Optimizing a Model's Hyper-parameters

The easiest way to optimize a model is with the `pykeen.hpo.hpo_pipeline()` function.

All of the following examples are about getting the best model when training `pykeen.models.TransE` on the `pykeen.datasets.Nations` dataset. Each gives a bit of insight into usage of the `hpo_pipeline()` function.

The minimal usage of the hyper-parameter optimization is to specify the dataset, the model, and how much to run. The following example shows how to optimize the TransE model on the Nations dataset a given number of times using the `n_trials` argument.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     dataset='Nations',
...     model='TransE',
... )
```

Alternatively, the `timeout` can be set. In the following example, as many trials as possible will be run in 60 seconds.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     timeout=60,
...     dataset='Nations',
...     model='TransE',
... )
```

The hyper-parameter optimization pipeline has the ability to optimize hyper-parameters for the corresponding `*_kwargs` arguments in the `pykeen.pipeline.pipeline()`:

- `model`
- `loss`
- `regularizer`
- `optimizer`
- `negative_sampler`
- `training`

Defaults

Each component's hyper-parameters have a reasonable default values. For example, every model in PyKEEN has default for its hyper-parameters chosen from the best-reported values in each model's original paper unless otherwise stated on the model's reference page. In case hyper-parameters for a model for a specific dataset were not available, we choose the hyper-parameters based on the findings in our large-scale benchmarking [ali2020a]. For most components (e.g., models, losses, regularizers, negative samples, training loops), these values are stored in the default valeues of the respective classes' *__init__()* functions. They can be viewed in the corresponding reference section of the docs.

Some components contain strategies for doing hyper-parameter optimization. When you call the `pykeen.hpo.hpo_pipeline()`, the following steps are taken to determine what happens for each hyper-parameter in each componenent:

1. If an explicit value was passed, use it.
2. If no explicit value was passed and an HPO strategy was passed, use the explicit strategy.
3. If no explicit value was passed and no HPO strategy was passed and there is a default HPO strategy, use the default strategy.
4. If no explicit value was passed, no HPO strategy was passed, and there is no default HPO strategy, use the default hyper-parameter value
5. If no explicit value was passed, no HPO strategy was passed, and there is no default HPO strategy, and there is no default hyper-parameter value, raise an `TypeError`

For example, the TransE model's default HPO strategy for its `embedding_dim` argument is to search between $[16, 256]$ with a step size of 16. The l_p norm is set to search as either 1 or 2. This will be overridden with 50 in the following code:

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='TransE',
...     model_kwargs=dict(embedding_dim=50),
... )
```

The strategy can be explicitly overridden with:

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='TransE',
...     model_kwargs_ranges=dict(
...         embedding_dim=dict(type=int, low=16, high=256, step=32),
...     ),
... )
```

Each model, loss, regularizer, negative sampler, and training loop specify a class variable called `hpo_defaults` in which there's a dictionary with all of the default strategies. They keys match up to the arguments in their respective `__init__()` functions.

Since optimizers aren't re-implemented in PyKEEN, there's a specfic dictionary at `pykeen.optimizers.optimizers_hpo_defaults` containing their strategies. It's debatable whether you should optimize the optimizers (yo dawg), so you can always choose to set the learning rate `lr` to a constant value.

Strategies

An HPO strategy is a Python `dict` with a `type` key corresponding to a categorical variable, boolean variable, integer variable, or floating point number variable. The value itself for `type` should be one of the following:

1. `"categorical"`
2. `bool` or `"bool"`
3. `int` or `"int"`
4. `float` or `"float"`

Several strategies can be grouped together in a dictionary where the key is the name of the hyper-parameter for the component in the `*_kwargs_ranges` arguments to the HPO pipeline.

