}{n!}x^n$$

	$+\frac{m(m-1)\cdots(m-n+1)}{(n+1)!}x^{n+1}(1+\xi)^{m-n-1}$ 或 $(1+x)^m=1+mx+\frac{m(m-1)}{2!}x^2+\cdots$ $+\frac{m(m-1)\cdots(m-n+1)}{n!}x^n+o(x^n)$
函数单调性的判别，函数的极值，函数的图形的凹凸性，拐点及渐近线，用函数图形描绘函数最大值和最小值，	1 函数单调性的判断： Th1 设函数 $f(x)$ 在 (a,b) 区间内可导，如果对 $\forall x\in(a,b)$，都有 $f'(x)>0$（或 $f'(x)<0$），则函数 $f(x)$ 在 (a,b) 内是单调增加的（或单调减少） Th2 （取极值的必要条件）设函数 $f(x)$ 在 x_0 处可导，且在 x_0 处取极值，则 $f'(x_0)=0$. Th3 （取极值的第一充分条件）设函数 $f(x)$ 在 x_0 的某一邻域内可微，且 $f'(x_0)=0$（或 $f(x)$ 在 x_0 处连续，但 $f'(x_0)$ 不存在.） (1)若当 x 经过 x_0 时，$f'(x)$ 由“+”变“-”，则 $f(x_0)$ 为极大值； (2)若当 x 经过 x_0 时，$f'(x)$ 由“-”变“+”，则 $f(x_0)$ 为极小值； (3)若 $f'(x)$ 经过 $x=x_0$ 的两侧不变号，则 $f(x_0)$ 不是极值. Th4 (取极值的第二充分条件)设 $f(x)$ 在点 x_0 处有 $f''(x)\neq 0$，且 $f'(x_0)=0$，则当 $f''(x_0)<0$ 时，$f(x_0)$ 为极大值； 当 $f''(x_0)>0$ 时，$f(x_0)$ 为极小值. 注：如果 $f''(x_0)=0$，此方法失效. 2 渐近线的求法： (1)水平渐近线　若 $\lim\limits_{x\to+\infty}f(x)=b$，或 $\lim\limits_{x\to-\infty}f(x)=b$，则 $y=b$ 称为函数 $y=f(x)$ 的水平渐近线. (2)铅直渐近线　若 $\lim\limits_{x\to x_0^-}f(x)=\infty$，或 $\lim\limits_{x\to x_0^+}f(x)=\infty$，则 $x=x_0$ 称为 $y=f(x)$ 的铅直渐近线. (3)斜渐近线　若 $a=\lim\limits_{x\to\infty}\frac{f(x)}{x}$，$b=\lim\limits_{x\to\infty}[f(x)-ax]$，则 $y=ax+b$ 称为 $y=f(x)$ 的斜渐近线 3 函数凹凸性的判断： Th1 (凹凸性的判别定理）若在 I 上 $f''(x)<0$（或 $f''(x)>0$）， 则 $f(x)$ 在 I 上是凸的（或凹的）.

	Th2 (拐点的判别定理 1)若在 x_0 处 $f''(x)=0$，（或 $f''(x)$ 不存在），当 x 变动经过 x_0 时，$f''(x)$ 变号，则 $(x_0, f(x_0))$ 为拐点. Th3 (拐点的判别定理 2)设 $f(x)$ 在 x_0 点的某邻域内有三阶导数，且 $f''(x)=0$，$f'''(x)\neq 0$，则 $(x_0, f(x_0))$ 为拐点
弧微分，曲率的概念，曲率半径	1. 弧微分：$dS=\sqrt{1+y'^2}dx.$ 2. 曲率：曲线 $y=f(x)$ 在点 (x,y) 处的曲率 $k=\frac{\lvert y''\rvert}{(1+y'^2)^{\frac{3}{2}}}.$ 对于参数方程 $\begin{cases}x=\varphi(t)\\ y=\psi(t)\end{cases}$，$k=\frac{\lvert\varphi'(t)\psi''(t)-\varphi''(t)\psi'(t)\rvert}{[\varphi'^2(t)+\psi'^2(t)]^{\frac{3}{2}}}.$ 3.曲率半径：曲线在点 M 处的曲率 $k(k\neq 0)$ 与曲线在点 M 处的曲率半径 ρ 有如下关系：$\rho=\frac{1}{k}.$

(一) 随机事件和概率

内容	对应概念、定理、公式
随机事件与样本空间，事件的关系与运算，完全事件组	1 事件的关系与运算 (1)子事件：$A\subset B$，若 A 发生，则 B 发生. (2)相等事件：A=B，即 $A\subset B$，且 $B\subset A$. (3)和事件：$A\cup B$（或 A+B），A 与 B 中至少有一个发生. (4)差事件：A-B，A 发生但 B 不发生. (5)积事件：$A\cap B$（或 AB），A 与 B 同时发生. (6)互斥事件（互不相容）：$A\cap B=\varnothing$. (7)互逆事件（对立事件）： $A\cap B=\varnothing,$且$A\cup B=\Omega,$记$A=\bar{B}$或$B=\bar{A}$ 2 运算律： (1)交换律：$A\cup B=B\cup A$，$A\cap B=B\cap A$ (2)结合律：$(A\cup B)\cup C=A\cup(B\cup C)$； $(A\cap B)\cap C=A\cap(B\cap C)$ (3)分配律：$(A\cup B)\cap C=(A\cap C)\cup(B\cap C)$ 3 德•摩根律：$\overline{A\cup B}=\bar{A}\cap\bar{B},\overline{A\cup B}=\bar{A}\cup\bar{B}$ 4 完全事件组：A_1，$A_2\cdots$，A_n，两两互斥，且和事件为必然事件，即 $A_i\cap A_j=\varnothing$，i≠j，$\bigcup_{i=1}^{n}=\Omega$。
概率的概念，概率的基本性	1 概率：事件发生的可能性大小的度量，其严格定义如下： 概率 $P(\bullet)$ 为定义在事件集合上的满足下面 3 个条件的函数：

