Saving Checkpoints during Training

Training may take days to weeks in extreme cases when using models with many parameters or big datasets. This introduces a large array of possible errors, e.g. session timeouts, server restarts etc., which would lead to a complete loss of all progress made so far. To avoid this PyKEEN supports built-in check-points that allow a straight-forward saving of the current training loop state and resumption of a saved state from saved checkpoints shown in Regular Checkpoints, as well as checkpoints on failure that are only saved when the training loop fails shown in Checkpoints on Failure. For understanding in more detail how the checkpoints work and how they can be used programmatically, please look at Checkpoints beyond the Pipeline and Technicalities. For fixing possible errors and safety fallbacks please also look at Word of Caution and Possible Errors.

Regular Checkpoints

The tutorial First Steps showed how the `pykeen.pipeline.pipeline()` function can be used to set up an entire KGEM for training and evaluation in just two lines of code. A slightly extended example is shown below:

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     optimizer='Adam',
...     training_kwargs=dict(
...         num_epochs=1000,
...     ),
... )
```

To enable checkpoints, all you have to do is add a `checkpoint_name` argument to the `training_kwargs`. This argument should have the name you would like the checkpoint files saved on your computer to be called.

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     optimizer='Adam',
...     training_kwargs=dict(
...         num_epochs=1000,
...         checkpoint_name='my_checkpoint.pt',
...     ),
... )
```

Furthermore, you can set the checkpoint frequency, i.e. how often checkpoints should be saved given in minutes, by setting the argument `checkpoint_frequency` with an integer. The default frequency is 30 minutes and setting it to `0` will cause the training loop to save a checkpoint after each epoch. Let's look at an example.

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     optimizer='Adam',
...     training_kwargs=dict(
...         num_epochs=1000,
...         checkpoint_name='my_checkpoint.pt',
...         checkpoint_frequency=5,
...     ),
... )
```

Here we have defined a pipeline that will save training loop checkpoints in the checkpoint file called `my_checkpoint.pt` every time an epoch finishes and at least *5* minutes have passed since saving previously. Assuming that e.g. this pipeline crashes after 200 epochs, you can simply execute **the same code** and the pipeline will load the last state from the checkpoint file and continue training as if nothing happened. The results will be exactly same as if you ran the pipeline for *1000* epoch without interruption.

Another nice feature is that using checkpoints the training loop will save the state whenever the training loop finishes or the early stopper stops it. Assuming that you successfully trained the KGEM above for *1000* epochs, but now decide that you would like to test the model with *2000* epochs, all you have to do is to change the number of epochs and execute the code like:

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     optimizer='Adam',
...     training_kwargs=dict(
...         num_epochs=2000,  # more epochs than before
...         checkpoint_name='my_checkpoint.pt',
...         checkpoint_frequency=5,
...     ),
... )
```

The above code will load the saved state after finishing *1000* epochs and continue to train to *2000* epochs, giving the exact same results as if you would have run it for *2000* epochs in the first place.

By default, your checkpoints will be saved in the `PYKEEN_HOME` directory that is defined in `pykeen.constants`, which is a subdirectory in your home directory, e.g. `~/.data/pykeen/checkpoints` (configured via `pystow`). Optionally, you can set the path to where you want the checkpoints to be saved by setting the `checkpoint_directory` argument with a string or a `pathlib.Path` object containing your desired root path, as shown in this example:

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     optimizer='Adam',
...     training_kwargs=dict(
...         num_epochs=2000,
...         checkpoint_name='my_checkpoint.pt',
...         checkpoint_directory='doctests/checkpoint_dir',
...     ),
... )
```

Checkpoints on Failure

In cases where you only would like to save checkpoints whenever the training loop might fail, you can use the argument `checkpoint_on_failure=True`, like:

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     optimizer='Adam',
...     training_kwargs=dict(
...         num_epochs=2000,
...         checkpoint_on_failure=True,
...     ),
... )
```

