Getting Started with NodePiece

This page gives more practical examples on using and configuring NodePiece.

Basic Usage

We'll use `pykeen.datasets.FB15k237` for illustrating purposes throughout the following examples.

```
from pykeen.models import NodePiece
from pykeen.datasets import FB15k237

# inverses are necessary for the current version of NodePiece
dataset = FB15k237(create_inverse_triples=True)
```

In the simplest usage of `pykeen.models.NodePiece`, we'll only use relations for tokenization. We can do this by with the following arguments:

1. Set the `tokenizers="RelationTokenizer"` to `pykeen.nn.node_piece.RelationTokenizer`. We can simply refer to the class name and it gets automatically resolved to the correct subclass of `pykeen.nn.node_piece.Tokenizer` by the `class_resolver`.
2. Set the `num_tokens=12` to sample 12 unique relations per node. If, for some entities, there are less than 12 unique relations, the difference will be padded with the auxiliary padding token.

Here's how the code looks:

```
model = NodePiece(
    triples_factory=dataset.training,
    tokenizers="RelationTokenizer",
    num_tokens=12,
    embedding_dim=64,
)
```

Next, we'll use a combination of tokenizers (`pykeen.nn.node_piece.AnchorTokenizer` and `pykeen.nn.node_piece.RelationTokenizer`) to replicate the full NodePiece tokenization with k anchors and m relational context. It's as easy as sending a list of tokenizers to `tokenizers` and sending a list of arguments to `num_tokens`:

```
model = NodePiece(
    triples_factory=dataset.training,
    tokenizers=["AnchorTokenizer", "RelationTokenizer"],
    num_tokens=[20, 12],
    embedding_dim=64,
)
```

Class resolver will automatically instantiate `pykeen.nn.node_piece.AnchorTokenizer` with 20 anchors per node and `pykeen.nn.node_piece.RelationTokenizer` with 12 relations per node, so the order of specifying `tokenizers` and `num_tokens` matters here.

Anchor Selection and Searching

The `pykeen.nn.node_piece.AnchorTokenizer` has two fields:

1. `selection` controls how we sample anchors from the graph (32 anchors by default)
2. `searcher` controls how we tokenize nodes using selected anchors (`pykeen.nn.node_piece.CSGraphAnchorSearcher` by default)

By default, our models above use 32 anchors selected as top-degree nodes with `pykeen.nn.node_piece.DegreeAnchorSelection` (those are default values for the anchor selection resolver) and nodes are tokenized using `pykeen.nn.node_piece.CSGraphAnchorSearcher` - it uses `scipy.sparse` to explicitly compute shortest paths from all nodes in the graph to all anchors in the deterministic manner. We can afford that for relatively small graphs of FB15k237 size.

For larger graphs, we recommend using the breadth-first search (BFS) procedure in `pykeen.nn.node_piece.ScipySparseAnchorSearcher` - it applies BFS by iteratively expanding node neighborhood until it finds a desired number of anchors - this dramatically saves compute time on graphs of size like `pykeen.datasets.OGBWikiKG2`.

32 unique anchors might be a bit too small for FB15k237 with 15k nodes - so let's create a `pykeen.models.NodePiece` model with 100 anchors selected with the top degree strategy by sending the `tokenizers_kwargs` list:

```
model = NodePiece(
    triples_factory=dataset.training,
    tokenizers=["AnchorTokenizer", "RelationTokenizer"],
    num_tokens=[20, 12],
    tokenizers_kwargs=[
        dict(
            selection="Degree",
            selection_kwargs=dict(
                num_anchors=100,
            ),
            searcher="CSGraph",
        ),
        dict(),  # empty dict for the RelationTokenizer - it doesn't need any kwargs
    ],
    embedding_dim=64,
)
```

`tokenizers_kwargs` expects the same number dictionaries as the number of tokenizers you used, so we have 2 dicts here - one for `AnchorTokenizer` and another one for `RelationTokenizer` (but this one doesn't need any kwargs so we just put an empty dict there).

