collie

Release 1.3.1

ShopRunner Data Science Team

Mar 31, 2023

API REFERENCE

Welcome to the Collie docs!

These docs are meant to be both readable and comprehensive. For the best understanding of the library, we suggest reading the docs in order, starting with *Interactions*.

CHAPTER

ONE

INTERACTIONS

What are Interactions?

The Interactions object is at the core of how data loading and retrieval works in Collie models.

An Interactions object is, in its simplest form, a torch.data.Dataset wrapper around a scipy.sparse.coo_matrix that supports iterating and batching data during model training. We supplement this with data consistency checks during initialization to catch potential errors sooner, a high-throughput and memory-efficient form of negative sampling, and a simple API. Indexing an Interactions object returns a user ID and an item ID that the user has interacted with, as well as an O(1) negative sample of item ID(s) a user has *not* interacted with, supporting the implicit loss functions built into Collie.

```
import pandas as pd

from collie.interactions import Interactions


df = pd.DataFrame(data={'user_id': [0, 0, 0, 1, 1, 2],
                        'item_id': [0, 1, 2, 3, 4, 5]})
interactions = Interactions(users=df['user_id'], items=df['item_id'], num_negative_
↪samples=2)

for _ in range(3):
    print(interactions[0])
```

```
# output structure: ((user IDs, positive item IDs), negative items IDs)
# notice all negative item IDs will be true negatives for user ``0``, e.g.
((0, 0), array([5., 3.]))
((0, 0), array([5., 4.]))
((0, 0), array([3., 5.]))
```

We can see this same idea holds when we instead create an InteractionsDataLoader, as such:

```
import pandas as pd

from collie.interactions import InteractionsDataLoader


df = pd.DataFrame(data={'user_id': [0, 0, 0, 1, 1, 2],
                        'item_id': [0, 1, 2, 3, 4, 5]})
interactions_loader = InteractionsDataLoader(
    users=df['user_id'], items=df['item_id'], num_negative_samples=2
```

(continues on next page)

(continued from previous page)

```
)

for batch in interactions_loader:
    print(batch)
```

```
# output structure: [[user IDs, positive item IDs], negative items IDs]
# users and positive items IDs is now a tensor of shape ``batch_size`` and
# negative items IDs is now a tensor of shape ``batch_size x num_negative_samples``
# notice all negative item IDs will still be true negatives, e.g.
[[tensor([0, 0, 0, 1, 1, 2], dtype=torch.int32),
  tensor([0, 1, 2, 3, 4, 5], dtype=torch.int32)],
 tensor([[4., 5.],
         [3., 5.],
         [4., 5.],
         [0., 1.],
         [5., 0.],
         [3., 4.]])]
```

Once data is in an `Interactions` form, you can easily perform data splits, train and evaluate a model, and much more. See *Cross Validation* and *Models* documentation for more information on this.

How can I speed up Interactions data loading?

While an `Interactions` object works out-of-the-box with a `torch.data.DataLoader`, such as the included `InteractionsDataLoader`, sampling true negatives for each Interactions element can become costly as the number of items grows. In this situation, it might be desirable to *trade exact negative sampling for a faster, approximate sampler*. For these scenarios, we use the `ApproximateNegativeSamplingInteractionsDataLoader`, an extension of the more traditional `InteractionsDataLoader` that samples data in batches, forgoing the expensive concatenation of individual data points an `InteractionsDataLoader` must do for each batch. Here, negative samples are simply returned as a collection of randomly sampled item IDs, meaning it is possible that a negative item ID returned for a user can actually be an item a user had positively interacted with. When the number of items is large, though, this scenario is increasingly rare, and the speedup benefit is worth the slight performance hit.

```
import pandas as pd

from collie.interactions import ApproximateNegativeSamplingInteractionsDataLoader


df = pd.DataFrame(data={'user_id': [0, 0, 0, 1, 1, 2],
                        'item_id': [0, 1, 2, 3, 4, 5]})
interactions_loader = ApproximateNegativeSamplingInteractionsDataLoader(
    users=df['user_id'], items=df['item_id'], num_negative_samples=2
)

for batch in interactions_loader:
    print(batch)
```

```
# output structure: [[user IDs, positive item IDs], "negative" items IDs]
# users and positive items IDs is now a tensor of shape ``batch_size`` and
# negative items IDs is now a tensor of shape ``batch_size x num_negative_samples``
# notice negative item IDs will *not* always be true negatives now, e.g.
[[tensor([0, 0, 0, 1, 1, 2], dtype=torch.int32),
  tensor([0, 1, 2, 3, 4, 5], dtype=torch.int32)],
```