This option differs from regular checkpoints, since regular checkpoints are only saved after a successful epoch. When saving checkpoints due to failure of the training loop there is no guarantee that all random states can be recovered correctly, which might cause problems with regards to the reproducibility of that specific training loop. Therefore, these checkpoints are saved with a distinct checkpoint name, which will be `PyKEEN_just_saved_my_day_{datetime}.pt` in the given `checkpoint_directory`, even when you also opted to use regular checkpoints as defined above, e.g. with this code:

```
>>> from pykeen.pipeline import pipeline
>>> pipeline_result = pipeline(
...     dataset='Nations',
...     model='TransE',
...     optimizer='Adam',
...     training_kwargs=dict(
...         num_epochs=2000,
...         checkpoint_name='my_checkpoint.pt',
...         checkpoint_on_failure=True,
...     ),
... )
```

Note: Use this argument with caution, since every failed training loop will create a distinct checkpoint file.

Checkpoints When Bringing Your Own Data

When continuing the training or in general using the model after resuming training, it is critical that the entity label to identifier (`entity_to_id`) and relation label to identifier (`relation_to_id`) mappings are the same as the ones that were used when saving the checkpoint. If they are not, then any downstream usage will be nonsense.

If you're using a dataset provided by PyKEEN, you're automatically covered. However, when using your own datasets (see Bring Your Own Data), you are responsible for making sure this is the case. Below are two typical examples of combining bringing your own data with checkpoints.

Resuming Training

The following example shows using custom triples factories for the training, validation, and testing datasets derived from files containing labeled triples. Note how the `entity_to_id` and `relation_to_id` arguments are used when creating the `validation` and `testing` triples factories in order to ensure that those datasets are created with the same mappings as the training dataset. Because the `checkpoint_name` is set to `'my_checkpoint.pt'`, PyKEEN saves the checkpoint in `~/.data/pykeen/checkpoints/my_checkpoint.pt`.

```
>>> from pykeen.pipeline import pipeline
>>> from pykeen.triples import TriplesFactory
>>> from pykeen.datasets.nations import NATIONS_TEST_PATH, NATIONS_TRAIN_PATH,
NATIONS_VALIDATE_PATH
>>> training = TriplesFactory.from_path(
...     path=NATIONS_TRAIN_PATH,
... )
>>> validation = TriplesFactory.from_path(
...     path=NATIONS_VALIDATE_PATH,
...     entity_to_id=train.entity_to_id,
...     relation_to_id=train.relation_to_id,
... )
>>> testing = TriplesFactory.from_path(
...     path=NATIONS_TEST_PATH,
...     entity_to_id=train.entity_to_id,
...     relation_to_id=train.relation_to_id,
... )
>>> pipeline_result = pipeline(
...     training=training,
...     validation=validation,
...     testing=testing,
...     model='TransE',
...     optimizer='Adam',
...     training_kwargs=dict(
...         num_epochs=2000,
...         checkpoint_name='my_checkpoint.pt',
...     ),
... )
```

When you are sure that your datasets shown above are the same, you can simply rerun that code and PyKEEN will automatically resume the training where it has left. However, if you only have changed the dataset or you sample it, you need to make sure that the mappings are correct when resuming training from the checkpoint. This can be done by loading the mappings from the checkpoint in the following way.

```
>>> import torch
>>> from pykeen.constants import PYKEEN_CHECKPOINTS
>>> checkpoint = torch.load(PYKEEN_CHECKPOINTS.joinpath('my_checkpoint.pt')
```

