Chapter 8: Mixed Precision Training

DGL is compatible with the PyTorch Automatic Mixed Precision (AMP) package for mixed precision training, thus saving both training time and GPU/CPU memory consumption. This feature requires DGL 0.9+ and 1.1+ for CPU bloat16.

Message-Passing with Half Precision

DGL allows message-passing on `float16 (fp16)` / `bfloat16 (bf16)` features for both UDFs (User Defined Functions) and built-in functions (e.g., `dgl.function.sum`, `dgl.function.copy_u`).

> **Note**
>
> Please check bfloat16 support via `torch.cuda.is_bf16_supported()` before using it. Typically it requires CUDA >= 11.0 and GPU compute capability >= 8.0.

The following example shows how to use DGL's message-passing APIs on half-precision features:

```
>>> import torch
>>> import dgl
>>> import dgl.function as fn
>>> dev = torch.device('cuda')
>>> g = dgl.rand_graph(30, 100).to(dev)  # Create a graph on GPU w/ 30 nodes and 100 edges.
>>> g.ndata['h'] = torch.rand(30, 16).to(dev).half()  # Create fp16 node features.
>>> g.edata['w'] = torch.rand(100, 1).to(dev).half()  # Create fp16 edge features.
>>> # Use DGL's built-in functions for message passing on fp16 features.
>>> g.update_all(fn.u_mul_e('h', 'w', 'm'), fn.sum('m', 'x'))
>>> g.ndata['x'].dtype
torch.float16
>>> g.apply_edges(fn.u_dot_v('h', 'x', 'hx'))
>>> g.edata['hx'].dtype
torch.float16
```

```
>>> # Use UDFs for message passing on fp16 features.
>>> def message(edges):
...     return {'m': edges.src['h'] * edges.data['w']}
...
>>> def reduce(nodes):
...     return {'y': torch.sum(nodes.mailbox['m'], 1)}
...
>>> def dot(edges):
...     return {'hy': (edges.src['h'] * edges.dst['y']).sum(-1, keepdims=True)}
...
>>> g.update_all(message, reduce)
>>> g.ndata['y'].dtype
torch.float16
>>> g.apply_edges(dot)
>>> g.edata['hy'].dtype
torch.float16
```

End-to-End Mixed Precision Training

DGL relies on PyTorch's AMP package for mixed precision training, and the user experience is exactly the same as PyTorch's.

By wrapping the forward pass with `torch.amp.autocast()`, PyTorch automatically selects the appropriate datatype for each op and tensor. Half precision tensors are memory efficient, most operators on half precision tensors are faster as they leverage GPU tensorcores and CPU special instructon set.

```
import torch.nn.functional as F
from torch.amp import autocast

def forward(device_type, g, feat, label, mask, model, amp_dtype):
    amp_enabled = amp_dtype in (torch.float16, torch.bfloat16)
    with autocast(device_type, enabled=amp_enabled, dtype=amp_dtype):
        logit = model(g, feat)
        loss = F.cross_entropy(logit[mask], label[mask])
        return loss
```

Small Gradients in `float16` format have underflow problems (flush to zero). PyTorch provides a `GradScaler` module to address this issue. It multiplies the loss by a factor and invokes backward pass on the scaled loss to prevent the underflow problem. It then unscales the computed gradients before the optimizer updates the parameters. The scale factor is determined automatically. Note that `bfloat16` doesn't require a `GradScaler`.

```
from torch.cuda.amp import GradScaler

scaler = GradScaler()

def backward(scaler, loss, optimizer):
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
```

