Bring Your Own Interaction

This is a tutorial about how to implement your own interaction modules (also known as scoring functions) as subclasses of `pykeen.nn.modules.Interaction` for use in PyKEEN.

Implementing your first Interaction Module

Imagine you've taken a time machine back to 2013 and you have just invented TransE, defined as:

$$f(h, r, t) = -\|\mathbf{e}_h + \mathbf{r}_r - \mathbf{e}_t\|_2$$

where $\mathbf{e}_i$ is the d-dimensional representation for entity i, $\mathbf{r}_j$ is the d-dimensional representation for relation j, and $\|\ldots\|_2$ is the L_2 norm.

To implement TransE in PyKEEN, you need to subclass the `pykeen.nn.modules.Interaction`. This class it itself a subclass of `torch.nn.Module`, which means that you need to provide an implementation of `torch.nn.Module.forward()`. However, the arguments are predefined as `h`, `r`, and `t`, which correspond to the representations of the head, relation, and tail, respectively.

```python
from pykeen.nn.modules import Interaction

class TransEInteraction(Interaction):
    def forward(self, h, r, t):
        return -(h + r - t).norm(p=2, dim=-1)
```

Note the `dim=-1` because this operation is actually defined over an entire batch of head, relation, and tail representations.

See also

A reference implementation is provided in `pykeen.nn.modules.TransEInteraction`

As a researcher who just invented TransE, you might wonder what would happen if you replaced the addition `+` with multiplication `*`. You might then end up with a new interaction like this (which just happens to be DistMult, which was published just a year after TransE):

$$f(h, r, t) = \mathbf{e}_h^T diag(\mathbf{r}_r)\mathbf{e}_t$$

where $\mathbf{e}_i$ is the d-dimensional representation for entity i, $\mathbf{r}_j$ is the d-dimensional representation for relation j.

```python
from pykeen.nn.modules import Interaction

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

See also

A reference implementation is provided in `pykeen.nn.modules.DistMultInteraction`

Interactions with Hyper-Parameters

While we previously defined TransE with the L_2 norm, it could be calculated with a different value for p:

$$f(h, r, t) = -\|\mathbf{e}_h + \mathbf{r}_r - \mathbf{e}_t\|_p$$

This could be incorporated into the interaction definition by using the `__init__()`, storing the value for p in the instance, then accessing it in `forward()`.

```python
from pykeen.nn.modules import Interaction

class TransEInteraction(Interaction):
    def __init__(self, p: int):
        super().__init__()
        self.p = p

    def forward(self, h, r, t):
        return -(h + r - t).norm(p=self.p, dim=-1)
```

In general, you can put whatever you want in `__init__()` to support the calculation of scores.

Interactions with Trainable Parameters

In ER-MLP, the multi-layer perceptron consists of an input layer with $3 \times d$ neurons, a hidden layer with d neurons and output layer with one neuron. The input is represented by the concatenation embeddings of the heads, relations and tail embeddings. It is defined as:

$$f(h, r, t) = W_2 ReLU(W_1 cat(h, r, t) + b_1) + b_2$$

with hidden dimension y, $W_1 \in \mathcal{R}^{3d \times y}$, $W_2 \in \mathcal{R}^y$, and biases $b_1 \in \mathcal{R}^y$ and $b_2 \in \mathcal{R}$.

W_1, W_1, b_1, and b_2 are *global* parameters, meaning that they are trainable, but are neither attached to the entities nor relations. Unlike the p in TransE, these global trainable parameters are not considered hyper-parameters. However, like hyper-parameters, they can also be defined in the *__init__* function of your `pykeen.nn.modules.Interaction` class. They are trained jointly with the entity and relation embeddings during training.

```
import torch.nn
from pykeen.nn.modules import Interaction
from pykeen.utils import broadcast_cat

class ERMLPInteraction(Interaction):
    def __init__(self, embedding_dim: int, hidden_dim: int):
        super().__init__()
        # The weights of this MLP will be learned.
        self.mlp = torch.nn.Sequential(
            torch.nn.Linear(in_features=3 * embedding_dim, out_features=hidden_dim, bias=True),
            torch.nn.ReLU(),
            torch.nn.Linear(in_features=hidden_dim, out_features=1, bias=True),
        )