You have now loaded the checkpoint that contains the mappings, which now can be used to create mappings that match the model saved in the checkpoint in the following way

```
>>> from pykeen.triples import TriplesFactory
>>> from pykeen.datasets.nations import NATIONS_TEST_PATH, NATIONS_TRAIN_PATH,
NATIONS_VALIDATE_PATH
>>> training = TriplesFactory.from_path(
...     path=NATIONS_TRAIN_PATH,
...     entity_to_id=checkpoint['entity_to_id_dict'],
...     relation_to_id=checkpoint['relation_to_id_dict'],
... )
>>> validation = TriplesFactory.from_path(
...     path=NATIONS_VALIDATE_PATH,
...     entity_to_id=checkpoint['entity_to_id_dict'],
...     relation_to_id=checkpoint['relation_to_id_dict'],
... )
>>> testing = TriplesFactory.from_path(
...     path=NATIONS_TEST_PATH,
...     entity_to_id=checkpoint['entity_to_id_dict'],
...     relation_to_id=checkpoint['relation_to_id_dict'],
... )
```

Now you can simply resume the pipeline with the same code as above:

```
>>> pipeline_result = pipeline(
...     training=training,
...     validation=validation,
...     testing=testing,
...     model='TransE',
...     optimizer='Adam',
...     training_kwargs=dict(
...         num_epochs=2000,
...         checkpoint_name='my_checkpoint.pt',
...     ),
... )
```

In case you feel that this is too much work we still got you covered, since PyKEEN will check in the background whether the provided triples factory mappings match those provided in the checkpoints and will warn you if that is not the case.

Loading Models Manually

Instead of just resuming training with checkpoints as shown above, you can also manually load models from checkpoints for investigation or performing prediction tasks. This can be done in the following way:

```
>>> import torch
>>> from pykeen.constants import PYKEEN_CHECKPOINTS
>>> from pykeen.pipeline import pipeline
>>> from pykeen.triples import TriplesFactory
>>> checkpoint = torch.load(PYKEEN_CHECKPOINTS.joinpath('my_checkpoint.pt'))
```

You have now loaded the checkpoint that contains both the model as well as the `entity_to_id` and `relation_to_id` mapping from the example above. To load these into PyKEEN you just have to do the following:

```
>>> from pykeen.datasets.nations import NATIONS_TRAIN_PATH
>>> train = TriplesFactory.from_path(
...     path=NATIONS_TRAIN_PATH,
...     entity_to_id=checkpoint['entity_to_id_dict'],
...     relation_to_id=checkpoint['relation_to_id_dict'],
... )
```

... now load the model and pass the train triples factory to the model

```
>>> from pykeen.models import TransE
>>> my_model = TransE(triples_factory=train)
>>> my_model.load_state_dict(checkpoint['model_state_dict'])
```

Now you have loaded the model and ensured that the mapping in the triples factory is aligned with the model weights. Enjoy!

Todo

Tutorial on recovery from hpo_pipeline.

Word of Caution and Possible Errors

When using checkpoints and trying out several configurations, which in return result in multiple different checkpoints, the inherent risk of overwriting checkpoints arises. This would naturally happen when you change the configuration of the KGEM, but don't change the `checkpoint_name` argument. To prevent this from happening, PyKEEN makes a hash-sum comparison of the configurations of the checkpoint and the one of the current configuration at hand. When these don't match, PyKEEN won't accept the checkpoint and raise an error.

In case you want to overwrite the previous checkpoint file with a new configuration, you have to delete it explicitly. The reason for this behavior is three-fold:

1. This allows a very easy and user friendly way of resuming an interrupted training loop by simply re-running the exact same code.
2. By explicitly requiring to name the checkpoint files the user controls the naming of the files and thus makes it easier to keep an overview.
3. Creating new checkpoint files implicitly for each run will lead most users to inadvertently spam their file systems with unused checkpoints that with ease can add up to hundred of GBs when running many experiments.

Checkpoints beyond the Pipeline and Technicalities

Currently, PyKEEN only supports checkpoints for training loops, implemented in the class `pykeen.training.TrainingLoop`. When using the `pykeen.pipeline.pipeline()` function as defined above, the pipeline actually uses the training loop functionality. Accordingly, those checkpoints save the states of the training loop and not the pipeline itself. Therefore, the checkpoints won't contain evaluation results that reside in the pipeline. However, PyKEEN makes sure the final results of the pipeline using training loop checkpoints are exactly the same compared to running uninterrupted without checkpoints, also for the evaluation results!