The following example trains a 3-layer GAT on the Reddit dataset (w/ 114 million edges). Pay attention to the differences in the code when AMP is activated or not.

```
import torch
import torch.nn as nn
import dgl
from dgl.data import RedditDataset
from dgl.nn import GATConv
from dgl.transforms import AddSelfLoop

amp_dtype = torch.bfloat16 # or torch.float16

class GAT(nn.Module):
    def __init__(self,
                 in_feats,
                 n_hidden,
                 n_classes,
                 heads):
        super().__init__()
        self.layers = nn.ModuleList()
        self.layers.append(GATConv(in_feats, n_hidden, heads[0], activation=F.elu))
        self.layers.append(GATConv(n_hidden * heads[0], n_hidden, heads[1], activation=F.elu))
        self.layers.append(GATConv(n_hidden * heads[1], n_classes, heads[2], activation=F.elu))

    def forward(self, g, h):
        for l, layer in enumerate(self.layers):
            h = layer(g, h)
            if l != len(self.layers) - 1:
                h = h.flatten(1)
            else:
                h = h.mean(1)
        return h

# Data loading
transform = AddSelfLoop()
data = RedditDataset(transform)
device_type = 'cuda' # or 'cpu'
dev = torch.device(device_type)

g = data[0]
g = g.int().to(dev)
train_mask = g.ndata['train_mask']
feat = g.ndata['feat']
label = g.ndata['label']

in_feats = feat.shape[1]
n_hidden = 256
n_classes = data.num_classes
heads = [1, 1, 1]
model = GAT(in_feats, n_hidden, n_classes, heads)
model = model.to(dev)
model.train()

# Create optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=5e-4)

for epoch in range(100):
    optimizer.zero_grad()
    loss = forward(device_type, g, feat, label, train_mask, model, amp_dtype)

    if amp_dtype == torch.float16:
        # Backprop w/ gradient scaling
        backward(scaler, loss, optimizer)
    else:
        loss.backward()
        optimizer.step()
```

```
    print('Epoch {} | Loss {}'.format(epoch, loss.item()))
```

On a NVIDIA V100 (16GB) machine, training this model without fp16 consumes 15.2GB GPU memory; with fp16 turned on, the training consumes 12.8G GPU memory, the loss converges to similar values in both settings. If we change the number of heads to `[2, 2, 2]`, training without fp16 triggers GPU OOM(out-of-memory) issue while training with fp16 consumes 15.7G GPU memory.

BFloat16 CPU example

DGL supports running training in the bfloat16 data type on the CPU. This data type doesn't require any CPU feature and can improve the performance of a memory-bound model. Starting with Intel Xeon 4th Generation, which has AMX instructon set, bfloat16 should significantly improve training and inference performance without huge code changes. Here is an example of simple GCN bfloat16 training:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import dgl
from dgl.data import CiteseerGraphDataset
from dgl.nn import GraphConv
from dgl.transforms import AddSelfLoop


class GCN(nn.Module):
    def __init__(self, in_size, hid_size, out_size):
        super().__init__()
        self.layers = nn.ModuleList()
        # two-layer GCN
        self.layers.append(
            GraphConv(in_size, hid_size, activation=F.relu)
        )
        self.layers.append(GraphConv(hid_size, out_size))
        self.dropout = nn.Dropout(0.5)

    def forward(self, g, features):
        h = features
        for i, layer in enumerate(self.layers):
            if i != 0:
                h = self.dropout(h)
            h = layer(g, h)
        return h


# Data loading
transform = AddSelfLoop()
data = CiteseerGraphDataset(transform=transform)

g = data[0]
g = g.int()
train_mask = g.ndata['train_mask']
feat = g.ndata['feat']
label = g.ndata['label']

in_size = feat.shape[1]
hid_size = 16
out_size = data.num_classes
model = GCN(in_size, hid_size, out_size)

# Convert model and graph to bfloat16
g = dgl.to_bfloat16(g)
feat = feat.to(dtype=torch.bfloat16)
model = model.to(dtype=torch.bfloat16)

model.train()

# Create optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=1e-2, weight_decay=5e-4)
loss_fcn = nn.CrossEntropyLoss()

for epoch in range(100):
    logits = model(g, feat)
    loss = loss_fcn(logits[train_mask], label[train_mask])

    loss.backward()
    optimizer.step()

    print('Epoch {} | Loss {}'.format(epoch, loss.item()))
```

The only difference with common training is model and graph conversion before training/inference.

DGL is still improving its half-precision support and the compute kernel's performance is far from optimal, please stay tuned to our future updates.