    def forward(self, h, r, t):
        x = broadcast_cat([h, r, t], dim=-1)
        return self.mlp(x)
```

Note that `pykeen.utils.broadcast_cat()` was used instead of the standard `torch.cat()` because of the standardization of shapes of head, relation, and tail vectors.

See also

A reference implementation is provided in `pykeen.nn.modules.ERMLPInteraction`

Interactions with Different Shaped Vectors

The Structured Embedding uses a 2-tensor for representing each relation, with an interaction defined as:

$$f(h, r, t) = -\|\mathbf{M}_r^{head}\mathbf{e}_h - \mathbf{M}_r^{tail}\mathbf{e}_t\|_p$$

where $\mathbf{e}_i$ is the d-dimensional representation for entity i, $\mathbf{M}_j^{head}$ is the $d \times d$-dimensional representation for relation j for head entities, $\mathbf{M}_j^{tail}$ is the $d \times d$-dimensional representation for relation j for tail entities, and $\|\ldots\|_2$ is the L_p norm.

For the purposes of this tutorial, we will propose a simplification to Strucuterd Embedding (also similar to TransR) where the same relation 2-tensor is used to project both the head and tail entities as in:

$$f(h, r, t) = -\|\mathbf{M}_r\mathbf{e}_h - \mathbf{M}_r\mathbf{e}_t\|_2$$

where $\mathbf{e}_i$ is the d-dimensional representation for entity i, $\mathbf{M}_j$ is the $d \times d$-dimensional representation for relation j, and $\|\ldots\|_2$ is the L_2 norm.

```python
from pykeen.nn.modules import Interaction

class SimplifiedStructuredEmbeddingInteraction(Interaction):
    relation_shape = ('dd',)

    def forward(self, h, r, t):
        h_proj = r @ h.unsqueeze(dim=-1)
        t_proj = r @ t.unsqueeze(dim=-1)
        return -(h_proj - t_proj).squeeze(dim=-1).norm(p=2, dim=-1)
```

Note the definition of the `relation_shape`. By default, the `entity_shape` and `relation_shape` are both equal to `('d', )`, which uses eigen-notation to show that they both are 1-tensors with the same shape. In this simplified version of Structured Embedding, we need to denote that the shape of the relation is $d \times d$, so it's written as `dd`.

See also

Reference implementations are provided in `pykeen.nn.modules.StructuredEmbeddingInteraction` and in `pykeen.nn.modules.TransRInteraction`.

Interactions with Multiple Representations

Sometimes, like in the canonical version of Structured Embedding, you need more than one representation for entities and/or relations. To specify this, you just need to extend the tuple for `relation_shape` with more entries, each corresponding to the sequence of representations.

```python
from pykeen.nn.modules import Interaction

class StructuredEmbeddingInteraction(Interaction):
    relation_shape = (
        'dd',  # Corresponds to $\mathbf{M}^{head}_j$
        'dd',  # Corresponds to $\mathbf{M}^{tail}_j$
    )

    def forward(self, h, r, t):
        # Since the relation_shape is more than length 1, the r value is given as a sequence
        # of the representations defined there. You can use tuple unpacking to get them out
        r_h, r_t = r
        h_proj = r_h @ h.unsqueeze(dim=-1)
        t_proj = r_t @ t.unsqueeze(dim=-1)
        return -(h_proj - t_proj).squeeze(dim=-1).norm(p=2, dim=-1)
```

Interactions with Different Dimension Vectors

TransD is an example of an interaction module that not only uses two different representations for each entity and two representations for each relation, but they are of different dimensions.

It can be implemented by choosing a different letter for use in the `entity_shape` and/or `relation_shape` dictionary. Ultimately, the letters used are arbitrary, but you need to remember what they are when using the `pykeen.models.make_model()`, `pykeen.models.make_model_cls()`, or `pykeen.pipeline.interaction_pipeline()` functions to instantiate a model, make a model class, or run the pipeline using your custom interaction module (respectively).

```python
from pykeen.nn.modules import Interaction
from pykeen.utils import project_entity

class TransDInteraction(Interaction):
    entity_shape = ("d", "d")
    relation_shape = ("e", "e")