(continues on next page)

(continued from previous page)

```
 tensor([[4, 5],
         [1, 2],
         [4, 2],
         [3, 5],
         [4, 0],
         [4, 3]])]

interactions = interactions_loader.interactions
# use this for cross validation, evaluation, etc.
```

What if my data cannot fit in memory?

For datasets that are too large to fit in memory, Collie includes the `HDF5InteractionsDataLoader` (which uses a `HDF5Interactions` dataset at its base, sharing many of the same features and methods as an `Interactions` object). A `HDF5InteractionsDataLoader` applies the same principles behind the `ApproximateNegativeSamplingInteractionsDataLoader`, but for data stored on disk in a HDF5 format. The main drawback to this approach is that when `shuffle=True`, data will only be shuffled within batches (as opposed to the true shuffle in `ApproximateNegativeSamplingInteractionsDataLoader`). For sufficiently large enough data, this effect on model performance should be negligible.

```
import pandas as pd

from collie.interactions import HDF5InteractionsDataLoader
from collie.utils import pandas_df_to_hdf5


# we'll write out a sample DataFrame to HDF5 format for this example
df = pd.DataFrame(data={'user_id': [0, 0, 0, 1, 1, 2],
                        'item_id': [0, 1, 2, 3, 4, 5]})
pandas_df_to_hdf5(df=df, out_path='sample_hdf5.h5')

interactions_loader = HDF5InteractionsDataLoader(
    hdf5_path='sample_hdf5.h5',
    user_col='user_id',
    item_col='item_id',
    num_negative_samples=2,
)

for batch in interactions_loader:
    print(batch)
```

```
# output structure: [[user IDs, positive item IDs], "negative" items IDs]
# users and positive items IDs is now a tensor of shape ``batch_size`` and
# negative items IDs is now a tensor of shape ``batch_size x num_negative_samples``
# notice negative item IDs will *not* always be true negatives now, e.g.
[[tensor([0, 0, 0, 1, 1, 2], dtype=torch.int32),
  tensor([0, 1, 2, 3, 4, 5], dtype=torch.int32)],
 tensor([[5, 4],
         [4, 5],
         [5, 2],
         [4, 3],
         [4, 2],
         [1, 3]])]
```

The table below shows the time differences to train a `MatrixFactorizationModel` for a single epoch on data using default parameters on the GPU on a `p3.2xlarge` EC2 instance[1].

DataLoader Type	Time to Train a Single Epoch
`InteractionsDataLoader`	1min 25s
`ApproximateNegativeSamplingInteractionsDataLoader`	1min 8s
`HDF5InteractionsDataLoader`	1min 10s

What if my data has explicit ratings in it?

Thus far, we've only discussed the scenario in which you have data *without* an explicit indicator showing to what degree a user loved an item. When you *do* have that data (i.e. star ratings for product reviews, number of times a user has interacted with an item, etc.), you have **explicit data**. Luckily, as of version `0.6.0` of Collie, this is now fully supported within the library, with the only differences between an explicit and implicit pipeline being 1) the dataset definition (detailed below) and 2) evaluation (detailed in *Evaluation Metrics*).

Note the similarities in the explicit example below with the examples shown thus far:

```python
import pandas as pd

from collie.interactions import ExplicitInteractions


explicit_df = pd.DataFrame(data={'user_id': [0, 0, 0, 1, 1, 2],
                                 'item_id': [0, 1, 2, 3, 4, 5],
                                 'ratings': [1, 2, 3, 4, 5, 3.5]})
explicit_interactions = ExplicitInteractions(users=explicit_df['user_id'],
                                             items=explicit_df['item_id'],
                                             ratings=explicit_df['ratings'])

for _ in range(3):
    print(explicit_interactions[0])

print('\n-----\n')
```

(continues on next page)

[1] Welcome to a detailed footnote about this experiment!

