Representations

In PyKEEN, a `pykeen.nn.representation.Representation` is used to map integer indices to numeric representations. A simple example is the `pykeen.nn.representation.Embedding` class, where the mapping is a simple lookup. However, more advanced representation modules are available, too.

Message Passing

Message passing representation modules enrich the representations of entities by aggregating the information from their graph neighborhood. Example implementations from PyKEEN include `pykeen.nn.representation.RGCNRepresentation` which uses RGCN layers for enrichment, or `pykeen.nn.representation.SingleCompGCNRepresentation`, which enrich via CompGCN layers.

Another way to utilize message passing is via the modules provided in `pykeen.nn.pyg`, which allow to use the message passing layers from PyTorch Geometric to enrich base representations via message passing.

Decomposition

Since knowledge graphs may contain a large number of entities, having independent trainable embeddings for each of them may result in an excessive amount of trainable parameters. Therefore, methods have been developed, which do not learn independent representations, but rather have a set of base representations, and create individual representations by combining them.

Low-Rank Factorization

A simple method to reduce the number of parameters is to use a low-rank decomposition of the embedding matrix, as implemented in `pykeen.nn.representation.LowRankEmbeddingRepresentation`. Here, each representation is a linear combination of shared base representations. Typically, the number of bases is chosen smaller than the dimension of each base representation.

NodePiece

Another example is NodePiece, which takes inspiration from tokenization we encounter in, e.g.. NLP, and represents each entity as a set of tokens. The implementation in PyKEEN, `pykeen.nn.representation.NodePieceRepresentation`, implements a simple yet effective variant thereof, which uses a set of randomly chosen incident relations (including inverse relations) as tokens.

See also

https://towardsdatascience.com/nodepiece-tokenizing-knowledge-graphs-6dd2b91847aa

Text-based

Text-based representations use the entities' (or relations') labels to derive representations. To this end, `pykeen.nn.representation.TextRepresentation` uses a (pre-trained) transformer model from the `transformers` library to encode the labels. Since the transformer models have been trained on huge corpora of text, their text encodings often contain semantic information, i.e., labels with similar semantic meaning get similar representations. While we can also benefit from these strong features by just initializing an `pykeen.nn.representation.Embedding` with the vectors, e.g., using `pykeen.nn.init.LabelBasedInitializer`, the `pykeen.nn.representation.TextRepresentation` include the transformer model as part of the KGE model, and thus allow fine-tuning the language model for the KGE task. This is beneficial, e.g., since it allows a simple form of obtaining an inductive model, which can make predictions for entities not seen during training.

```python
from pykeen.pipeline import pipeline
from pykeen.datasets import get_dataset
from pykeen.nn import TextRepresentation
from pykeen.models import ERModel

dataset = get_dataset(dataset="nations")
entity_representations = TextRepresentation.from_dataset(
    triples_factory=dataset,
    encoder="transformer",
)
result = pipeline(
    dataset=dataset,
    model=ERModel,
    model_kwargs=dict(
        interaction="ermlpe",
        interaction_kwargs=dict(
            embedding_dim=entity_representations.shape[0],
        ),
        entity_representations=entity_representations,
        relation_representations_kwargs=dict(
            shape=entity_representations.shape,
        ),
    ),
    training_kwargs=dict(
        num_epochs=1,
    ),
)
model = result.model
```

We can use the label-encoder part to generate representations for unknown entities with labels. For instance, *"uk"* is an entity in *nations*, but we can also put in *"united kingdom"*, and get a roughly equivalent vector representations

```python
entity_representation = model.entity_representations[0]
label_encoder = entity_representation.encoder
uk, united_kingdom = label_encoder(labels=["uk", "united kingdom"])
```

Thus, if we would put the resulting representations into the interaction function, we would get similar scores

```
# true triple from train: ['brazil', 'exports3', 'uk']
relation_representation = model.relation_representations[0]
h_repr = entity_representation.get_in_more_canonical_shape(
    dim="h",
    indices=torch.as_tensor(dataset.entity_to_id["brazil"]).view(1),
)
r_repr = relation_representation.get_in_more_canonical_shape(
    dim="r",
    indices=torch.as_tensor(dataset.relation_to_id["exports3"]).view(1),
)
scores = model.interaction(
    h=h_repr,
    r=r_repr,
    t=torch.stack([uk, united_kingdom]),
)
print(scores)
```

As a downside, this will usually substantially increase the computational cost of computing triple scores.

Biomedical Entities

If your dataset is labeled with compact uniform resource identifiers (e.g., CURIEs) for biomedical entities like chemicals, proteins, diseases, and pathways, then the `pykeen.nn.representation.BiomedicalCURIERepresentation` representation can make use of `pyobo` to look up names (via CURIE) via the `pyobo.get_name()` function, then encode them using the text encoder.

All biomedical knowledge graphs in PyKEEN (at the time of adding this representation), unfortunately do not use CURIEs for referencing biomedical entities. In the future, we hope this will change.

To learn more about CURIEs, please take a look at the Bioregistry and this blog post on CURIEs.