质，古典概率，几何型概率	(1)对任何事件 A， $P(A)\geq 0$; (2)对必然事件 Ω， $P(\Omega)=1$; (3)对 A_1，$A_2,\cdots$，$A_n,\cdots$，若 $A_iA_j=\varnothing(i\neq j)$，则 $P(\bigcup_{i=1}^{\infty}A_i)=\sum_{i=1}^{\infty}P(A).$ 2 概率的基本性质 (1) $P(\bar{A})=1-P(A);$ (2) $P(A-B)=P(A)-P(AB);$ (3) $P(A\cup B)=P(A)+P(B)-P(AB);$ 特别， 当 $B\subset A$ 时， $P(A-B)=P(A)-P(B)$ 且 $P(B)\leq P(A)$； $P(A\cup B\cup C)=P(A)+P(B)+P(C)-P(AB)-P(BC)$ $-P(AC)+P(ABC);$ (4)若 $A_1,A_2,\cdots,A_n$ 两两互斥，则 $P(\bigcup_{i=1}^{n}A_i)=\sum_{i=1}^{n}(P(A_i).$ 3 古典型概率: 实验的所有结果只有有限个， 且每个结果发生的可能性相同，其概率计算公式： $P(A)=\frac{\text{事件}A\text{发生的基本事件数}}{\text{基本事件总数}}$ 4 几何型概率: 样本空间 Ω 为欧氏空间中的一个区域， 且每个样本点的出现具有等可能性，其概率计算公式： $P(A)=\frac{A\text{的度量（长度、面积、体积）}}{\Omega\text{的度量（长度、面积、体积）}}$
概率的基本公式，事件的独立性，独立重复试验	1 概率的基本公式: (1)条件概率: $P(B\mid A)=\frac{P(AB)}{P(A)}$，表示A发生的条件下，B发生的概率 (2)全概率公式： $P(A)=\sum_{i=1}^{n}P(A\mid B_i)P(B_i),B_iB_j=\varnothing,i\neq j,\bigcup_{i=1}^{n}B_i=\Omega.$ (3) Bayes 公式： $P(B_j\mid A)=\frac{P(A\mid B_j)P(B_j)}{\sum_{i=1}^{n}P(A\mid B_i)P(B_i)},j=1,2,\cdots,n$ 注：上述公式中事件 B_i 的个数可为可列个. (4)乘法公式： $P(A_1A_2)=P(A_1)P(A_2\mid A_1)=P(A_2)P(A_1\mid A_2)$ $P(A_1A_2\cdots A_n)=P(A_1)P(A_2\mid A_1)P(A_3\mid A_1A_2)\cdots P(A_n\mid A_1A_2\cdots A_{n-1})$

	2 事件的独立性 (1)A 与 B 相互独立 $\Leftrightarrow P(AB)=P(A)P(B)$ (2)A，B，C 两两独立 $\Leftrightarrow P(AB)=P(A)P(B);\ P(BC)=P(B)P(C);\quad P(AC)=P(A)P(C);$ (3)A，B，C 相互独立 $\Leftrightarrow P(AB)=P(A)P(B);\qquad P(BC)=P(B)P(C);$ $P(AC)=P(A)P(C);\qquad P(ABC)=P(A)P(B)P(C).$ 3 独立重复试验: 将某试验独立重复 n 次，若每次实验中事件 A 发生的概率为 p，则 n 次试验中 A 发生 k 次的概率为： $P(X=k)=C_n^k p^k(1-p)^{n-k}.$ **4 重要公式与结论** $(1)P(\bar{A})=1-P(A)$ $(2)P(A\cup B)=P(A)+P(B)-P(AB)$ $P(A\cup B\cup C)=P(A)+P(B)+P(C)-P(AB)-P(BC)$ $-P(AC)+P(ABC)$ $(3)P(A-B)=P(A)-P(AB)$ $(4)P(A\bar{B})=P(A)-P(AB),P(A)=P(AB)+P(A\bar{B}),$ $P(A\cup B)=P(A)+P(\bar{A}B)=P(AB)+P(A\bar{B})+P(\bar{A}B)$ (5)条件概率 $P(\bullet\mid B)$ 满足概率的所有性质， 例如：. $P(\bar{A}_1\mid B)=1-P(A_1\mid B)$ $P(A_1\cup A_2\mid B)=P(A_1\mid B)+P(A_2\mid B)-P(A_1A_2\mid B)$ $P(A_1A_2\mid B)=P(A_1\mid B)P(A_2\mid A_1B)$ (6)若 $A_1,A_2,\cdots,A_n$ 相互独立，则 $P(\bigcap_{i=1}^{n}A_i)=\prod_{i=1}^{n}P(A_i),$ $P(\bigcup_{i=1}^{n}A_i)=\prod_{i=1}^{n}(1-P(A_i))$ (7)互斥、互逆与独立性之间的关系： A 与 B 互逆$\Rightarrow$A 与 B 互斥，但反之不成立，A 与 B 互斥（或互逆）且均非零概率事件$\Rightarrow$A 与 B 不独立. (8)若 $A_1,A_2,\cdots,A_m,B_1,B_2,\cdots,B_n$ 相互独立，则 $f(A_1,A_2,\cdots,A_m)$ 与 $g(B_1,B_2,\cdots,B_n)$ 也相互独立，其中 $f(\bullet),g(\bullet)$ 分别表示对相应事件做任意事件运算后所得的事件，另外，概率为 1（或 0）的事件与任何事件相互独立.