- The MovieLens 10M data was preprocessed using Collie utility functions in *Utility Functions* that keeps all ratings above a `4` and removes users with fewer than 3 interactions. This left us with `5,005,398` total interactions.
- For a much faster training time, we recommend setting `sparse=True` (see the point below this) in the model definition and using a larger batch size with `pin_memory=True` in the DataLoader.
- Since we used default parameters, the embeddings of the `MatrixFactorizationModel` were not sparse. Had we used sparse embeddings and a Sparse Adam optimizer, the table would show:

DataLoader Type	Time to Train a Single Epoch
`InteractionsDataLoader`	1min 21s
`ApproximateNegativeSamplingInteractionsDataLoader`	1min 4s
`HDF5InteractionsDataLoader`	1min 7s

  These times are more dramatically different with larger datasets (1M+ items). While these options are certainly faster, having sparse settings be the default limits the optimizer options and general flexibility of customizing an architecture, since not all PyTorch operations support sparse layers. For that reason, we made the default parameters non-sparse, which works best for small-sized datasets.

- We have also noticed drastic changes in training time depending on the version of PyTorch used. While we used `torch@1.8.0` here, we have noticed the fastest training times using `torch@1.6.0`. If you understand why this is, make a PR updating these docs with that information!

(continued from previous page)

```
for idx in range(len(explicit_interactions)):
    print(explicit_interactions[idx])
```

```
# output structure: (user IDs, positive item IDs, ratings)
# notice that unlike implicit interactions, there is no negative sampling going
# on under the hood, meaning this printout will always be deterministic
(0, 0, 1.0)
(0, 0, 1.0)
(0, 0, 1.0)

-----

(0, 0, 1.0)
(0, 1, 2.0)
(0, 2, 3.0)
(1, 3, 4.0)
(1, 4, 5.0)
(2, 5, 3.5)
```

Once the `ExplicitInteractions` dataset is defined, you can use the built-in `InteractionsDataLoader` to batch and iterate through the data!

```
import pandas as pd

from collie.interactions import ExplicitInteractions, InteractionsDataLoader


# the same setup code from the code snippet above
explicit_df = pd.DataFrame(data={'user_id': [0, 0, 0, 1, 1, 2],
                                 'item_id': [0, 1, 2, 3, 4, 5],
                                 'ratings': [1, 2, 3, 4, 5, 3.5]})
explicit_interactions = ExplicitInteractions(users=explicit_df['user_id'],
                                             items=explicit_df['item_id'],
                                             ratings=explicit_df['ratings'])

explicit_interactions_loader = InteractionsDataLoader(interactions=explicit_interactions)

for batch in explicit_interactions_loader:
    print(batch)
```

```
# output structure: [user IDs, positive item IDs, ratings]
[tensor([0, 0, 0, 1, 1, 2], dtype=torch.int32),
 tensor([0, 1, 2, 3, 4, 5], dtype=torch.int32),
 tensor([1.0000, 2.0000, 3.0000, 4.0000, 5.0000, 3.5000], dtype=torch.float64)]
```

All Collie models support both implicit and explicit data, and can be instantiating by either passing in the `Interactions`/`ExplicitInteractions` data or the dataset wrapped in a DataLoader. See *Models* for more details on this.

1.1 Datasets

1.1.1 Implicit Interactions Dataset

class collie.interactions.**Interactions**(*mat: Optional[Union[coo_matrix, array]] = None*, *users: Optional[Iterable[int]] = None*, *items: Optional[Iterable[int]] = None*, *ratings: Optional[Iterable[int]] = None*, *num_negative_samples: int = 10*, *allow_missing_ids: bool = False*, *remove_duplicate_user_item_pairs: bool = True*, *num_users: int = 'infer'*, *num_items: int = 'infer'*, *check_num_negative_samples_is_valid: bool = True*, *max_number_of_samples_to_consider: int = 200*, *seed: Optional[int] = None*)

Bases: `BaseInteractions`

PyTorch `Dataset` for implicit user-item interactions data.