Let's create a model with 500 top-pagerank anchors selected with the BFS strategy - we'll just modify the `selection` and `searcher` args:

```
model = NodePiece(
    triples_factory=dataset.training,
    tokenizers=["AnchorTokenizer", "RelationTokenizer"],
    num_tokens=[20, 12],
    tokenizers_kwargs=[
        dict(
            selection="PageRank",
            selection_kwargs=dict(
                num_anchors=500,
            ),
            searcher="ScipySparse",
        ),
        dict(),  # empty dict for the RelationTokenizer - it doesn't need any kwargs
    ],
    embedding_dim=64,
)
```

Looks nice, but fasten your seatbelts 🚀 - we can use several anchor selection strategies sequentially to select more diverse anchors! Mindblowing 😍

Let's create a model with 500 anchors where 50% of them will be top degree nodes and another 50% will be top PageRank nodes - for that we have a `pykeen.nn.node_piece.MixtureAnchorSelection` class!

```
model = NodePiece(
    triples_factory=dataset.training,
    tokenizers=["AnchorTokenizer", "RelationTokenizer"],
    num_tokens=[20, 12],
    tokenizers_kwargs=[
        dict(
            selection="MixtureAnchorSelection",
            selection_kwargs=dict(
                selections=["degree", "pagerank"],
                ratios=[0.5, 0.5],
                num_anchors=500,
            ),
            searcher="ScipySparse",
        ),
        dict(),  # empty dict for the RelationTokenizer - it doesn't need any kwargs
    ],
    embedding_dim=64,
)
```

Now the `selection_kwargs` controls which strategies we'll be using and how many anchors each of them will sample - in our case `selections=['degree', 'pagerank']`. Using the `ratios` argument we control the ratio of those sampled anchors in the total pool - in our case `ratios=[0.5, 0.5]` which means that both `degree` and `pagerank` strategies each will sample 50% from the total number of anchors. Since the total number is 500, there will be 250 top-degree anchors and 250 top-pagerank anchors. `ratios` **must** sum up to 1.0

Important: sampled anchors are **unique** - that is, if a node appears to be in top-K degree and top-K pagerank, it will be used only once, the sampler will just skip it in the subsequent strategies.

At the moment, we have 3 anchor selection strategies: **degree**, **pagerank**, and **random**. The latter just samples random nodes as anchors.

Let's create a tokenization setup reported in the original NodePiece paper for FB15k237 with 40% top degree anchors, 40% top pagerank, and 20% random anchors:

```
model = NodePiece(
    triples_factory=dataset.training,
    tokenizers=["AnchorTokenizer", "RelationTokenizer"],
    num_tokens=[20, 12],
    tokenizers_kwargs=[
        dict(
            selection="MixtureAnchorSelection",
            selection_kwargs=dict(
                selections=["degree", "pagerank", "random"],
                ratios=[0.4, 0.4, 0.2],
                num_anchors=500,
            ),
            searcher="ScipySparse",
        ),
        dict(),  # empty dict for the RelationTokenizer - it doesn't need any kwargs
    ],
    embedding_dim=64,
)
```

Note on Anchor Distances: As of now, the anchor distances are considered implicitly, i.e., when performing actual tokenization via shortest paths or BFS we do sort anchors by proximity and keep top-K nearest. The anchor distance embedding as a positional feature to be added to anchor embedding is not yet implemented.

How many total anchors *num_anchors* and anchors & relations *num_tokens* do I need for my graph?

This is a good question with deep theoretical implications and NP-hard problems like k-Dominating Sets and Vertex Cover Sets . We don't have a closed-form solution for each possible dataset, but we found some empirical heuristics:

- keeping `num_anchors` as 1-10% of total nodes in the graph is a good start
- graph density is a major factor: the denser the graph, the fewer `num_anchors` you'd need. For dense FB15k237 100 total anchors (over 15k total nodes) seems to be good enough, while for sparser WN18RR we needed at least 500 anchors (over 40k total nodes). For dense OGB WikiKG2 of 2.5M nodes a vocab of 20K anchors (< 1%) already leads to SOTA results
- the same applies to anchors per node: you'd need more tokens for sparser graphs and fewer for denser
- the size of the relational context depends on the density and number of unique relations in the graph, eg, in FB15k237 we have 237 * 2 = 474 unique relations and only 11 * 2 = 22 in WN18RR. If we select a too large context, most tokens will be `PADDING_TOKEN` and we don't want that.
- reported relational context sizes (relations per node) in the NodePiece paper are 66th percentiles of the number of unique incident relations per node, eg 12 for FB15k237 and 5 for WN18RR

In some tasks, you might not need anchors at all and could use RelationTokenizer only! Check the paper for more results.

