First Steps

The easiest way to train and evaluate a model is with the `pykeen.pipeline.pipeline()` function.

It provides a high-level entry point into the extensible functionality of this package. Full reference documentation for the pipeline and related functions can be found at `pykeen.pipeline`.

Training a Model

The following example shows how to train and evaluate the `pykeen.models.TransE` model on the `pykeen.datasets.Nations` dataset. Throughout the documentation, you'll notice that each asset has a corresponding class in PyKEEN. You can follow the links to learn more about each and see the reference on how to use them specifically. Don't worry, in this part of the tutorial, the `pykeen.pipeline.pipeline()` function will take care of everything for you.

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

The results are returned in a `pykeen.pipeline.PipelineResult` instance, which has attributes for the trained model, the training loop, and the evaluation.

In this example, the model was given as a string. A list of available models can be found in `pykeen.models`. Alternatively, the class corresponding to the implementation of the model could be used as in:

```
>>> from pykeen.pipeline import pipeline
>>> from pykeen.models import TransE
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model=TransE,
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

In this example, the dataset was given as a string. A list of available datasets can be found in `pykeen.datasets`. Alternatively, a subclass of `pykeen.datasets.Dataset` could be used as in:

```
>>> from pykeen.pipeline import pipeline
>>> from pykeen.models import TransE
>>> from pykeen.datasets import Nations
>>> pipeline_result = pipeline(
...     dataset=Nations,
...     model=TransE,
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

In each of the previous three examples, the training approach, optimizer, and evaluation scheme were omitted. By default, the model is trained under the stochastic local closed world assumption (sLCWA; `pykeen.training.SLCWATrainingLoop`). This can be explicitly given as a string:

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     training_loop='sLCWA',
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

Alternatively, the model can be trained under the local closed world assumption (LCWA; `pykeen.training.LCWATrainingLoop`) by giving `'LCWA'`. No additional configuration is necessary, but it's worth reading up on the differences between these training approaches. A list of available training assumptions can be found in `pykeen.training`.

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     training_loop='LCWA',
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

One of these differences is that the sLCWA relies on *negative sampling*. The type of negative sampling can be given as in:

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     training_loop='sLCWA',
...     negative_sampler='basic',
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

In this example, the negative sampler was given as a string. A list of available negative samplers can be found in `pykeen.sampling`. Alternatively, the class corresponding to the implementation of the negative sampler could be used as in:

```
>>> from pykeen.pipeline import pipeline
>>> from pykeen.sampling import BasicNegativeSampler
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     training_loop='sLCWA',
...     negative_sampler=BasicNegativeSampler,
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

Warning

The `negative_sampler` keyword argument should not be used if the LCWA is being used. In general, all other options are available under either training approach.

The type of evaluation perfomed can be specified with the `evaluator` keyword. By default, rank-based evaluation is used. It can be given explictly as in:

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     evaluator='RankBasedEvaluator',
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

In this example, the evaluator string. A list of available evaluators can be found in `pykeen.evaluation`. Alternatively, the class corresponding to the implementation of the evaluator could be used as in:

```
>>> from pykeen.pipeline import pipeline
>>> from pykeen.evaluation import RankBasedEvaluator
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     evaluator=RankBasedEvaluator,
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

PyKEEN implements early stopping, which can be turned on with the `stopper` keyword argument as in:

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     stopper='early',
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

In PyKEEN you can also use the learning rate schedulers provided by PyTorch, which can be turned on with the `lr_scheduler` keyword argument together with the `lr_scheduler_kwargs` keyword argument to specify arguments for the learning rate scheduler as in:

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     lr_scheduler='ExponentialLR',
...     lr_scheduler_kwargs=dict(
...         gamma=0.99,
...     ),
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

Deeper Configuration

Arguments for the model can be given as a dictionary using `model_kwargs`.

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     model_kwargs=dict(
...         scoring_fct_norm=2,
...     ),
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

The entries in `model_kwargs` correspond to the arguments given to `pykeen.models.TransE.__init__()`. For a complete listing of models, see `pykeen.models`, where there are links to the reference for each model that explain what kwargs are possible. Each model's default hyper-parameters were chosen based on the best reported values from the paper originally publishing the model unless otherwise noted on the model's reference page.