If `mat` is provided, the `Interactions` instance will act as a wrapper for a sparse matrix in COOrdinate format, typically looking like:

- Users comprising the rows
- Items comprising the columns
- Ratings given by that user for that item comprising the elements of the matrix

`Interactions` can be instantiated instead by passing in single arrays with corresponding user_ids, item_ids, and ratings (by default, set to 1 for implicit recommenders) values with the same functionality as a matrix. Note that with this approach, the number of users and items will be the maximum values in those two columns, respectively, and it is expected that all integers between 0 and the maximum ID should appear somewhere in the data.

By default, exact negative sampling will be used during each `__getitem__` call. To use approximate negative sampling, set `max_number_of_samples_to_consider = 0`. This will avoid building a positive item lookup dictionary during initialization.

Unlike in `ExplicitInteractions`, we rely on negative sampling for implicit data. Each `__getitem__` call will thus return a nested tuple containing user IDs, item IDs, and sampled negative item IDs. This nested vs. non-nested structure is key for the model to determine where it should be implicit or explicit. Use the table below for reference:

`__getitem__` Format	Expected Meaning	Model Type
`((X, Y), Z)`	`((user IDs, item IDs), negative item IDs)`	**Implicit**
`(X, Y, Z)`	`(user IDs, item IDs, ratings)`	**Explicit**

Parameters

- **`mat`** (*`scipy.sparse.coo_matrix or numpy.array, 2-dimensional`*) – Interactions matrix, which, if provided, will be used instead of `users`, `items`, and `ratings` arguments
- **`users`** (*`Iterable[int], 1-d`*) – Array of user IDs, starting at 0
- **`items`** (*`Iterable[int], 1-d`*) – Array of corresponding item IDs to `users`, starting at 0
- **`ratings`** (*`Iterable[int], 1-d`*) – Array of corresponding ratings to both `users` and `items`. If `None`, will default to each user in `user` interacting with an item with a rating value of 1

- **num_negative_samples** (*int*) – Number of negative samples to return with each `__getitem__` call
- **allow_missing_ids** (*bool*) – If `False`, will check that both `users` and `items` contain each integer from 0 to the maximum value in the array. This check only applies when initializing an `Interactions` instance using 1-dimensional arrays `users` and `items`
- **remove_duplicate_user_item_pairs** (*bool*) – Will check for and remove any duplicate user, item ID pairs from the `Interactions` matrix during initialization. Note that this will create a second sparse matrix held in memory to efficiently check, which could cause memory concerns for larger data. If you are sure that there are no duplicated, user, item ID pairs, set to `False`
- **num_users** (*int*) – Number of users in the dataset. If `num_users == 'infer'`, this will be set to the `mat.shape[0]` or `max(users) + 1`, depending on the input
- **num_items** (*int*) – Number of items in the dataset. If `num_items == 'infer'`, this will be set to the `mat.shape[1]` or `max(items) + 1`, depending on the input
- **check_num_negative_samples_is_valid** (*bool*) – Check that `num_negative_samples` is less than the maximum number of items a user has interacted with. If it is not, then for all users who have fewer than `num_negative_samples` items not interacted with, a random sample including positive items will be returned as negative
- **max_number_of_samples_to_consider** (*int*) – Number of samples to try for a given user before returning an approximate negative sample. This should be greater than `num_negative_samples`. If set to `0`, approximate negative sampling will be used by default in `__getitem__` and a positive item lookup dictionary will NOT be built
- **seed** (*int*) – Seed for random sampling

head(*n: int = 5*) → array

Return the first `n` rows of the dense matrix as a np.array, 2-d.

tail(*n: int = 5*) → array

Return the last `n` rows of the dense matrix as a np.array, 2-d.

toarray() → array

Transforms `BaseInteractions` instance sparse matrix to np.array, 2-d.

todense() → matrix

Transforms `BaseInteractions` instance sparse matrix to np.matrix, 2-d.

1.1.2 Explicit Interactions Dataset

class collie.interactions.**ExplicitInteractions**(*mat: Optional[Union[coo_matrix, array]] = None*, *users: Optional[Iterable[int]] = None*, *items: Optional[Iterable[int]] = None*, *ratings: Optional[Iterable[int]] = None*, *allow_missing_ids: bool = False*, *remove_duplicate_user_item_pairs: bool = True*, *num_users: int = 'infer'*, *num_items: int = 'infer'*)

Bases: `BaseInteractions`

PyTorch `Dataset` for explicit user-item interactions data.