- In inductive link prediction tasks we don't use anchors as inference graphs are disconnected from training ones;
- in relation prediction we found that just a relational context is better than anchors + relations;
- in node classification (currently, this pipeline is not available in PyKEEN) on dense relation-rich graphs like Wikidata, we found that just a relational context is better than anchors + relations.

Using NodePiece with `pykeen.pipeline.pipeline()`

Let's pack the last NodePiece model into the pipeline:

```
import torch.nn

from pykeen.models import NodePiece
from pykeen.pipeline import pipeline

result = pipeline(
    dataset="fb15k237",
    dataset_kwargs=dict(
        create_inverse_triples=True,
    ),
    model=NodePiece,
    model_kwargs=dict(
        tokenizers=["AnchorTokenizer", "RelationTokenizer"],
        num_tokens=[20, 12],
        tokenizers_kwargs=[
            dict(
                selection="MixtureAnchorSelection",
                selection_kwargs=dict(
                    selections=["degree", "pagerank", "random"],
                    ratios=[0.4, 0.4, 0.2],
                    num_anchors=500,
                ),
                searcher="ScipySparse",
            ),
            dict(),  # empty dict for the RelationTokenizer - it doesn't need any kwargs
        ],
        embedding_dim=64,
        interaction="rotate",
    ),
)
```

Pre-Computed Vocabularies

We have a `pykeen.nn.node_piece.PrecomputedPoolTokenizer` that can be instantiated with a precomputed vocabulary either from a local file or using a downloadable link.

For a local file, specify `path`:

```
precomputed_tokenizer = tokenizer_resolver.make(
    "precomputedpool", path=Path("path/to/vocab.pkl")
)

model = NodePiece(
    triples_factory=dataset.training,
    num_tokens=[20, 12],
    tokenizers=[precomputed_tokenizer, "RelationTokenizer"],
)
```

For a remote file, specify the `url`:

```
precomputed_tokenizer = tokenizer_resolver.make(
    "precomputedpool", url="http://link/to/vocab.pkl"
)
```

Generally, `pykeen.nn.node_piece.PrecomputedPoolTokenizer` can use any `pykeen.nn.node_piece.PrecomputedTokenizerLoader` as a custom processor of vocabulary formats. Right now there is one such loader, `pykeen.nn.node_piece.GalkinPrecomputedTokenizerLoader` that expects a dictionary of the following format:

```
node_id: {
    "ancs": [a list of used UNMAPPED anchor nodes sorted from nearest to farthest],
    "dists": [a list of anchor distances for each anchor in ancs, ascending]
}
```

As of now, we don't use anchor distances, but we expect the anchors in `ancs` to be already sorted from nearest to farthest, so the example of a precomputed vocab can be:

```
1: {'ancs': [3, 10, 5, 9, 220, ...]}  # anchor 3 is the nearest for node 1
2: {'ancs': [22, 37, 14, 10, ...]}  # anchors 22 is the nearest for node 2
```

Unmapped anchors means that anchor IDs are the same node IDs from the total set of entities `0... N-1`. In the pickle processing we'll convert them to a contiguous range `0 ... num_anchors-1`. Any negative indices in the lists will be treated as padding tokens (we used -99 in the precomputed vocabularies).

The original NodePiece repo has an example of building such a vocabulary format for OGB WikiKG 2.