Because the pipeline takes care of looking up classes and instantiating them, there are several other parameters to `pykeen.pipeline.pipeline()` that can be used to specify the parameters during their respective instantiations.

Arguments can be given to the dataset with `dataset_kwargs`. These are passed on to the `pykeen.datasets.Nations`

Loading a pre-trained Model

Many of the previous examples ended with saving the results using the `pykeen.pipeline.PipelineResult.save_to_directory()`. One of the artifacts written to the given directory is the `trained_model.pkl` file. Because all PyKEEN models inherit from `torch.nn.Module`, we use the PyTorch mechanisms for saving and loading them. This means that you can use `torch.load()` to load a model like:

```python
import torch

my_pykeen_model = torch.load('trained_model.pkl')
```

More information on PyTorch's model persistence can be found at: https://pytorch.org/tutorials/beginner/saving_loading_models.html.

Mapping Entity and Relation Identifiers to their Names

While PyKEEN internally maps entities and relations to contiguous identifiers, it's still useful to be able to interact with datasets, triples factories, and models using the labels of the entities and relations.

We can map a triples factory's entities to identifiers using `TriplesFactory.entities_to_ids()` like in the following example:

```python
from pykeen.datasets import Nations

triples_factory = Nations().training

# Get tensor of entity identifiers
entity_ids = torch.as_tensor(triples_factory.entities_to_ids(["china", "egypt"]))
```

Similarly, we can map a triples factory's relations to identifiers using `TriplesFactory.relations_to_ids` like in the following example:

```python
relation_ids = torch.as_tensor(triples_factory.relations_to_ids(["independence", "embassy"]))
```

Warning

It's important to notice that we should use a triples factory with the same mapping that was used to train the model - otherwise we might end up with incorrect IDs.

Using Learned Embeddings

The embeddings learned for entities and relations are not only useful for link prediction (see Prediction), but also for other downstream machine learning tasks like clustering, regression, and classification.

Knowledge graph embedding models can potentially have multiple entity representations and multiple relation representations, so they are respectively stored as sequences in the `entity_representations` and `relation_representations` attributes of each model. While the exact contents of these sequences are model-dependent, the first element of each is usually the "primary" representation for either the entities or relations.

Typically, the values in these sequences are instances of the `pykeen.nn.representation.Embedding`. This implements a similar, but more powerful, interface to the built-in `torch.nn.Embedding` class. However, the values in these sequences can more generally be instances of any subclasses of `pykeen.nn.representation.Representation`. This allows for more powerful encoders those in GNNs such as `pykeen.models.RGCN` to be implemented and used.

The entity representations and relation representations can be accessed like this:

```
from typing import List

import pykeen.nn
from pykeen.pipeline import pipeline

result = pipeline(model='TransE', dataset='UMLS')
model = result.model

entity_representation_modules: List['pykeen.nn.Representation'] = model.entity_representations
relation_representation_modules: List['pykeen.nn.Representation'] =
model.relation_representations
```

Most models, like `pykeen.models.TransE`, only have one representation for entities and one for relations. This means that the `entity_representations` and `relation_representations` lists both have a length of 1. All of the entity embeddings can be accessed like:

```
entity_embeddings: pykeen.nn.Embedding = entity_representation_modules[0]
relation_embeddings: pykeen.nn.Embedding = relation_representation_modules[0]
```

Since all representations are subclasses of `torch.nn.Module`, you need to call them like functions to invoke the *forward()* and get the values.

```
entity_embedding_tensor: torch.FloatTensor = entity_embeddings()
relation_embedding_tensor: torch.FloatTensor = relation_embeddings()
```

The *forward()* function of all `pykeen.nn.representation.Representation` takes an `indices` parameter. By default, it is `None` and returns all values. More explicitly, this looks like:

```
entity_embedding_tensor: torch.FloatTensor = entity_embeddings(indices=None)
relation_embedding_tensor: torch.FloatTensor = relation_embeddings(indices=None)
```

If you'd like to only look up certain embeddings, you can use the `indices` parameter and pass a `torch.LongTensor` with their corresponding indices.