If `mat` is provided, the `Interactions` instance will act as a wrapper for a sparse matrix in COOrdinate format, typically looking like:

- Users comprising the rows
- Items comprising the columns
- Ratings given by that user for that item comprising the elements of the matrix

`Interactions` can be instantiated instead by passing in single arrays with corresponding user_ids, item_ids, and ratings values with the same functionality as a matrix. Note that with this approach, the number of users and items will be the maximum values in those two columns, respectively, and it is expected that all integers between 0 and the maximum ID should appear somewhere in the user or item ID data.

Unlike in `Interactions`, there is no need for negative sampling for explicit data. Each `__getitem__` call will thus return a single, non-nested tuple containing user IDs, item IDs, and ratings. This nested vs. non-nested structure is key for the model to determine where it should be implicit or explicit. Use the table below for reference:

`__getitem__` Format	Expected Meaning	Model Type
`((X, Y), Z)`	`((user IDs, item IDs), negative item IDs)`	**Implicit**
`(X, Y, Z)`	`(user IDs, item IDs, ratings)`	**Explicit**

Parameters

- **mat** (*scipy.sparse.coo_matrix or numpy.array, 2-dimensional*) – Interactions matrix, which, if provided, will be used instead of `users`, `items`, and `ratings` arguments
- **users** (*Iterable[int], 1-d*) – Array of user IDs, starting at 0
- **items** (*Iterable[int], 1-d*) – Array of corresponding item IDs to `users`, starting at 0
- **ratings** (*Iterable[int], 1-d*) – Array of corresponding ratings to both `users` and `items`. If `None`, will default to each user in `user` interacting with an item with a rating value of 1
- **allow_missing_ids** (*bool*) – If `False`, will check that both `users` and `items` contain each integer from 0 to the maximum value in the array. This check only applies when initializing an `ExplicitInteractions` instance using 1-dimensional arrays `users` and `items`
- **remove_duplicate_user_item_pairs** (*bool*) – Will check for and remove any duplicate user, item ID pairs from the `ExplicitInteractions` matrix during initialization. Note that this will create a second sparse matrix held in memory to efficiently check, which could cause memory concerns for larger data. If you are sure that there are no duplicated, user, item ID pairs, set to `False`
- **num_users** (*int*) – Number of users in the dataset. If `num_users == 'infer'`, this will be set to the `mat.shape[0]` or `max(users) + 1`, depending on the input
- **num_items** (*int*) – Number of items in the dataset. If `num_items == 'infer'`, this will be set to the `mat.shape[1]` or `max(items) + 1`, depending on the input

head(*n: int = 5*) → array

Return the first `n` rows of the dense matrix as a np.array, 2-d.

property num_negative_samples: int

Does not exist for explicit data.

tail(*n: int = 5*) → array

Return the last `n` rows of the dense matrix as a np.array, 2-d.

toarray() → array

Transforms `BaseInteractions` instance sparse matrix to np.array, 2-d.

todense() → matrix

Transforms `BaseInteractions` instance sparse matrix to np.matrix, 2-d.

1.1.3 HDF5 Interactions Dataset

class `collie.interactions.`**HDF5Interactions**(*hdf5_path: str*, *user_col: str = 'users'*, *item_col: str = 'items'*, *num_negative_samples: int = 10*, *num_users: int = 'infer'*, *num_items: int = 'infer'*, *seed: Optional[int] = None*, *shuffle: bool = False*)

Bases: `Dataset`

Create an `Interactions`-like object for data in the HDF5 format that might be too large to fit in memory.

Many of the same features of `Interactions` are implemented here, with the exception that approximate negative sampling will always be used.