    def forward(self, h, r, t):
        h, h_proj = h
        r, r_proj = r
        t, t_proj = t
        h_bot = project_entity(
            e=h,
            e_p=h_p,
            r_p=r_p,
        )
        t_bot = project_entity(
            e=t,
            e_p=t_p,
            r_p=r_p,
        )
        return -(h_bot + r - t_bot).norm(p=2, dim=-1)
```

Note

The `pykeen.utils.project_entity()` function was used in this implementation to reduce the complexity. So far, it's the case that all of the models using multiple different representation dimensions are quite complicated and don't fall into the paradigm of presenting simple examples.

See also

A reference implementation is provided in `pykeen.nn.modules.TransDInteraction`

Differences between `pykeen.nn.modules.Interaction` and `pykeen.models.Model`

The high-level `pipeline()` function allows you to pass pre-defined subclasses of `pykeen.models.Model` such as `pykeen.models.TransE` or `pykeen.models.DistMult`. These classes are high-level wrappers around the interaction functions `pykeen.nn.modules.TransEInteraction` and `nn.modules.DistMultInteraction` that are more suited for running benchmarking experiments or practical applications of knowledge graph embeddings that include lots of information about default hyper-parameters, recommended hyper-parameter optimization strategies, and more complex applications of regularization schemas.

As a researcher, the `pykeen.nn.modules.Interaction` is a way to quickly translate ideas into new models that can be used without all of the overhead of defining a `pykeen.models.Model`. These components are also completely reusable throughout PyKEEN (e.g., in self-rolled training loops) and can be used as standalone components outside of PyKEEN.

If you are happy with your interaction module and would like to go the next step to making it generally reusable, check the “Extending the Models” tutorial.

Ad hoc Models from Interactions

A `pykeen.models.ERModel` can be constructed from `pykeen.nn.modules.Interaction`.

The new style-class, `pykeen.models.ERModel` abstracts the interaction away from the representations such that different interactions can be used interchangably. A new model can be constructed directly from the interaction module, given a `dimensions` mapping. In each `pykeen.nn.modules.Interaction`, there is a field called `entity_shape` and `relation_shape` that allows for using eigen-notation for defining the different dimensions of the model. Most models share the `d` dimensionality for both the entity and relation vectors. Some (but not all) exceptions are:

- `pykeen.nn.modules.RESCALInteraction`, which uses a square matrix for relations written as `dd`
- `pykeen.nn.modules.TransDInteraction`, which uses `d` for entity shape and `e` for a different relation shape.

With this in mind, you’ll have to investigate the dimensions of the vectors through the PyKEEN documentation. If you’re implementing your own, you have control over this and will know which dimensions to specify (though the `d` for both entities and relations is standard). As a shorthand for `{'d': value}`, you can directly pass `value` for the dimension and it will be automatically interpreted as the `{'d': value}`.

Make a model class from lookup of an interaction module class:

```
>>> from pykeen.nn.modules import TransEInteraction
>>> from pykeen.models import make_model_cls
>>> embedding_dim = 3
>>> model_cls = make_model_cls(
...     dimensions={"d": embedding_dim},
...     interaction='TransE',
...     interaction_kwargs={'p': 2},
... )
```

If there's only one dimension in the `entity_shapes` and `relation_shapes`, it can be directly given as an integer as a shortcut.

```
>>> # Implicitly can also be written as:
>>> model_cls_alt = make_model_cls(
...     dimensions=embedding_dim,
...     interaciton='TransE',
...     interaction_kwargs={'p': 2},
... )
```

Make a model class from an interaction module class:

```
>>> from pykeen.nn.modules import TransEInteraction
>>> from pykeen.models import make_model_cls
>>> embedding_dim = 3
>>> model_cls = make_model_cls({"d": embedding_dim}, TransEInteraction, {'p': 2})
```

Make a model class from an instantiated interaction module:

```
>>> from pykeen.nn.modules import TransEInteraction
>>> from pykeen.models import make_model_cls
>>> embedding_dim = 3
>>> model_cls = make_model_cls({"d": embedding_dim}, TransEInteraction(p=2))
```

All of these model classes can be passed directly into the `model` argument of `pykeen.pipeline.pipeline()`.

Interaction Pipeline

The `pykeen.pipeline.pipeline()` also allows passing of an interaction such that the following code block can be compressed:

```
from pykeen.pipeline import pipeline
from pykeen.nn.modules import TransEInteraction

model = make_model_cls(
    interaction=TransEInteraction,
    interaction_kwargs={'p': 2},
    dimensions={'d': 100},
)
results = pipeline(
    dataset='Nations',
    model=model,
    ...
)
```

into:

```
from pykeen.pipeline import pipeline
from pykeen.nn.modules import TransEInteraction

results = pipeline(
    dataset='Nations',
    interaction=TransEInteraction,
    interaction_kwargs={'p': 2},
    dimensions={'d': 100},
    ...
)
```

This can be used with any subclass of the `pykeen.nn.modules.Interaction`, not only ones that are implemented in the PyKEEN package.