Categorical

The only other key to use inside a categorical variable is `choices`. For example, if you want to choose between Kullback-Leibler divergence or expected likelihood as similarity used in the KG2E model, you can write a strategy like:

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='KG2E',
...     model_kwargs_ranges=dict(
...         dist_similarity=dict(type='categorical', choices=['KL', 'EL']),
...     ),
... )
```

Boolean

The boolean variable actually doesn't need any extra keys besides the type, so a strategy for a boolean variable always looks like `dict(type='bool')`. Under the hood, this is automatically translated to a categorical variable with `choices=[True, False]`.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='TransE',
...     training_loop='sLCWA',
...     negative_sampler_kwargs_ranges=dict(
...         filtered=dict(type=boolean),
...     ),
... )
```

Integers and Floating Point Numbers

The integer and floating point number strategies share several aspects. Both require a `low` and `high` entry like in `dict(type=float, low=0.0, high=1.0)` or `dict(type=int, low=1, high=10)`.

Linear Scale

By default, you don't need to specify a `scale`, but you can be explicit by setting `scale='linear'`. This behavior should be self explanatory - there is no rescaling and you get back uniform distribution within the bounds specified by the `low` and `high` arguments. This applies to both `type=int` and `type=float`. The following example uniformly choose from [1,100]:

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='TransE',
...     training_loop='sLCWA',
...     negative_sampler_kwargs_ranges=dict(
...         num_negs_per_pos=dict(type=int, low=1, high=100),
...     ),
... )
```

Power Scale (`type=int` only)

The power scale was originally implemented as `scale='power_two'` to support `pykeen.models.ConvE`'s `output_channels` parameter. However, using two as a base is a bit limiting, so we also implemented a more general `scale='power'` where you can set set `base`. Here's an example to optimize over the number of negatives per positive ratio using *base=10*:

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='TransE',
...     training_loop='sLCWA',
...     negative_sampler_kwargs_ranges=dict(
...         num_negs_per_pos=dict(type=int, scale='power', base=10, low=0, high=2),
...     ),
... )
```

The power scale can only be used with *type=int* and not bool, categorical, or float. I like this scale because it can quickly discretize a large search space. In this example, you will get *[10**0, 10**1, 10**2]* as choices then uniformly choose from them.

Logarithmic Reweighting

The evil twin to the power scale is logarithmic reweighting on the linear scale. This is applicable `type=int` and `type=float`. Rather than changing the choices themselves, the log scale uses Optuna's built in `log` functionality to reassign the probabilities uniformly over the log'd distribution. The same example as above could be accomplished with:

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='TransE',
...     training_loop='sLCWA',
...     negative_sampler_kwargs_ranges=dict(
...         num_negs_per_pos=dict(type=int, low=1, high=100, log=True),
...     ),
... )
```

but this time, it's not discretized. However, you're just as likely to pick from $[1, 10]$ as $[10, 100]$.

Stepping

With the linear scale, you can specify the `step` size. This discretizes the distribution in linear space, so if you want to pick from $10, 20, \ldots 100$, you can do:

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='TransE',
...     training_loop='sLCWA',
...     negative_sampler_kwargs_ranges=dict(
...         num_negs_per_pos=dict(type=int, low=10, high=100, step=10),
...     ),
... )
```

This actually also works with logarithmic reweighting, since it is still technically on a linear scale, but with probabilites reweighted logarithmically. So now you'd pick from one of $[10]$ or $[20, 30, 40, \ldots, 100]$ with the same probability

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='TransE',
...     training_loop='sLCWA',
...     negative_sampler_kwargs_ranges=dict(
...         num_negs_per_pos=dict(type=int, low=10, high=100, step=10, log=True),
...     ),
... )
```

Custom Strategies

While the default values for hyper-parameters are encoded with the python syntax for default values of the `__init__()` function of each model, the ranges/scales can be found in the class variable `pykeen.models.Model.hpo_default`. For example, the range for TransE's embedding dimension is set to optimize between 50 and 350 at increments of 25 in `pykeen.models.TransE.hpo_default`. TransE also has a scoring function norm that will be optimized by a categorical selection of {1, 2} by default.

> **Note**
>
> These hyper-parameter ranges were chosen as reasonable defaults for the benchmark datasets FB15k-237 / WN18RR. When using different datasets, the ranges might be suboptimal.

All hyper-parameters defined in the `hpo_default` of your chosen model will be optimized by default. If you already have a value that you're happy with for one of them, you can specify it with the `model_kwargs` attribute. In the following example, the `embedding_dim` for a TransE model is fixed at 200, while the rest of the parameters will be optimized using the pre-defined HPO strategies in the model. For TransE, that means that the scoring function norm will be optimized as 1 or 2.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     model='TransE',
...     model_kwargs=dict(
...         embedding_dim=200,
...     ),
...     dataset='Nations',
...     n_trials=30,
... )
```