Parameters

- **hdf5_path** (*str*) –
- **user_col** (*str*) – Column in HDF5 file with user IDs. IDs must begin at 0
- **item_col** (*str*) – Column in HDF5 file with item IDs. IDs must begin at 0
- **num_negative_samples** (*int*) – Number of negative samples to return with each `__getitem__` call
- **num_users** (*int*) – Number of users in the dataset. If `num_users == 'infer'` and there is not a `meta` key in `hdf5_path`'s HDF5 dataset, this will be set to the the maximum value in `user_col` + 1, found by iterating through the entire dataset
- **num_items** (*int*) – Number of items in the dataset. If `num_items == 'infer'` and there is not an `meta` key in `hdf5_path`'s HDF5 dataset, this will be set to the the maximum value in `item_col` + 1, found by iterating through the entire dataset
- **seed** (*int*) – Seed for random sampling and shuffling if `shuffle is True`
- **shuffle** (*bool*) – Shuffle data in a batch. For example, if one calls `__getitem__` with `start_idx_and_batch_size = (0, 4)` and `shuffle is False`, this will always return the data at indices 0, 1, 2, 3 in order. However, the same call with `shuffle = True` will return a random shuffle of 0, 1, 2, 3 each call. This is recommended for use in a `HDF5InteractionsDataLoader` for training data in lieu of true data shuffling

head(*n: int = 5*) → DataFrame

Return the first `n` rows of the underlying pd.DataFrame.

tail(*n: int = 5*) → DataFrame

Return the last `n` rows of the underlying pd.DataFrame.

1.2 DataLoaders

1.2.1 Interactions DataLoader

`class collie.interactions.`**`InteractionsDataLoader`**(*interactions: Optional[BaseInteractions] = None*, *mat: Optional[Union[coo_matrix, array]] = None*, *users: Optional[Iterable[int]] = None*, *items: Optional[Iterable[int]] = None*, *ratings: Optional[Iterable[int]] = None*, *batch_size: int = 1024*, *shuffle: bool = False*, ***kwargs*)

Bases: `BaseInteractionsDataLoader`

A light wrapper around a `torch.utils.data.DataLoader` for `Interactions`-type datasets.

For implicit data, batches will be created one-point-at-a-time using exact negative sampling (unless configured not to in `interactions`), which is optimal when datasets are smaller (< 1M+ interactions) and model training speed is not a concern. This is the default `DataLoader` for `Interactions` datasets.

For explicit data, negative sampling is not used, but batches will still be created one-point-at-a-time.

Parameters

- **interactions** (*BaseInteractions*) – If not provided, an `Interactions` object will be created with `mat` or all of `users`, `items`, and `ratings`
- **mat** (*scipy.sparse.coo_matrix or numpy.array, 2-dimensional*) – If `interactions is None`, will be used instead of `users`, `items`, and `ratings` arguments to create an `Interactions` object
- **users** (*Iterable[int], 1-d*) – If `interactions is None and mat is None`, array of user IDs, starting at 0
- **items** (*Iterable[int], 1-d*) – If `interactions is None and mat is None`, array of corresponding item IDs to `users`, starting at 0
- **ratings** (*Iterable[int], 1-d*) – If `interactions is None and mat is None`, array of corresponding ratings to both `users` and `items`. If `None`, will default to each user in `user` interacting with an item with a rating value of 1
- **batch_size** (*int*) – Number of samples per batch to load
- **shuffle** (*bool*) – Whether to shuffle the order of data returned or not. This is especially useful for training data to ensure the model does not overfit to a specific order of data
- ****kwargs** (*keyword arguments*) – Relevant keyword arguments will be passed into `Interactions` object creation, if `interactions is None` and the keyword argument matches one of `Interactions.__init__.__code__.co_varnames`. All other keyword arguments will be passed into `torch.utils.data.DataLoader`: https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

`interactions`

Type

Interactions (default) or *ExplicitInteractions*

Original ``torch.utils.data.DataLoader`` docstring as follows

########

```
Data loader. Combines a dataset and a sampler, and provides an iterable over

the given dataset.

The :class:`~torch.utils.data.DataLoader` supports both map-style and

iterable-style datasets with single- or multi-process loading, customizing

loading order and optional automatic batching (collation) and memory pinning.

See :py:mod:`torch.utils.data` documentation page for more details.

Args
```

dataset (Dataset): dataset from which to load the data. batch_size (int, optional): how many samples per batch to load

(default: `1`).

shuffle (bool, optional): set to `True` to have the data reshuffled
at every epoch (default: `False`).

sampler (Sampler or Iterable, optional): defines the strategy to draw
samples from the dataset. Can be any `Iterable` with `__len__` implemented. If specified, `shuffle` must not be specified.