(二) 随机变量及其概率分布

内容	对应公式、概念、定理
随机变量，随机变量的分	1 随机变量及概率分布: 取值带有随机性的变量，严格地说是定义在样本空间上，取值于实数的函数称为随机变量，概率分布通常指分布函数或分布律 2 分布函数的概念与性质

部函数的概念及其性质	定义： $F(x)=P(X\le x),-\infty<x<+\infty$ 性质：(1) $0\le F(x)\le 1$ (2) $F(x)$ 单调不减 (3)右连续 $F(x+0)=F(x)$ (4) $F(-\infty)=0,F(+\infty)=1$
离散型随机变量的概率分布，连续型随机变量的概率密度性质	1 离散型随机变量的概率分布 $P(X=x_i)=p_i,i=1,2,\cdots,n,\cdots \quad p_i\ge 0,\sum_{i=1}^{\infty}p_i=1$ **2** 连续型随机变量的概率密度 概率密度 $f(x)$;非负可积，且 (1) $f(x)\ge 0,$ (2) $\int_{-\infty}^{+\infty}f(x)dx=1$ (3) x为$f(x)$的连续点，则$f(x)=F'(x)$分布函数$F(x)=\int_{-\infty}^{x}f(t)dt$
常见随机变量的概率分布，随机变量函数的概率分布	**1 常见分布** **(1)** 0-1 分布： $P(X=k)=p^k(1-p)^{1-k},k=0,1$ (2) 二项分布 $B(n,p)$： $P(X=k)=C_n^k p^k(1-p)^{n-k},k=0,1,\cdots,n$ (3) Poisson 分布 $p(\lambda)$： $P(X=k)=\frac{\lambda^k}{k!}e^{-\lambda},\lambda>0,k=0,1,2\cdots$ (4) 均匀分布 U（a，b）： $f(x)=\begin{cases}\frac{1}{b-a},a<x<b\\0,\text{其他}\end{cases}$ (5) 正态分布 $N(\mu,\sigma^2)$: $\varphi(x)=\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}},\sigma>0,-\infty<x<+\infty$ (6)指数分布 $E(\lambda)$: $f(x)=\begin{cases}\lambda e^{-\lambda x},x>0,\lambda>0\\0,\text{其他}\end{cases}$ (7)几何分布 $G(p):P(X=k)=(1-p)^{k-1}p,0<p<1,k=1,2,\cdots.$ (8)超几何分布 $H(N,M,n):P(X=k)=\frac{C_M^k C_{N-M}^{n-k}}{C_N^n},\ k=0,1,\cdots,\min(n,M)$ 2 随机变量函数的概率分布 (1)离散型： $P(X=x_1)=p_i,Y=g(X)$ 则 $P(Y=y_j)=\sum_{g(x_i)=y_i}P(X=x_i)$ (2)连续型： $X\sim f_X(x),Y=g(x)$ 则 $F_y(y)=P(Y\le y)=P(g(X)\le y)=\int_{g(x)\le y}f_x(x)dx,$ $f_Y(y)=F'_Y(y)$

	3 重要公式与结论 $(1)X \sim N(0,1) \Rightarrow \varphi(0)=\frac{1}{\sqrt{2\pi}}, \Phi(0)=\frac{1}{2},$ $\Phi(-a)=P(X \le -a)=1-\Phi(a)$ $(2)X \sim N(\mu,\sigma^2) \Rightarrow \frac{X-\mu}{\sigma} \sim N(0,1)$且$P(X \le a)=\Phi(\frac{a-\mu}{\sigma})$ $(3)X \sim E(\lambda) \Rightarrow P(X>s+t \mid X>s)=P(X>t)$ $(4)X \sim G(p) \Rightarrow P(X=m+k \mid X>m)=P(X=k)$ (5)离散型随机变量的分布函数为阶梯间断函数；连续型随机变量的分布函数为连续函数，但不一定为处处可导函数. (6)存在既非离散也非连续型随机变量.

