Performance Tricks

PyKEEN uses a combination of techniques to promote efficient calculations during training/evaluation and tries to maximize the utilization of the available hardware (currently focused on single GPU usage).

Entity and Relation IDs

Entities and relations in triples are usually stored as strings. Because KGEMs aim at learning vector representations for these entities and relations such that the chosen interaction function learns a useful scoring on top of them, we need a mapping from the string representations to vectors. Moreover, for computational efficiency, we would like to store all entity/relation embeddings in matrices. Thus, the mapping process comprises two parts: Mapping strings to IDs, and using the IDs to access the embeddings (=row indices).

In PyKEEN, the mapping process takes place in `pykeen.triples.TriplesFactory`. The triples factory maintains the sets of unique entity and relation labels and ensures that they are mapped to unique integer IDs on $[0, \text{num\_unique\_entities})$ for entities and $[0, \text{num\_unique\_relations})$. The mappings are respectively accessible via the attributes :data: `pykeen.triples.TriplesFactory.entity_label_to_id` and :data: `pykeen.triples.TriplesFactory.relation_label_to_id`.

To improve the performance, the mapping process takes place only once, and the ID-based triples are stored in a tensor :data: `pykeen.triples.TriplesFactory.mapped_triples`.

Tuple Broadcasting

Interaction functions are usually only given for the standard case of scoring a single triple (h, r, t). This function is in PyKEEN implemented in the `pykeen.models.base.Model.score_hrt()` method of each model, e.g. `pykeen.models.DistMult.score_hrt()` for `pykeen.models.DistMult`. When training under the local closed world assumption (LCWA), evaluating a model, and performing the link prediction task, the goal is to score all entities/relations for a given tuple, i.e. (h, r), (r, t) or (h, t). In these cases a single tuple is used many times for different entities/relations.

For example, we want to rank all entities for a single tuple (h, r) with `pykeen.models.DistMult` for the `pykeen.datasets.FB15k237`. This dataset contains 14,505 entities, which means that there are 14,505 (h, r, t) combinations, whereas h and r are constant. Looking at the interaction function of `pykeen.models.DistMult`, we can observe that the $h \odot r$ part causes half of the mathematical operations to calculate $h \odot r \odot t$. Therefore, calculating the $h \odot r$ part only once and reusing it spares us half of the mathematical operations for the other 14,504 remaining entities, making the calculations roughly twice as fast in total. The speed-up might be significantly higher in cases where the broadcasted part has a high relative complexity compared to the overall interaction function, e.g. `pykeen.models.ConvE`.

To make this technique possible, PyKEEN models have to provide an explicit broadcasting function via following methods in the model class:

- `pykeen.models.base.Model.score_h()` - Scoring all possible head entities for a given (r, t) tuple
- `pykeen.models.base.Model.score_r()` - Scoring all possible relations for a given (h, t) tuple
- `pykeen.models.base.Model.score_t()` - Scoring all possible tail entities for a given (h, r) tuple

The PyKEEN architecture natively supports these methods and makes use of this technique wherever possible without any additional modifications. Providing these methods is completely optional and not required when implementing new models.

Filtering with Index-based Masking

In this example, it is given a knowledge graph $\mathcal{K} \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$ and disjoint unions of $\mathcal{K}$ in training triples $\mathcal{K}_{train}$, testing triples $\mathcal{K}_{test}$, and validation triples $\mathcal{K}_{val}$. The same operations are performed on $\mathcal{K}_{test}$ and $\mathcal{K}_{val}$, but only $\mathcal{K}_{test}$ will be given as example in this section.