To show how to use the checkpoint functionality without the pipeline, we define a KGEM first:

```
>>> from pykeen.models import TransE
>>> from pykeen.training import SLCWATrainingLoop
>>> from pykeen.triples import TriplesFactory
>>> from torch.optim import Adam
>>> triples_factory = Nations().training
>>> model = TransE(
...     triples_factory=triples_factory,
...     random_seed=123,
... )
>>> optimizer = Adam(params=model.get_grad_params())
>>> training_loop = SLCWATrainingLoop(model=model, optimizer=optimizer)
```

At this point we have a model, dataset and optimizer all setup in a training loop and are ready to train the model with the `training_loop`'s method `pykeen.training.TrainingLoop.train()`. To enable checkpoints all you have to do is setting the function argument `checkpoint_name` to the name you would like it to have. Furthermore, you can set the checkpoint frequency, i.e. how often checkpoints should be saved given in minutes, by setting the argument `checkpoint_frequency` with an integer. The default frequency is 30 minutes and setting it to `0` will cause the training loop to save a checkpoint after each epoch. Optionally, you can set the path to where you want the checkpoints to be saved by setting the `checkpoint_directory` argument with a string or a `pathlib.Path` object containing your desired root path. If you didn't set the `checkpoint_directory` argument, your checkpoints will be saved in the `PYKEEN_HOME` directory that is defined in `pykeen.constants`, which is a subdirectory in your home directory, e.g. `~/.data/pykeen/checkpoints`.

Here is an example:

```
>>> losses = training_loop.train(
...     num_epochs=1000,
...     checkpoint_name='my_checkpoint.pt',
...     checkpoint_frequency=5,
... )
```

With this code we have started the training loop with the above defined KGEM. The training loop will save a checkpoint in the `my_checkpoint.pt` file, which will be saved in the `~/.data/pykeen/checkpoints/` directory, since we haven't set the argument `checkpoint_directory`. The checkpoint file will be saved after 5 minutes since starting the training loop or the last time a checkpoint was saved and the epoch finishes, i.e. when one epoch takes 10 minutes the checkpoint will be saved after 10 minutes. In addition, checkpoints are always saved when the early stopper stops the training loop or the last epoch was finished.

Let's assume you were anticipative, saved checkpoints and your training loop crashed after 200 epochs. Now you would like to resume from the last checkpoint. All you have to do is to rerun the **exact same code** as above and PyKEEN will smoothly start from the given checkpoint. Since PyKEEN stores all random states as well as the states of the model, optimizer and early stopper, the results will be exactly the same compared to running the training loop uninterruptedly. Of course, PyKEEN will also continue saving new checkpoints even when resuming from a previous checkpoint.

On top of resuming interrupted training loops you can also resume training loops that finished successfully. E.g. the above training loop finished successfully after 1000 epochs, but you would like to train the same model from that state for 2000 epochs. All you have have to do is to change the argument `num_epochs` in the above code to:

```
>>> losses = training_loop.train(
...     num_epochs=2000,
...     checkpoint_name='my_checkpoint.pt',
...     checkpoint_frequency=5,
... )
```

and now the training loop will resume from the state at 1000 epochs and continue to train until 2000 epochs.

As shown in Checkpoints on Failure, you can also save checkpoints only in cases where the training loop fails. To do this you just have to set the argument *checkpoint_on_failure=True*, like:

```
>>> losses = training_loop.train(
...     num_epochs=2000,
...     checkpoint_directory='/my/secret/dir',
...     checkpoint_on_failure=True,
... )
```

This code will save a checkpoint in case the training loop fails. Note how we also chose a new checkpoint directory by setting the *checkpoint_directory* argument to `/my/secret/dir`.