Extending the Models

You should first read the tutorial on bringing your own interaction module. This tutorial is about how to wrap a custom interaction module with a model module for general reuse and application.

Implementing a model by subclassing `pykeen.models.ERModel`

The following code block demonstrates how an interaction model can be used to define a full KGEM using the `pykeen.models.ERModel` base class.

```python
from pykeen.models import ERModel
from pykeen.nn import Embedding, Interaction


class DistMultInteraction(Interaction):
    def forward(self, h, r, t):
        return (h * r * t).sum(dim=-1)


class DistMult(ERModel):
    def __init__(
        self,
        # When defining your class, any hyper-parameters that can be configured should be
        # made as arguments to the __init__() function. When running the pipeline(), these
        # are passed via the ``model_kwargs``.
        embedding_dim: int = 50,
        # All remaining arguments are simply passed through to the parent constructor. If you
        # want access to them, you can name them explicitly. See the pykeen.models.ERModel
        # documentation for a full list
        **kwargs,
    ) -> None:
        # since this is a python class, you can feel free to get creative here. One example of
        # pre-processing is to derive the shape for the relation representation based on the
        # embedding dimension.
        super().__init__(
            # Pass an instance of your interaction function. This is also a place where you can
            # pass hyper-parameters, such as the L_p norm, from the KGEM to the interaction
function
            interaction=DistMultInteraction,
            # interaction_kwargs=dict(...),
            # Define the entity representations using a dict. By default, each
            # embedding is a vector. You can use the ``shape`` kwarg to specify higher
dimensional
            # tensor shapes.
            entity_representations=Embedding,
            entity_representations_kwargs=dict(
                embedding_dim=embedding_dim,
            ),
            # Define the relation representations the same as the entities
            relation_representations=Embedding,
            relation_representations_kwargs=dict(
                embedding_dim=embedding_dim,
            ),
            # All other arguments are passed through, such as the ``triples_factory``,
``loss``,
            # ``preferred_device``, and others. These are all handled by the pipeline()
function
            **kwargs,
        )
```

The actual implementation of DistMult can be found in `pykeen.models.DistMult`. Note that it additionally contains configuration for the initializers, constrainers, and regularizers for each of the embeddings as well as class-level defaults for hyper-parameters and hyper-parameter optimization. Modifying these is covered in other tutorials.

Specifying Defaults

If you have a preferred loss function for your model, you can add the `loss_default` class variable where the value is the loss class.

```
from typing import ClassVar

from pykeen.models import ERModel
from pykeen.losses import Loss, NSSALoss

class DistMult(ERModel):
    loss_default: ClassVar[Type[Loss]] = NSSALoss
    ...
```

Now, when using the pipeline, the `pykeen.losses.NSSALoss`. loss is used by default if none is given. The same kind of modifications can be made to set a default regularizer with `regularizer_default`.

Specifying Hyper-parameter Optimization Default Ranges

All subclasses of `pykeen.models.Model` can specify the default ranges or values used during hyper-parameter optimization (HPO). PyKEEN implements a simple dictionary-based configuration that is interpreted by `pykeen.hpo.hpo.suggest_kwargs()` in the HPO pipeline.

HPO default ranges can be applied to all keyword arguments appearing in the `__init__()` function of your model by setting a class-level variable called `hpo_default`.

For example, the `embedding_dim` can be specified as being on a range between 100 and 150 with the following:

```
class DistMult(ERModel):
    hpo_default = {
        'embedding_dim': dict(type=int, low=100, high=150)
    }
    ...
```

A step size can be imposed with `q`:

```
class DistMult(ERModel):
    hpo_default = {
        'embedding_dim': dict(type=int, low=100, high=150 q=5)
    }
    ...
```

An alternative scale can be imposed with `scale`. Right now, the default is linear, and `scale` can optionally be set to `power_two` for integers as in:

```python
class DistMult(ERModel):
    hpo_default = {
        # will uniformly give 16, 32, 64, 128 (left inclusive, right exclusive)
        'hidden_dim': dict(type=int, low=4, high=8, scale='power_two')
    }
    ...
```

Warning

Alternative scales can not currently be used in combination with step size (`q`).

There are other possibilities for specifying the `type` as `float`, `categorical`, or as `bool`.

With `float`, you can't use the `q` option nor set the scale to `power_two`, but the scale can be set to `log` (see `optuna.distributions.LogUniformDistribution`).

```python
hpo_default = {
    # will uniformly give floats on the range of [1.0, 2.0) (exclusive)
    'alpha': dict(type='float', low=1.0, high=2.0),

    # will uniformly give 1.0, 2.0, or 4.0 (exclusive)
    'beta': dict(type='float', low=1.0, high=8.0, scale='log'),
}
```

With `categorical`, you can form a dictionary like the following using `type='categorical'` and giving a `choices` entry that contains a sequence of either integers, floats, or strings.

```python
hpo_default = {
    'similarity': dict(type='categorical', choices=[...])
}
```

With `bool`, you can simply use `dict(type=bool)` or `dict(type='bool')`.

Note

The HPO rules are subject to change as they are tightly coupled to `optuna`, which since version 2.0.0 has introduced several new possibilities.

Implementing a model by instantiating

`pykeen.models.ERModel`

Instead of creating a new class, you can also directly use the `pykeen.models.ERModel`, e.g.

```
from pykeen.models import ERModel
from pykeen.losses import BCEWithLogitsLoss

model = ERModel(
    triples_factory=...,
    loss="BCEWithLogits",
    interaction="transformer",
    entity_representations_kwargs=dict(embedding_dim=64),
    relation_representations_kwargs=dict(embedding_dim=64),
)
```

Using a Custom Model with the Pipeline

We can use this new model with all available losses, evaluators, training pipelines, inverse triple modeling, via the `pykeen.pipeline.pipeline()`, since in addition to the names of models (given as strings), it can also take model classes in the `model` argument.

```
from pykeen.pipeline import pipeline

pipeline(
    model=DistMult,
    dataset='Nations',
    loss='NSSA',
)
```