If you would like to set your own HPO strategy for the model's hyperparameters, you can do so with the `model_kwargs_ranges` argument. In the example below, the embeddings are searched over a larger range (`low` and `high`), but with a higher step size (`q`), such that 100, 200, 300, 400, and 500 are searched.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_result = hpo_pipeline(
...     n_trials=30,
...     dataset='Nations',
...     model='TransE',
...     model_kwargs_ranges=dict(
...         embedding_dim=dict(type=int, low=100, high=500, q=100),
...     ),
... )
```

Warning

If the given range is not divisible by the step size, then the upper bound will be omitted.

If you want to optimize the entity initializer, you can use the `type='categorical'` type, which requres a `choices=[...]` key with a list of choices. This works for strings, integers, floats, etc.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_result = hpo_pipeline(
...     n_trials=30,
...     dataset='Nations',
...     model='TransE',
...     model_kwargs_ranges=dict(
...         entity_initializer=dict(type='categorical', choices=[
...             'xavier_uniform',
...             'xavier_uniform_norm',
...             'uniform',
...         ]),
...     ),
... )
```

The same could be used for constrainers, normalizers, and regularizers over both entities and relations. However, different models might have different names for the initializer, normalizer, constrainer and regularizer since there could be multiple representations for either the entity, relation, or both. Check your desired model's documentation page for the kwargs that you can optimize over.

Keys of `pykeen.nn.representation.initializers` can be passed as initializers as strings and keys of `pykeen.nn.representation.constrainers` can be passed as constrainers as strings.

The HPO pipeline does not support optimizing over the hyper-parameters for each initializer. If you are interested in this, consider rolling your own ablation study pipeline.

Optimizing the Loss

While each model has its own default loss, you can explicitly specify a loss the same way as in `pykeen.pipeline.pipeline()`.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     dataset='Nations',
...     model='TransE',
...     loss='MarginRankingLoss',
... )
```

As stated in the documentation for `pykeen.pipeline.pipeline()`, each model specifies its own default loss function in `pykeen.models.Model.loss_default`. For example, the TransE model defines the margin ranking loss as its default in `pykeen.models.TransE.loss_default`.

Each model also specifies default hyper-parameters for the loss function in `pykeen.models.Model.loss_default_kwargs`. For example, DistMultLiteral explicitly sets the margin to *0.0* in `pykeen.models.DistMultLiteral.loss_default_kwargs`.

Unlike the model's hyper-parameters, the models don't store the strategies for optimizing the loss functions' hyper-parameters. The pre-configured strategies are stored in the loss function's class variable `pykeen.models.Loss.hpo_default`.

However, similarily to how you would specify `model_kwargs_ranges`, you can specify the `loss_kwargs_ranges` explicitly, as in the following example.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     dataset='Nations',
...     model='TransE',
...     loss='MarginRankingLoss',
...     loss_kwargs_ranges=dict(
...         margin=dict(type=float, low=1.0, high=2.0),
...     ),
... )
```

Optimizing the Negative Sampler

When the stochastic local closed world assumption (sLCWA) training approach is used for training, a negative sampler (subclass of `pykeen.sampling.NegativeSampler`) is chosen. Each has a strategy stored in `pykeen.sampling.NegativeSampler.hpo_default`.

Like models and regularizers, the rules are the same for specifying `negative_sampler`, `negative_sampler_kwargs`, and `negative_sampler_kwargs_ranges`.

Optimizing the Optimizer

Yo dawg, I heard you liked optimization, so we put an optimizer around your optimizer so you can optimize while you optimize. Since all optimizers used in PyKEEN come from the PyTorch implementations, they obviously do not have `hpo_defaults` class variables. Instead, every

optimizer has a default optimization strategy stored in `pykeen.optimizers.optimizers_hpo_defaults` the same way that the default strategies for losses are stored externally.