batch_sampler (Sampler or Iterable, optional): like `sampler`, but
returns a batch of indices at a time. Mutually exclusive with `batch_size`, `shuffle`, `sampler`, and `drop_last`.

num_workers (int, optional): how many subprocesses to use for data
loading. `0` means that the data will be loaded in the main process. (default: `0`)

collate_fn (Callable, optional): merges a list of samples to form a
mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.

pin_memory (bool, optional): If `True`, the data loader will copy Tensors
into device/CUDA pinned memory before returning them. If your data elements are a custom type, or your `collate_fn` returns a batch that is a custom type, see the example below.

drop_last (bool, optional): set to `True` to drop the last incomplete batch,
if the dataset size is not divisible by the batch size. If `False` and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: `False`)

timeout (numeric, optional): if positive, the timeout value for collecting a batch
from workers. Should always be non-negative. (default: `0`)

worker_init_fn (Callable, optional): If not `None`, this will be called on each
worker subprocess with the worker id (an int in `[0, num_workers - 1]`) as input, after seeding and before data loading. (default: `None`)

generator (torch.Generator, optional): If not `None`, this RNG will be used
by RandomSampler to generate random indexes and multiprocessing to generate *base_seed* for workers. (default: `None`)

prefetch_factor (int, optional, keyword-only arg): Number of batches loaded
in advance by each worker. `2` means there will be a total of 2 * num_workers batches prefetched across all workers. (default: `2`)

persistent_workers (bool, optional): If `True`, the data loader will not shutdown
the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. (default: `False`)

pin_memory_device (str, optional): the data loader will copy Tensors
into device pinned memory before returning them if pin_memory is set to true.

Warning: If the `spawn` start method is used, `worker_init_fn` cannot be an unpicklable object, e.g., a lambda function. See multiprocessing-best-practices on more details related to multiprocessing in PyTorch.

Warning: `len(dataloader)` heuristic is based on the length of the sampler used. When `dataset` is an `IterableDataset`, it instead returns an estimate based on `len(dataset) / batch_size`, with proper rounding depending on `drop_last`, regardless of multi-process loading configurations. This represents the best guess PyTorch can make because PyTorch trusts user `dataset` code in correctly handling multi-process loading to avoid duplicate data.

However, if sharding results in multiple workers having incomplete last batches, this estimate can still be inaccurate, because (1) an otherwise complete batch can be broken into multiple ones and (2) more than one batch worth of samples can be dropped when `drop_last` is set. Unfortunately, PyTorch can not detect such cases in general.

See **`Dataset Types`_** for more details on these two types of datasets and how `IterableDataset` interacts with **`Multi-process data loading`_**.

Warning: See reproducibility, and dataloader-workers-random-seed, and data-loading-randomness notes for random seed related questions.

`property mat: coo_matrix`

Sparse COO matrix of `interactions`.

`property num_interactions: int`

Number of interactions in `interactions`.

`property num_items: int`

Number of items in `interactions`.

`property num_negative_samples: int`

Number of negative samples in `interactions`.

`property num_users: int`

Number of users in `interactions`.

1.2.2 Approximate Negative Sampling Interactions DataLoader

class collie.interactions.**ApproximateNegativeSamplingInteractionsDataLoader**(*interactions: Optional[Interactions] = None*, *mat: Optional[Union[coo_matrix, array]] = None*, *users: Optional[Iterable[int]] = None*, *items: Optional[Iterable[int]] = None*, *ratings: Optional[Iterable[int]] = None*, *batch_size: int = 1024*, *shuffle: bool = False*, ***kwargs*)

Bases: `BaseInteractionsDataLoader`

A computationally more efficient `DataLoader` for `Interactions` data using approximate negative sampling for negative items.

This DataLoader groups `__getitem__` calls together into a single operation, which dramatically speeds up a traditional DataLoader's process of calling `__getitem__` one index at a time, then concatenating them together before returning. In an effort to batch operations together, all negative samples returned will be approximate, meaning this does not check if a user has previously interacted with the item. With a sufficient number of interactions (1M+), we have found a speed increase of 2x at the cost of a 1% reduction in MAP @ 10 performance compared to `InteractionsDataLoader`.