(三) 多维随机变量及其分布

内容	对应公式、概念、定理
多维随机变量及其分布，二维离散型随机变量的概率分布、边缘分布和条件分布	1 二维随机变量及其联合分布 由两个随机变量构成的随机向量（X，Y）， 联合分布为$F(x,y)=P(X \le x, Y \le y)$ 2 二维离散型随机变量的联合概率分布、边缘分布、条件分布(1)联合概率分布律 $P\{X=x_i, Y=y_j\}=p_{ij}; i,j=1,2,\cdots$ (1) 边缘分布律 $p_{i\cdot}=\sum_{j=1}^{\infty} p_{ij}, i=1,2,\cdots$ $p_{\cdot j}=\sum_{i}^{\infty} p_{ij}, j=1,2,\cdots$ (2) 条件分布律 $P\{X=x_i \mid Y=y_j\}=\frac{p_{ij}}{p_{\cdot j}}$ $P\{Y=y_j \mid X=x_i\}=\frac{p_{ij}}{p_{i\cdot}}$
二维连续性随机变量的概率密度、边缘概率密度和条件密度	1 联合概率密度 $f(x,y)$: (1) $f(x,y) \ge 0$ (2) $\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(x,y)dxdy=1$ 2 分布函数： $F(x,y)=\int_{-\infty}^{x}\int_{-\infty}^{y} f(u,v)dudv$ 3 边缘概率密度： $f_X(x)=\int_{-\infty}^{+\infty} f(x,y)dy \qquad f_Y(y)=\int_{-\infty}^{+\infty} f(x,y)dx$ 4 条件概率密度： $f_{X\mid Y}(x \mid y)=\frac{f(x,y)}{f_Y(y)} \qquad f_{Y\mid X}(y \mid x)=\frac{f(x,y)}{f_X(x)}$

随机变量的独立性和不相关性，常用二维随机变量的分布

1 常见二维随机变量的联合分布

(1)二维均匀分布：$(x,y)\sim U(D)$ ，$f(x,y)=\begin{cases}\dfrac{1}{S(D)},(x,y)\in D\\ 0,\text{其他}\end{cases}$

(2)二维正态分布：（X，Y）$\sim N(\mu_1,\ \mu_2,\ \sigma_1^2,\ \sigma_2^2,\ \rho)$

$$f(x,y)=\frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}}$$

- $$\exp\left\{\frac{-1}{2(1-\rho^2)}\left[\frac{(x-\mu_1)^2}{\sigma_1^2}-2\rho\frac{(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2}+\frac{(y-\mu_2)^2}{\sigma_2^2}\right]\right\}$$

2 随机变量的独立性和相关性

X 和 Y 的相互独立 $\Leftrightarrow F(x,y)=F_X(x)F_Y(y)$，

$\Leftrightarrow p_{ij}=p_{i\cdot}\cdot p_{\cdot j}$(离散型）$\Leftrightarrow f(x,y)=f_X(x)f_Y(y)$(连续型）

X 和 Y 的相关性：相关系数 $\rho_{XY}=0$ 时，称 X 和 Y 不相关，否则称 X 和 Y 相关

两个及两个以上随机变量简单函数的分布

1 两个随机变量简单函数的概率分布

(1)离散型：

$P(X=x_i,Y=y_i)=p_{ij},Z=g(X,Y)$则

$$P(Z=z_k)=P\{g(X,Y)=z_k\}=\sum_{g(x_i,y_i)=z_k}P(X=x_i,Y=y_j)$$

(2)连续型：

$(X,Y)\sim f(x,y),Z=g(X,Y)$则

$F_z(z)=P\{g(X,Y)\le z\}=\iint\limits_{g(x,y)\le z}f(x,y)dxdy$，$f_z(z)=F'_z(z)$

2 重要公式与结论

(1) 边缘密度公式：

$$f_X(x)=\int_{-\infty}^{+\infty}f(x,y)dy,\qquad f_Y(y)=\int_{-\infty}^{+\infty}f(x,y)dx.$$

(2) $P\{(X,Y)\in D\}=\iint\limits_D f(x,y)dxdy$

(3)若（X，Y）服从二维正态分布 $N(\mu_1,\mu_2,\sigma_1^2,\sigma_2^2,\rho)$ 则有

① $X\sim N(\mu_1,\sigma_1^2),Y\sim N(\mu_2,\sigma_2^2).$

②X 与 Y 相互独立 $\Leftrightarrow\rho=0$，即 X 与 Y 不相关.

③ $C_1X+C_2Y\sim N(C_1\mu_1+C_2\mu_2,C_1^2\sigma_1^2+C_2^2\sigma_2^2+2C_1C_2\sigma_1\sigma_2\rho).$

④X 关于 Y=y 的条件分布为：

$$N(\mu_1+\rho\frac{\sigma_1}{\sigma_2}(y-\mu_2),\sigma_1^2(1-\rho^2)).$$

⑤Y 关于 X=x 的条件分布为：

	$N(\mu_2+\rho\frac{\sigma_2}{\sigma_1}(x-\mu_1),\sigma_2^2(1-\rho^2)).$ (4)若 X 与 Y 独立，且分别服从 $N(\mu_1,\sigma_1^2),\ N(\mu_1,\sigma_2^2),$ 则 $(X,Y)\sim N(\mu_1,\mu_2,\sigma_1^2,\sigma_2^2,0),$ $C_1X+C_2Y\sim N(C_1\mu_1+C_2\mu_2,C_1^2\sigma_1^2+C_2^2\sigma_2^2).$ (5)若 X 与 Y 相互独立， $f(x)$和$g(x)$ 为连续函数， 则 $f(X)$与$g(Y)$ 也相互独立.