Optimizing the Optimized Optimizer - a.k.a. Learning Rate Schedulers

If optimizing your optimizer doesn't cut it for you, you can turn it up a notch and use learning rate schedulers (lr_scheduler) that will vary the learning rate of the optimizer. This can e.g. be useful to have a more aggressive learning rate in the beginning to quickly make progress while lowering the learning rate over time to allow the model to smoothly converge to the optimum.

PyKEEN allows you to use the learning rate schedulers provided by PyTorch, which you can simply specify as you would in the `pykeen.pipeline.pipeline()`.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='TransE',
...     lr_scheduler='ExponentialLR',
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

The same way as the optimizers don't come with `hpo_defaults` class variables, lr_schedulers rely on their own optimization strategies provided in `pykeen.lr_schedulers.lr_schedulers_hpo_defaults` In case you are ready to explore even more you can of course also set your own ranges with the `lr_scheduler_kwargs_ranges` keyword argument as in:

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     dataset='Nations',
...     model='TransE',
...     lr_scheduler='ExponentialLR',
...     lr_scheduler_kwargs_ranges=dict(
...         gamma=dict(type=float, low=0.8, high=1.0),
...     ),
... )
>>> pipeline_result.save_to_directory('nations_transe')
```

Optimizing Everything Else

Without loss of generality, the following arguments to `pykeen.pipeline.pipeline()` have corresponding **_kwargs* and **_kwargs_ranges*:

- `training_loop` (only kwargs, not kwargs_ranges)
- `evaluator`

- `evaluation`

Early Stopping

Early stopping can be baked directly into the `optuna` optimization.

The important keys are `stopper='early'` and `stopper_kwargs`. When using early stopping, the `hpo_pipeline()` automatically takes care of adding appropriate callbacks to interface with `optuna`.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     dataset='Nations',
...     model='TransE',
...     stopper='early',
...     stopper_kwargs=dict(frequency=5, patience=2, relative_delta=0.002),
... )
```

These stopper kwargs were chosen to make the example run faster. You will likely want to use different ones.

Configuring Optuna

Choosing a Search Algorithm

Because PyKEEN's hyper-parameter optimization pipeline is powered by Optuna, it can directly use all of Optuna's built-in samplers listed on `optuna.samplers` or any custom subclass of `optuna.samplers.BaseSampler`.

By default, PyKEEN uses the Tree-structured Parzen Estimator (TPE; `optuna.samplers.TPESampler`), a probabilistic search algorithm. You can explicitly set the sampler using the `sampler` argument (not to be confused with the negative sampler used when training under the sLCWA):

```
>>> from pykeen.hpo import hpo_pipeline
>>> from optuna.samplers import TPESampler
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     sampler=TPESampler,
...     dataset='Nations',
...     model='TransE',
... )
```

You can alternatively pass a string so you don't have to worry about importing Optuna. PyKEEN knows that sampler classes always end in "Sampler" so you can pass either "TPE" or "TPESampler" as a string. This is case-insensitive.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     sampler="tpe",
...     dataset='Nations',
...     model='TransE',
... )
```

It's also possible to pass a sampler instance directly:

```
>>> from pykeen.hpo import hpo_pipeline
>>> from optuna.samplers import TPESampler
>>> sampler = TPESampler(prior_weight=1.1)
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     sampler=sampler,
...     dataset='Nations',
...     model='TransE',
... )
```

If you're working in a JSON-based configuration setting, you won't be able to instantiate the sampler with your desired settings like this. As a solution, you can pass the keyword arguments via the `sampler_kwargs` argument in combination with specifying the sampler as a string/class to the HPO pipeline like in:

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     sampler="tpe",
...     sampler_kwargs=dict(prior_weight=1.1),
...     dataset='Nations',
...     model='TransE',
... )
```

To emulate most hyper-parameter optimizations that have used random sampling, use `optuna.samplers.RandomSampler` like in:

```
>>> from pykeen.hpo import hpo_pipeline
>>> from optuna.samplers import RandomSampler
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     sampler=RandomSampler,
...     dataset='Nations',
...     model='TransE',
... )
```