Two calculations are performed for each test triple $(h, r, t) \in \mathcal{K}_{test}$ during standard evaluation of a knowledge graph embedding model with interaction function $f : \mathcal{E} \times \mathcal{R} \times \mathcal{E} \rightarrow \mathbb{R}$ for the link prediction task:

1. (h, r) is combined with all possible tail entities $t' \in \mathcal{E}$ to make triples $T_{h,r} = \{(h, r, t') \mid t' \in \mathcal{E}\}$
2. (r, t) is combined with all possible head entities $h' \in \mathcal{E}$ to make triples $H_{r,t} = \{(h', r, t) \mid h' \in \mathcal{E}\}$

Finally, the ranking of (h, r, t) is calculated against all $(h, r, t') \in T_{h,r}$ and $(h', r, t) \in H_{r,t}$ triples with respect to the interaction function f.

In the filtered setting, $T_{h,r}$ is not allowed to contain tail entities $(h, r, t') \in \mathcal{K}_{train}$ and $H_{r,t}$ is not allowed to contain head entities leading to $(h', r, t) \in \mathcal{K}_{train}$ triples found in the train dataset. Therefore, their definitions could be amended like:

- $T^{\text{filtered}}_{h,r} = \{(h, r, t') \mid t' \in \mathcal{E}\} \setminus \mathcal{K}_{train}$
- $H^{\text{filtered}}_{r,t} = \{(h', r, t) \mid h' \in \mathcal{E}\} \setminus \mathcal{K}_{train}$

While this easily defined theoretically, it poses several practical challenges. For example, it leads to the computational challenge that all new possible triples $(h, r, t') \in T_{h,r}$ and $(h', r, t) \in H_{r,t}$ must be enumerated and then checked for existence in $\mathcal{K}_{train}$. Considering a dataset like `pykeen.datasets.FB15k237` that has almost 15,000 entities, each test triple $(h, r, t) \in \mathcal{K}_{test}$ leads to $2 * |\mathcal{E}| = 30,000$ possible new triples, which have to be checked against the train dataset and then removed.

To obtain very fast filtering, PyKEEN combines the technique presented above in Entity and Relation IDs and Tuple Broadcasting together with the following mechanism, which in our case has led to a 600,000 fold increase in speed for the filtered evaluation compared to the mechanisms used in previous versions.

As a starting point, PyKEEN will always compute scores for all triples in $H_{r,t}$ and $T_{h,r}$, even in the filtered setting. Because the number of positive triples on average is very low, few results have to be removed. Additionally, due to the technique presented in Tuple Broadcasting, scoring extra entities has a marginally low cost. Therefore, we start with the score vectors from `pykeen.models.base.Model.score_t()` for all triples $(h, r, t') \in H_{r,t}$ and from `pykeen.models.base.Model.score_h()` for all triples $(h', r, t) \in T_{h,r}$.

Following, the sparse filters $\mathbf{f}_t \in \mathbb{B}^{|\mathcal{E}|}$ and $\mathbf{f}_h \in \mathbb{B}^{|\mathcal{E}|}$ are created, which state which of the entities would lead to triples found in the train dataset. To achieve this we will rely on the technique presented in Entity and Relation IDs, i.e. all entity/relation IDs correspond to their exact position in the respective embedding tensor. As an example we take the tuple (h, r) from the test triple $(h, r, t) \in \mathcal{K}_{test}$ and are interested in all tail entities t' that should be removed from $T_{h,r}$ in order to obtain $T_{h,r}^{\text{filtered}}$. This is achieved by performing the following steps:

1. Take r and compare it to the relations of all triples in the train dataset, leading to a boolean vector of the size of number of triples contained in the train dataset, being true where any triple had the relation r
2. Take h and compare it to the head entities of all triples in the train dataset, leading to a boolean vector of the size of number of triples contained in the train dataset, being true where any triple had the head entity h
3. Combine both boolean vectors, leading to a boolean vector of the size of number of triples contained in the train dataset, being true where any triple had both the head entity h and the relation r
4. Convert the boolean vector to a non-zero index vector, stating at which indices the train dataset contains triples that contain both the head entity h and the relation r, having the size of the number of non-zero elements
5. The index vector is now applied on the tail entity column of the train dataset, returning all tail entity IDs t' that combined with h and r lead to triples contained in the train dataset
6. Finally, the t' tail entity ID index vector is applied on the initially mentioned vector returned by `pykeen.models.base.Model.score_t()` for all possible triples (h, r, t') and all affected scores are set to `float('nan')` following the IEEE-754 specification, which makes

these scores non-comparable, effectively leading to the score vector for all possible novel triples $(h, r, t') \in T_{h,r}^{\text{filtered}}$.