(四)　随机变量的数字特征

内容	对应概念、定义、定理、公式
随机变量的数学期望（均值）、方差和标准差及其性质	1 数学期望 离散型： $P\{X=x_i\}=p_i,E(X)=\sum_i x_i p_i$ ； 连续型： $X\sim f(x),E(X)=\int_{-\infty}^{+\infty}xf(x)dx$ 性质： (1) $E(C)=C,E[E(X)]=E(X)$ (2) $E(C_1X+C_2Y)=C_1E(X)+C_2E(Y)$ (3)若 X 和 Y 独立，则 $E(XY)=E(X)E(Y)$ (4) $[E(XY)]^2\le E(X^2)E(Y^2)$ 2 方差： $D(X)=E[X-E(X)]^2=E(X^2)-[E(X)]^2$ 3 标准差： $\sqrt{D(X)}$ ， 4 离散型： $D(X)=\sum_i[x_i-E(X)]^2\ p_i$ 5 连续型： $D(X)=\int_{-\infty}^{+\infty}[x-E(X)]^2f(x)dx$ 性质： (1) $D(C)=0,D[E(X)]=0,D[D(X)]=0$ (2)X 与 Y 相互独立，则 $D(X\pm Y)=D(X)+D(Y)$ (3) $D(C_1X+C_2)=C_1^2D(X)$ (4)一般有 $D(X\pm Y)=D(X)+D(Y)\pm 2Cov(X,Y)=D(X)+D(Y)\pm 2\rho\sqrt{D(X)}\sqrt{D(Y)}$ (5) $D(X)<E(X-C)^2,C\ne E(X)$ (6) $D(X)=0\Leftrightarrow P\{X=C\}=1$
随机变量函数的数	1 随机变量函数的数学期望 (1)对于函数 $Y=g(x)$

学期望，矩、协方差，相关系数的数字特征

X 为离散型：$P\{X=x_i\}=p_i, E(Y)=\sum_i g(x_i)p_i$ ；X 为连续型：

$X\sim f(x), E(Y)=\int_{-\infty}^{+\infty} g(x)f(x)dx$

(2) $Z=g(X,Y)$；$(X,Y)\sim P\{X=x_i,Y=y_j\}=p_{ij}$；

$$E(Z)=\sum_i\sum_j g(x_i,y_j)p_{ij}$$

$$(X,Y)\sim f(x,y); E(Z)=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} g(x,y)f(x,y)dxdy$$

2 协方差 $Cov(X,Y)=E[(X-E(X)(Y-E(Y))]$

3 相关系数 $\rho_{XY}=\frac{Cov(X,Y)}{\sqrt{D(X)}\sqrt{D(Y)}}$，k 阶原点矩 $E(X^k)$；

k 阶中心矩 $E\{[X-E(X)]^k\}$

性质：

(1) $Cov(X,Y)=Cov(Y,X)$

(2) $Cov(aX,bY)=abCov(Y,X)$

(3) $Cov(X_1+X_2,Y)=Cov(X_1,Y)+Cov(X_2,Y)$

(4) $|\rho(X,Y)|\le 1$

(5) $\rho(X,Y)=1\Leftrightarrow P(Y=aX+b)=1,$其中$a>0$

$\rho(X,Y)=-1\Leftrightarrow P(Y=aX+b)=1,$其中$a<0$

4 重要公式与结论

（1）$D(X)=E(X^2)-E^2(X)$

（2）$Cov(X,Y)=E(XY)-E(X)E(Y)$

（3）$|\rho(X,Y)|\le 1,$且

$\rho(X,Y)=1\Leftrightarrow P(Y=aX+b)=1,$其中$a>0$

$\rho(X,Y)=-1\Leftrightarrow P(Y=aX+b)=1,$其中$a<0$

（4）下面 5 个条件互为充要条件：

$\rho(X,Y)=0$

$\Leftrightarrow Cov(X,Y)=0$

$\Leftrightarrow E(X,Y)=E(X)E(Y)$

$\Leftrightarrow D(X+Y)=D(X)+D(Y)$

$\Leftrightarrow D(X-Y)=D(X)+D(Y)$

注：X 与 Y 独立为上述 5 个条件中任何一个成立的充分条件，但非必要条件.

(五)　大数定律和中心极限定理

内容	对应概念、定理、重要公式
切比雪夫（Chebyshev）不等式，切比雪夫大数定律	1 切比雪夫不等式：$P\{\mid X-E(X)\mid \geq \varepsilon\} \leq \frac{D(X)}{\varepsilon^2}$ 或 $P\{\mid X-E(X)\mid < \varepsilon\} \geq 1-\frac{D(X)}{\varepsilon^2}$ 2 切比雪夫大数定律：设 $X_1, X_2 \cdots, X_n, \cdots$ 相互独立，且 $E(X_i)=\mu, D(X_i)=\sigma^2 (i=1,2,\cdots)$, 则对于任意正数 ε，有 $\lim\limits_{n\to\infty} P\left\{\left\lvert \frac{1}{n}\sum\limits_{i=1}^{n} X_i - \mu \right\rvert < \varepsilon\right\}=1$
伯努利大数定律，辛钦（Khinchine）大数定律	1 伯努利大数定律 设 $X_1, X_2 \cdots, X_n, \cdots$ 相互独立，同 0-1 分布 $B(1,p)$，则对任意正数 ε，有 $\lim\limits_{n\to\infty} P\left\{\left\lvert \frac{1}{n}\sum\limits_{i=1}^{n} X_i - p \right\rvert < \varepsilon\right\}=1$ 2 辛钦大数定律 设 $X_1, X_2 \cdots, X_n, \cdots$ 相互独立同分布，$EX_i=\mu, i=1,2$, 则对于任 意正数 ε，有 $\lim\limits_{n\to\infty} P\left\{\left\lvert \frac{1}{n}\sum\limits_{i=1}^{n} X_i - \mu \right\rvert < \varepsilon\right\}=1$
隶莫弗—拉普拉斯（De Movire-Laplace）定理，列维—林德伯格（Levy-Undbe）定理	1 棣莫弗---拉普斯定理 设 $\eta_n \sim B(n,p)$,（即 $X_1, X_2 \cdots, X_n$, 相互独立且同服从 0-1 分布 $\eta_n=\sum\limits_{i=1}^{n} X_i$ ）则有 $\lim\limits_{n\to\infty} P\left\{\frac{\eta_n - np}{\sqrt{np(1-p)}} \leq x\right\}=\int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt$ 2 列维---林德伯格定理 设 $X_1, X_2 \cdots, X_n, \cdots$ 相互独立分布， $E(X_i)=\mu, D(X_i)=\sigma^2 (\sigma \neq 0) i=1,2,\cdots,$ 则 $\lim\limits_{n\to\infty} P\left\{\frac{\sum\limits_{i=1}^{n} X_i - n\mu}{\sqrt{n}\sigma} \leq x\right\}=\int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt$