Grid search can be performed using `optuna.samplers.GridSampler`. Notice that this sampler expected an additional *search_space* argument in its *sampler_kwargs*, e.g.,

```
>>> from pykeen.hpo import hpo_pipeline
>>> from optuna.samplers import GridSampler
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     sampler=GridSampler,
...     sampler_kwargs=dict(
...         search_space={
...             "model.embedding_dim": [32, 64, 128],
...             "model.scoring_fct_norm": [1, 2],
...             "loss.margin": [1.0],
...             "optimizer.lr": [1.0e-03],
...             "negative_sampler.num_negs_per_pos": [32],
...             "training.num_epochs": [100],
...             "training.batch_size": [128],
...         },
...     ),
...     dataset='Nations',
...     model='TransE',
... )
```

Also notice that the search space of grid search grows fast with increasing number of studied hyper-parameters, and thus grid search is less efficient than other search strategies in finding good configurations, cf. https://jmlr.csail.mit.edu/papers/v13/bergstra12a.html.

Full Examples

The examples above have shown the permutation of one setting at a time. This section has some more complete examples.

The following example sets the optimizer, loss, training, negative sampling, evaluation, and early stopping settings.

```
>>> from pykeen.hpo import hpo_pipeline
>>> hpo_pipeline_result = hpo_pipeline(
...     n_trials=30,
...     dataset='Nations',
...     model='TransE',
...     model_kwargs=dict(embedding_dim=20, scoring_fct_norm=1),
...     optimizer='SGD',
...     optimizer_kwargs=dict(lr=0.01),
...     loss='marginranking',
...     loss_kwargs=dict(margin=1),
...     training_loop='slcwa',
...     training_kwargs=dict(num_epochs=100, batch_size=128),
...     negative_sampler='basic',
...     negative_sampler_kwargs=dict(num_negs_per_pos=1),
...     evaluator_kwargs=dict(filtered=True),
...     evaluation_kwargs=dict(batch_size=128),
...     stopper='early',
...     stopper_kwargs=dict(frequency=5, patience=2, relative_delta=0.002),
... )
```

If you have the configuration as a dictionary:

```
>>> from pykeen.hpo import hpo_pipeline_from_config
>>> config = {
...     'optuna': dict(
...         n_trials=30,
...     ),
...     'pipeline': dict(
...         dataset='Nations',
...         model='TransE',
...         model_kwargs=dict(embedding_dim=20, scoring_fct_norm=1),
...         optimizer='SGD',
...         optimizer_kwargs=dict(lr=0.01),
...         loss='marginranking',
...         loss_kwargs=dict(margin=1),
...         training_loop='slcwa',
...         training_kwargs=dict(num_epochs=100, batch_size=128),
...         negative_sampler='basic',
...         negative_sampler_kwargs=dict(num_negs_per_pos=1),
...         evaluator_kwargs=dict(filtered=True),
...         evaluation_kwargs=dict(batch_size=128),
...         stopper='early',
...         stopper_kwargs=dict(frequency=5, patience=2, relative_delta=0.002),
...     )
... }
... hpo_pipeline_result = hpo_pipeline_from_config(config)
```

If you have a configuration (in the same format) in a JSON file:

```
>>> import json
>>> config = {
...     'optuna': dict(
...         n_trials=30,
...     ),
...     'pipeline': dict(
...         dataset='Nations',
...         model='TransE',
...         model_kwargs=dict(embedding_dim=20, scoring_fct_norm=1),
...         optimizer='SGD',
...         optimizer_kwargs=dict(lr=0.01),
...         loss='marginranking',
...         loss_kwargs=dict(margin=1),
...         training_loop='slcwa',
...         training_kwargs=dict(num_epochs=100, batch_size=128),
...         negative_sampler='basic',
...         negative_sampler_kwargs=dict(num_negs_per_pos=1),
...         evaluator_kwargs=dict(filtered=True),
...         evaluation_kwargs=dict(batch_size=128),
...         stopper='early',
...         stopper_kwargs=dict(frequency=5, patience=2, relative_delta=0.002),
...     )
... }
... with open('config.json', 'w') as file:
...     json.dump(config, file, indent=2)
... hpo_pipeline_result = hpo_pipeline_from_path('config.json')
```

See also

- https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f # noqa:E501