$H_{r,t}^{\text{filtered}}$ is obtained from $H_{r,t}$ in a similar fashion.

Sub-batching & Slicing

With growing model and dataset sizes the KGEM at hand is likely to exceed the memory provided by GPUs. Especially during training it might be desired to train using a certain batch size. When this batch size is too big for the hardware at hand, PyKEEN allows to set a sub-batch size in the range of $[1, \text{batch\_size}]$. When the sub-batch size is set, PyKEEN automatically accumulates the gradients after each sub-batch and clears the computational graph during training. This allows to train KGEMs on GPU that otherwise would be too big for the hardware at hand, while the obtained results are identical to training without sub-batching.

Note

In order to guarantee equivalent results, not all models support sub-batching, since certain components, e.g. batch normalization, require the entire batch to be calculated in one pass to avoid altering statistics.

Note

Sub-batching is sometimes also called *Gradient Accumulation*, e.g., by huggingface's transformer library, since we accumulate the gradients over multiple sub-batches before updating the parameters.

For some large configurations, even after applying the sub-batching trick, out-of-memory errors may still occur. In this case, PyKEEN implements another technique, called *slicing*. Note that we often compute more than one score for each batch element: in sLCWA, we have $1 + \text{num\_negative\_samples}$ scores, and in LCWA, we have num_entities scores for each batch element. In slicing, we do not compute all of these scores at once, but rather in smaller "batches". For old-style models, i.e., those subclassing from `pykeen.models.base._OldAbstractModel`, this has to be implemented individually for each of them. New-style models, i.e., those deriving from `pykeen.models.nbase.ERModel` have a generic implementation enabling slicing for *all* interactions.

Note

Slicing computes the scores in smaller batches, but still needs to compute the gradient over all scores, since some loss functions require access to them.

Automated Memory Optimization

Allowing high computational throughput while ensuring that the available hardware memory is not exceeded during training and evaluation requires the knowledge of the maximum possible training and evaluation batch size for the current model configuration. However, determining the training and evaluation batch sizes is a tedious process, and not feasible when a large set of heterogeneous experiments are run. Therefore, PyKEEN has an automatic memory optimization step that computes the maximum possible training and evaluation batch sizes for the current model configuration and available hardware before the actual calculation starts. If the user-provided batch size is too large for the used hardware, the automatic memory optimization determines the maximum sub-batch size for training and accumulates the gradients with the above described process Sub-batching & Slicing. The batch sizes are determined using binary search taking into consideration the CUDA architecture which ensures that the chosen batch size is the most CUDA efficient one.

Evaluation Fallback

Usually the evaluation is performed on the GPU for faster speeds. In addition, users might choose a batch size upfront in their evaluation configuration to fully utilize the GPU to achieve the fastest evaluation speeds possible. However, during larger setups testing different model configurations and dataset partitions such as e.g. HPO the hardware requirements might change drastically, which might cause that the evaluation no longer can be run with the pre-set batch size or not on the GPU at all for larger datasets and memory intense models. Since PyKEEN will abide by the user configurations, the evaluation will crash in these cases even though the training finished successfully and thus loose the progress achieved and/or leave trials unfinished. Given that the batch size and the device have no impact on the evaluation results, PyKEEN offers a way to overcome this problem through the evaluation fallback option of the pipeline. This will cause the evaluation to fall back to using a smaller batch size in cases where the evaluation failed using the GPU with a set batch size and in the last instance to evaluate on the CPU, if even the smallest possible batch size is too big for the GPU. Note: This can lead to significantly longer evaluation times in cases where the evaluation falls back to using the CPU.