(六)　数理统计的基本概念

内容	对应公式、概念、定理
总体，个体，简单随机样本，统计量，样本均值，样本方差和样本矩	总体：研究对象的全体，它是一个随机变量，用 X 表示 个体：组成总体的每个基本元素 简单随机样本：来自总体 X 的 n 个相互独立且与总体同分布的随机变量 $X_1, X_2 \cdots, X_n$, 称为容量为 n 的简单随机样本，简称样本 统计量：设 $X_1, X_2 \cdots, X_n$, 是来自总体 X 的一个样本，$g(X_1, X_2 \cdots, X_n)$ ）是样本的连续函数，且 $g(\bullet)$ 中不含任何未知参数，则称 $g(X_1, X_2 \cdots, X_n)$ 为统计量 样本均值： $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ 样本方差： $S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2$ 样本矩：样本 k 阶原点矩： $A_k = \frac{1}{n}\sum_{i=1}^{n} X_i^k, k = 1,2,\cdots$ 样本 k 阶中心矩： $B_k = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^k, k = 1,2,\cdots$
χ^2 分布，t 分布，F 分布，分位数	χ^2 分布： $\chi^2 = X_1^2 + X_2^2 + \cdots + X_n^2 \sim \chi^2(n)$，其中 $X_1, X_2 \cdots, X_n$, 相互独立，且同服从 $N(0,1)$ t 分布： $T = \frac{X}{\sqrt{Y/n}} \sim t(n)$ 其中 $X \sim N(0,1), Y \sim \chi^2(n)$, 且 X，Y 相互独立 F 分布： $F = \frac{X/n_1}{Y/n_2} \sim F(n_1, n_2)$，其中 $X \sim \chi^2(n_1), Y \sim \chi^2(n_2)$, 且 X，Y 相互独立 分位数：若 $P(X \le x_\alpha) = \alpha$, 则称 x_α 为 X 的 α 分位数
正态总体的常用样本分布	1 设 $X_1, X_2 \cdots, X_n$ 为来自正态总体 $N(\mu, \sigma^2)$ 的样本， $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i, S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2$, 则 (1) $\bar{X} \sim N(\mu, \frac{\sigma^2}{n})$或$\frac{\bar{X} - \mu}{\sigma/\sqrt{n}} \sim N(0,1)$ (2) $\frac{(n-1)S^2}{\sigma^2} = \frac{1}{\sigma^2}\sum_{i=1}^{n}(X_i - \bar{X})^2 \sim \chi^2(n-1)$ (3) $\frac{1}{\sigma^2}\sum_{i=1}^{n}(X_i - \mu)^2 \sim \chi^2(n)$ (4) $\frac{\bar{X} - \mu}{S/\sqrt{n}} \sim t(n-1)$ **重要公式与结论** （1）对于 $\chi^2 \sim \chi^2(n)$，有 $E(\chi^2(n)) = n, D(\chi^2(n)) = 2n;$

	（2）对于 $T \sim t(n)$，有 $E(T)=0, D(T)=\frac{n}{n-2}(n>2)$； （3）对于 $F \sim F(m,n)$，有 $\frac{1}{F} \sim F(n,m), F_{a/2}(m,n)=\frac{1}{F_{1-a/2}(n,m)}$; （4）对于任意总体 X，有 $E(\bar{X})=E(X), E(S^2)=D(X), D(\bar{X})=\frac{D(X)}{n}$