Configuring the Interaction Function

you can use literally any interaction function available in PyKEEN as a scoring function! By default, NodePiece uses DistMult, but it's easy to change as in any `pykeen.models.ERModel`, let's use the RotatE interaction:

```
model = NodePiece(
    triples_factory=dataset.training,
    tokenizers=["AnchorTokenizer", "RelationTokenizer"],
    num_tokens=[20, 12],
    interaction="rotate",
    embedding_dim=64,
)
```

Well, for RotatE we might want to initialize relations as phases (`init_phases`) and use an additional relation constrainer to keep `|r| = 1` (`complex_normalize`), and use `xavier_uniform_` for anchor embedding initialization - let's add that, too:

```
model = NodePiece(
    triples_factory=dataset.training,
    tokenizers=["AnchorTokenizer", "RelationTokenizer"],
    num_tokens=[20, 12],
    embedding_dim=64,
    interaction="rotate",
    relation_initializer="init_phases",
    relation_constrainer="complex_normalize",
    entity_initializer="xavier_uniform_",
)
```

Configuring the Aggregation Function

This section is about the `aggregation` keyword argument. This is an encoder function that actually builds entity representations from token embeddings. It is supposed to be a function that maps a set of tokens (anchors, relations, or both) to a single vector:

$$f([a_1, a_2, \dots, a_k, r_1, r_2, \dots, r_m]) \in \mathbb{R}^{(k+m)\times d} \to \mathbb{R}^d$$

Right now, by default we use a simple 2-layer MLP (`pykeen.nn.perceptron.ConcatMLP`) that concatenates all tokens to one long vector and projects it down to model's embedding dimension:

```
hidden_dim = int(ratio * embedding_dim)
super().__init__(
    nn.Linear(num_tokens * embedding_dim, hidden_dim),
    nn.Dropout(dropout),
    nn.ReLU(),
    nn.Linear(hidden_dim, embedding_dim),
)
```

Aggregation can be parameterized with any neural network (`torch.nn.Module`) that would return a single vector from a set of inputs. Let's be fancy 😎 and create a DeepSet encoder:

```
class DeepSet(torch.nn.Module):
    def __init__(self, hidden_dim=64):
        super().__init__()
        self.encoder = torch.nn.Sequential(
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim),
        )
        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim),
        )

    def forward(self, x, dim=-2):
        x = self.encoder(x).mean(dim)
        x = self.decoder(x)
        return x


model = NodePiece(
    triples_factory=dataset.training,
    tokenizers=["AnchorTokenizer", "RelationTokenizer"],
    num_tokens=[20, 12],
    embedding_dim=64,
    interaction="rotate",
    relation_initializer="init_phases",
    relation_constrainer="complex_normalize",
    entity_initializer="xavier_uniform_",
    aggregation=DeepSet(hidden_dim=64),
)
```

We can even put a Transformer with pooling here. The only thing to keep in mind is the complexity of the encoder - we found `pykeen.nn.perceptron.ConcatMLP` to be a good balance between speed and final performance, although at the cost of being not permutation invariant to the input set of tokens.

The aggregation function resembles that of GNNs. Non-parametric avg/min/max did not work that well in the current tokenization setup, so some non-linearity is definitely useful - hence the choice for MLP / DeepSets / Transformer as an aggregation function.

Let's wrap our cool NodePiece model with 40/40/20 degree/pagerank/random tokenization with the BFS searcher and DeepSet aggregation into a pipeline:

```
result = pipeline(
    dataset="fb15k237",
    dataset_kwargs=dict(
        create_inverse_triples=True,
    ),
    model=NodePiece,
    model_kwargs=dict(
        tokenizers=["AnchorTokenizer", "RelationTokenizer"],
        num_tokens=[20, 12],
        tokenizers_kwargs=[
            dict(
                selection="MixtureAnchorSelection",
                selection_kwargs=dict(
                    selections=["degree", "pagerank", "random"],
                    ratios=[0.4, 0.4, 0.2],
                    num_anchors=500,
                ),
                searcher="ScipySparse",
            ),
            dict(),  # empty dict for the RelationTokenizer - it doesn't need any kwargs
        ],
        embedding_dim=64,
        interaction="rotate",
        relation_initializer="init_phases",
        relation_constrainer="complex_normalize",
        entity_initializer="xavier_uniform_",
        aggregation=DeepSet(hidden_dim=64),
    ),
)
```

NodePiece + GNN

It is also possible to add a message passing GNN on top of obtained NodePiece representations to further enrich node states - we found it shows even better results in inductive LP tasks. We have that implemented with `pykeen.models.InductiveNodePieceGNN` that uses a 2-layer CompGCN encoder - please check the Inductive Link Prediction tutorial.