You might want to detach them from the GPU and convert to a `numpy.ndarray` with

```
entity_embedding_tensor = model.entity_representations[0](indices=None).detach().numpy()
```

Warning

Some old-style models (e.g., ones inheriting from `pykeen.models.EntityRelationEmbeddingModel`) don't fully implement the `entity_representations` and `relation_representations` interface. This means that they might have additional embeddings stored in attributes that aren't exposed through these sequences. For example, `pykeen.models.TransD` has a secondary entity embedding in `pykeen.models.TransD.entity_projections`. Eventually, all models will be upgraded to new-style models and this won't be a problem.

Beyond the Pipeline

While the pipeline provides a high-level interface, each aspect of the training process is encapsulated in classes that can be more finely tuned or subclassed. Below is an example of code that might have been executed with one of the previous examples.

```
>>> # Get a training dataset
>>> from pykeen.datasets import Nations
>>> dataset = Nations()
>>> training_triples_factory = dataset.training

>>> # Pick a model
>>> from pykeen.models import TransE
>>> model = TransE(triples_factory=training_triples_factory)

>>> # Pick an optimizer from Torch
>>> from torch.optim import Adam
>>> optimizer = Adam(params=model.get_grad_params())

>>> # Pick a training approach (sLCWA or LCWA)
>>> from pykeen.training import SLCWATrainingLoop
>>> training_loop = SLCWATrainingLoop(
...     model=model,
...     triples_factory=training_triples_factory,
...     optimizer=optimizer,
... )

>>> # Train like Cristiano Ronaldo
>>> _ = training_loop.train(
...     triples_factory=training_triples_factory,
...     num_epochs=5,
...     batch_size=256,
... )

>>> # Pick an evaluator
>>> from pykeen.evaluation import RankBasedEvaluator
>>> evaluator = RankBasedEvaluator()

>>> # Get triples to test
>>> mapped_triples = dataset.testing.mapped_triples

>>> # Evaluate
>>> results = evaluator.evaluate(
...     model=model,
...     mapped_triples=mapped_triples,
...     batch_size=1024,
...     additional_filter_triples=[
...         dataset.training.mapped_triples,
...         dataset.validation.mapped_triples,
...     ],
... )
>>> # print(results)
```

Preview: Evaluation Loops

PyKEEN is currently in the transition to use torch's data-loaders for evaluation, too. While not being active for the high-level *pipeline*, you can already use it explicitly:

```
>>> # get a dataset
>>> from pykeen.datasets import Nations
>>> dataset = Nations()

>>> # Pick a model
>>> from pykeen.models import TransE
>>> model = TransE(triples_factory=dataset.training)

>>> # Pick a training approach (sLCWA or LCWA)
>>> from pykeen.training import SLCWATrainingLoop
>>> training_loop = SLCWATrainingLoop(
...     model=model,
...     triples_factory=dataset.training,
... )

>>> # Train like Cristiano Ronaldo
>>> _ = training_loop.train(
...     triples_factory=training_triples_factory,
...     num_epochs=5,
...     batch_size=256,
...     # NEW: validation evaluation callback
...     callbacks="evaluation-loop",
...     callback_kwargs=dict(
...         prefix="validation",
...         factory=dataset.validation,
...     ),
... )

>>> # Pick an evaluation loop (NEW)
>>> from pykeen.evaluation import LCWAEvaluationLoop
>>> evaluation_loop = LCWAEvaluationLoop(
...     model=model,
...     triples_factory=dataset.testing,
... )

>>> # Evaluate
>>> results = evaluation_loop.evaluate()
>>> # print(results)
```

Training Callbacks

PyKEEN allows interaction with the training loop through callbacks. One particular use case is regular evaluation (outside of an early stopper). The following example shows how to evaluate on the training triples on every tenth epoch

```
from pykeen.datasets import get_dataset
from pykeen.pipeline import pipeline

dataset = get_dataset(dataset="nations")
result = pipeline(
    dataset=dataset,
    model="mure",
    training_kwargs=dict(
        num_epochs=100,
        callbacks="evaluation",
        callback_kwargs=dict(
            evaluation_triples=dataset.training.mapped_triples,
            tracker="console",
            prefix="training",
        ),
    ),
)
```

For further information about different result trackers, take a look at the section on Result Trackers.

Next Steps

The first steps tutorial taught you how to train and use a model for some of the most common tasks. There are several other topic-specific tutorials in the section of the documentation. You might also want to jump ahead to the Troubleshooting section in case you're having trouble, or look through questions and discussions that others have posted on GitHub.