(七) 参数估计

内容	对应公式、概念、定理			
点估计的概念，估计量与估计值，矩估计法，最大似然估计法	1 $\hat{\theta}$ 为 θ 的矩估计，g（x）为连续函数，则 g（$\hat{\theta}$）为 g（θ）的矩估计. 2 $\hat{\theta}$ 为 θ 的极大似数估计，g（x）为单调函数，则 $g(\hat{\theta})$ 为 $g(\theta)$ 的极大似然估计 3 $E(\bar{X})=E(X), E(S^2)=D(X),$ 即 $\bar{X}$，S^2 分别为总体 $E(X)$, $D(X)$ 的无偏估计量. 4 由大数定律易知 $\bar{X}$，S^2 也分别是 $E(X)$, $D(X)$ 的一致估量. 5 若 $E(\hat{\theta})=\theta, D(\hat{\theta})\to 0(n\to\infty)$ 则 $\hat{\theta}$ 为 θ 的一致估计.			
估计量的评选标准区间估计的概念	1 估计量的选取标准：无偏性、有效性、相合性 2 $(\hat{\theta}_1,\hat{\theta}_2)$ 为 θ 的置信度是 $1-\alpha$ 的置信区间，g（x）为单调增加（或单调减少）函数，则 $(g(\hat{\theta}_1), g(\hat{\theta}_2)$ 或 $g(\hat{\theta}_2)$，$g(\hat{\theta}_1))$ 为 g（θ）的置信度是 $1-\alpha$ 的置信区间			
单个正态总体的均值和方差	正态总体均值与方差的置信区间			
	待估参数		抽样分布	双侧置信区间
	μ	σ^2 已知	$U=\frac{\bar{X}-\mu}{\frac{\sigma}{\sqrt{n}}}\sim N(0,1)$	$(\bar{X}-\mu_{\frac{\alpha}{2}}, \bar{X}+\mu_{\frac{\alpha}{2}})$ $P\{\lvert\mu\rvert\geq\mu_{\frac{\alpha}{2}}\}=\alpha$

的区间估计，两个正态总体的均值差和方差比的区间估计		σ^2 未知	$T=\dfrac{\overline{X}-\mu}{S/\sqrt{n}}\sim t(n-1)$	$(\overline{X}-t_{\frac{\alpha}{2}}\cdot\dfrac{S}{\sqrt{n}},\overline{X}+t_{\frac{\alpha}{2}}\cdot\dfrac{S}{\sqrt{n}})$ $P\{\lvert T\rvert\geq t_{\frac{\alpha}{2}}\}=\alpha$
	σ^2	μ 已知	$W^{'}=\dfrac{1}{\sigma^2}\sum\limits_{i=1}^{n}(X_i-\mu)^2$ $\sim\chi^2(n)$	$(\dfrac{\sum\limits_{i=1}^{n}(X_i-\mu)^2}{\chi^2_{\frac{\alpha}{2}}(n)},\dfrac{\sum\limits_{i=1}^{n}(X_i-\mu)^2}{\chi^2_{1-\frac{\alpha}{2}}(n)})$ $P\{W^{'}\geq\chi^2_{\frac{\alpha}{2}}(n)\}=$ $P\{W^{'}\leq\chi^2_{1-\frac{\alpha}{2}}(n)\}=\dfrac{\alpha}{2}$
		μ 未知	$W=\dfrac{(n-1)S^2}{\sigma^2}\sim\chi^2(n-1)$	$\left(\dfrac{(n-1)S^2}{\chi^2_{\frac{\alpha}{2}}(n-1)},\dfrac{(n-1)S^2}{\chi^2_{1-\frac{\alpha}{2}}(n-1)}\right)$
	$\mu_1-\mu_2$	σ_1^2,σ_2^2 已知	$U=\dfrac{(\overline{X_1}-\overline{X_2})-(\mu_1-\mu_2)}{\sqrt{\dfrac{\sigma_1^2}{n_1}+\dfrac{\sigma_2^2}{n_2}}}$ $\sim N(0,1)$	$\left((\overline{X_1}-\overline{X_2})-\mu_{\frac{\alpha}{2}}\sqrt{\dfrac{\sigma_1^2}{n_1}+\dfrac{\sigma_2^2}{n_2}},\right.$ $\left.(\overline{X_1}-\overline{X_2})+\mu_{\frac{\alpha}{2}}\sqrt{\dfrac{\sigma_1^2}{n_1}+\dfrac{\sigma_2^2}{n_2}}\right)$ $P\{\lvert U\rvert\geq\mu_{\frac{\alpha}{2}}\}=\alpha$
		已知 $\sigma_1^2=\sigma_2^2=\sigma^2$，但$\sigma^2$未知	$T=\dfrac{(\overline{X_1}-\overline{X_2})-(\mu_1-\mu_2)}{S\sqrt{\dfrac{1}{n_1}+\dfrac{1}{n_2}}}\sim t(n_1+n_2-2)$ $S^2=\dfrac{(n_1-1)S_1^2+(n_2-1)S_2^2}{n_1+n_2-2}$	$\left((\overline{X_1}-\overline{X_2})-t_{\frac{\alpha}{2}}(n_1+n_2-2)\cdot S\sqrt{\dfrac{1}{n_1}+\dfrac{1}{n_2}},\right.$ $\left.(\overline{X_1}-\overline{X_2})+t_{\frac{\alpha}{2}}(n_1+n_2-2)\cdot S\sqrt{\dfrac{1}{n_1}+\dfrac{1}{n_2}}\right)$ $P\{\lvert T\rvert\geq t_{\frac{\alpha}{2}}\}=\alpha$
		$\dfrac{\sigma_1^2}{\sigma_2^2}$	$F=\dfrac{\dfrac{S_1^2}{\sigma_2^2}}{\dfrac{S_1^2}{\sigma_2^2}}\sim F(n_1-1,n_2-1)$	$\left(\dfrac{1}{F_{\frac{\alpha}{2}}(n_1-1,n_2-1)}\cdot\dfrac{S_1^2}{S_2^2},\right.$ $\left.F_{\frac{\alpha}{2}}(n_2-1,n_1-1)\cdot\dfrac{S_1^2}{S_2^2}\right)$ $P\{F\geq F_{\frac{\alpha}{2}}(n_1-1,n_2-1)\}=\dfrac{\alpha}{2}$ $P\{\dfrac{1}{F}\geq F_{\frac{\alpha}{2}}(n_2-1,n_1-1)\}=\dfrac{\alpha}{2}$