Tokenizing Large Graphs with METIS

Mining anchors and running tokenization on whole graphs larger than 1M nodes might be computationally expensive. Due to the inherent locality of NodePiece, i.e., tokenization via nearest anchors and incident relations, we recommend using graph partitioning to reduce time and memory costs of tokenization. With graph partitioning, anchor search and tokenization can be performed independently within each partition with a final merging of all results into a single vocabulary.

We designed the partitioning tokenization strategy using METIS, a min-cut graph partitioning algorithm with an efficient implementation available in torch-sparse. Along with METIS, we leverage *torch-sparse* to offer a new, faster BFS procedure that can run on a GPU.

The main tokenizer class is `pykeen.nn.node_piece.MetisAnchorTokenizer`. You can place it instead of the vanilla `AnchorTokenizer`. With the Metis-based tokenizer, we first partition the input training graph into *k* separate partitions and then run anchor selection and anchor search sequentially and independently **for each partition**.

You can use any existing anchor selection and anchor search strategy described above although for larger graphs we recommend using a new `pykeen.nn.node_piece.SparseBFSSearcher` as anchor searcher – it implements faster sparse matrix multiplication kernels and can be run on a GPU. The only difference from the vanilla tokenizer is that now the `num_anchors` argument defines how many anchors will be mined **for each partition**.

The new tokenizer has two special arguments:

- `num_partitions` - number of partitions the graph will be divided into. You can expect METIS to produce partitions of about the same size, e.g., `num_partitions=10` for a graph of 1M nodes would produce 10 partitions with about 100K nodes in each. The total number of mined anchors will be `num_partitions * num_anchors`
- `device` - the device to run METIS on. It can be different from the device on which an `AnchorSearcher` will run. We found `device="cpu"` works faster on larger graphs and does not require limited GPU memory, although you can keep the device to be resolved automatically or put `device="cuda"` to try running it on a GPU.

It is still advisable to run large graph tokenization using `pykeen.nn.node_piece.SparseBFSSearcher` on a GPU thanks to more efficient sparse CUDA kernels. If a GPU is available, it will be used automatically by default.

Let's use the new tokenizer for the Wikidata5M graph of 5M nodes and 20M edges.

```python
from pykeen.datasets import Wikidata5M

dataset = Wikidata5M(create_inverse_triples=True)

model = NodePiece(
    triples_factory=dataset.training,
    tokenizers=["MetisAnchorTokenizer", "RelationTokenizer"],
    num_tokens=[20, 12],  # 20 anchors per node in for the Metis strategy
    embedding_dim=64,
    interaction="rotate",
    tokenizers_kwargs=[
        dict(
            num_partitions=20,  # each partition will be of about 5M / 20 = 250K nodes
            device="cpu",  # METIS on cpu tends to be faster
            selection="MixtureAnchorSelection",  # we can use any anchor selection strategy
here
            selection_kwargs=dict(
                selections=['degree', 'random'],
                ratios=[0.5, 0.5],
                num_anchors=1000,  # overall, we will have 20 * 1000 = 20000 anchors
            ),
            searcher="SparseBFSSearcher",  # a new efficient anchor searcher
            searcher_kwargs=dict(
                max_iter=5  # each node will be tokenized with anchors in the 5-hop
neighborhood
            )
        ),
        dict()
    ],
    aggregation="mlp"
)

# we can save the vocabulary of tokenized nodes
from pathlib import Path
model.entity_representations[0].base[0].save_assignment(Path("./anchors_assignment.pt"))
```

On a machine with 32 GB RAM and 32 GB GPU, processing of Wikidata5M takes about 10 minutes:

- ~ 3 min for partitioning into 20 clusters on a cpu;
- ~ 7 min overall for anchor selection and search in each partition

How many partitions do I need for my graph?

It largely depends on the hardware and memory at hand, but as a rule of thumb we would recommend having partitions of size < 500K nodes each