(八) 假设检验

<table>
<tr><th>内容</th><th colspan="4">对应公式、概念、定理</th></tr>
<tr><td>显著性检验，假设检验的两类错误</td><td colspan="4">1 假设检验的一般步骤
(1)确定所要检验的基本假设 H_0；
(2)选择检验的统计量，并要求知道其在一定条件下的分布；
(3)对确定的显著性水平 α，查相应的概率分布，得临界值，从而确定否定域；
(4)由样本计算统计量，并判断其是否落入否定域，从而对假设 H_0 作出拒绝还是接受的判断
2 假设检验的两类错误
统计推断是由样本推断总体，所作的结论不能保证绝对不犯错误，而只能以较大概率来保证其可靠性.
第一类错误是否定了真实的假设，即假设本来成立，但被错误地否认了，成为“弃真”，检验水平 α 就是犯第一类错误的概率的最大允许值.
第二类错误是把本来不成立的假设错误地接受了，称为“存伪”.犯这类错误的大小一般用 β 表示，它的大小要视具体
情况而定.</td></tr>
<tr><td rowspan="6">单个及两个正态总体的均值和方差的假设检验</td><td></td><td>原假设 H_0</td><td>H_0 下的检验统计量及分布</td><td>H_0 的拒绝域</td></tr>
<tr><td rowspan="4">一个正态总体</td><td>$\mu=\mu_0$
(σ^2已知）</td><td>$U=\dfrac{\bar{X}-\mu_0}{\sigma/\sqrt{n}}\sim N(0,1)$</td><td>$\lvert u\rvert=\left\lvert\dfrac{\bar{x}-\mu_0}{\sigma/\sqrt{n}}\right\rvert\geq u_{\frac{a}{2}}$</td></tr>
<tr><td>$\mu=\mu_0$
(σ^2未知）</td><td>$T=\dfrac{\bar{X}-\mu_0}{S/\sqrt{n}}\sim t(n-1)$</td><td>$\lvert t\rvert=\left\lvert\dfrac{\bar{x}-\mu_0}{S/\sqrt{n}}\right\rvert\geq t_{\frac{\alpha}{2}}(n-1)$</td></tr>
<tr><td>$\sigma^2=\sigma_0^2$
(μ 已知)</td><td>$W=\sum_{i=1}^{n}\left(\dfrac{X_i-\mu}{\sigma_0}\right)^2\sim\chi^2(n)$</td><td>$w=\sum_{i=1}^{n}\left(\dfrac{x_i-\mu}{\sigma_0}\right)^2\geq\chi^2_{\frac{a}{2}}(n)$ 或 $w\leq\chi^2_{1-\frac{a}{2}}(n)$</td></tr>
<tr><td>$\sigma^2=\sigma_0^2$
(μ 未知)</td><td>$W=\dfrac{(n-1)S^2}{\sigma_0^2}\sim\chi^2(n-1)$</td><td>$w=\dfrac{(n-1)S^2}{\sigma_0^2}\geq\chi^2_{\frac{a}{2}}(n-1)$ 或 $w\leq\chi^2_{1-\frac{a}{2}}(n-1)$</td></tr>
<tr><td>两个正</td><td>$\mu_1-\mu_2=\delta$
(σ_1^2,σ_2^2
已知）</td><td>$U=\dfrac{\bar{X}_1-\bar{X}_2-\delta}{\sqrt{\dfrac{\sigma_1^2}{n_1}+\dfrac{\sigma_2^2}{n_2}}}\sim N(0,1)$</td><td>$\lvert u\rvert=\left\lvert\dfrac{\bar{X}_1-\bar{X}_2-\delta}{\sqrt{\dfrac{\sigma_1^2}{n_1}+\dfrac{\sigma_2^2}{n_2}}}\right\rvert\geq u_{\frac{\alpha}{2}}$</td></tr>
</table>

	态总体	$\mu_1-\mu_2=\delta$ $(\sigma_1^2,\sigma_2^2$未知，但$\sigma_1^2=\sigma_2^2)$	$T=\dfrac{\bar{X}_1-\bar{X}_2-\delta}{S_W\sqrt{\dfrac{1}{n_1}+\dfrac{1}{n_2}}}$ $\sim t(n_1+n_2-2)$ $S_W^2=\dfrac{(n_1-1)S_1^2+(n_2-1)S_2^2}{n_1+n_2-2}$	$\lvert t\rvert=\left\lvert\dfrac{\bar{X}_1-\bar{X}_2-\delta}{S_W\sqrt{\dfrac{1}{n_1}+\dfrac{1}{n_2}}}\right\rvert$ $\geq t_{\frac{a}{2}}(n_1+n_2-2)$
		$\sigma_1^2=\sigma_2^2$ （μ_1，μ_2未知）	$F=\dfrac{S_1^2}{S_2^2}$ $\sim F(n_1-1,n_2-1)$	$f=\dfrac{S_1^2}{S_2^2}\geq F_{\frac{a}{2}}(n_1-1,n_2-1)$或 $f\leq F_{\frac{a}{2}}^{-1}(n_2-1,n_1-1)$