For greater efficiency, we disable automated batching by setting the DataLoader's `batch_size` attribute to `None`. Thus, to access the "true" batch size that the sampler uses, access `ApproximateNegativeSamplingInteractionsDataLoader.approximate_negative_sampler.batch_size`.

Parameters

- **interactions** (Interactions) – If not provided, an `Interactions` object will be created with `mat` or all of `users`, `items`, and `ratings` with `max_number_of_samples_to_consider=0`
- **mat** (*scipy.sparse.coo_matrix or numpy.array, 2-dimensional*) – If `interactions is None`, will be used instead of `users`, `items`, and `ratings` arguments to create an `Interactions` object
- **users** (*Iterable[int], 1-d*) – If `interactions is None and mat is None`, array of user IDs, starting at 0
- **items** (*Iterable[int], 1-d*) – If `interactions is None and mat is None`, array of corresponding item IDs to `users`, starting at 0
- **ratings** (*Iterable[int], 1-d*) – If `interactions is None and mat is None`, array of corresponding ratings to both `users` and `items`. If `None`, will default to each user in `user` interacting with an item with a rating value of 1
- **batch_size** (*int*) – Number of samples per batch to load
- **shuffle** (*bool*) – Whether to shuffle the order of data returned or not. This is especially useful for training data to ensure the model does not overfit to a specific order of data

- **`**kwargs`** (*`keyword arguments`*) – Relevant keyword arguments will be passed into `Interactions` object creation, if `interactions is None` and the keyword argument matches one of `Interactions.__init__.__code__.co_varnames`. All other keyword arguments will be passed into `torch.utils.data.DataLoader`: https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

`interactions`

Type

Interactions

`Original ``torch.utils.data.DataLoader`` docstring as follows`

`########`

`Data loader. Combines a dataset and a sampler, and provides an iterable over`

`the given dataset.`

``The :class:`~torch.utils.data.DataLoader` supports both map-style and``

`iterable-style datasets with single- or multi-process loading, customizing`

`loading order and optional automatic batching (collation) and memory pinning.`

``See :py:mod:`torch.utils.data` documentation page for more details.``

`Args`

dataset (Dataset): dataset from which to load the data. batch_size (int, optional): how many samples per batch to load

(default: `1`).

shuffle (bool, optional): set to `True` to have the data reshuffled

at every epoch (default: `False`).

sampler (Sampler or Iterable, optional): defines the strategy to draw

samples from the dataset. Can be any `Iterable` with `__len__` implemented. If specified, `shuffle` must not be specified.

batch_sampler (Sampler or Iterable, optional): like `sampler`, but

returns a batch of indices at a time. Mutually exclusive with `batch_size`, `shuffle`, `sampler`, and `drop_last`.

num_workers (int, optional): how many subprocesses to use for data

loading. `0` means that the data will be loaded in the main process. (default: `0`)

collate_fn (Callable, optional): merges a list of samples to form a

mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.

pin_memory (bool, optional): If `True`, the data loader will copy Tensors

into device/CUDA pinned memory before returning them. If your data elements are a custom type, or your `collate_fn` returns a batch that is a custom type, see the example below.

drop_last (bool, optional): set to `True` to drop the last incomplete batch,

if the dataset size is not divisible by the batch size. If `False` and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: `False`)

timeout (numeric, optional): if positive, the timeout value for collecting a batch

from workers. Should always be non-negative. (default: `0`)

worker_init_fn (Callable, optional): If not `None`, this will be called on each
worker subprocess with the worker id (an int in `[0, num_workers - 1]`) as input, after seeding and before data loading. (default: `None`)

generator (torch.Generator, optional): If not `None`, this RNG will be used
by RandomSampler to generate random indexes and multiprocessing to generate *base_seed* for workers. (default: `None`)

prefetch_factor (int, optional, keyword-only arg): Number of batches loaded
in advance by each worker. `2` means there will be a total of 2 * num_workers batches prefetched across all workers. (default: `2`)

persistent_workers (bool, optional): If `True`, the data loader will not shutdown
the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. (default: `False`)

pin_memory_device (str, optional): the data loader will copy Tensors
into device pinned memory before returning them if pin_memory is set to true.

Warning: If the `spawn` start method is used, `worker_init_fn` cannot be an unpicklable object, e.g., a lambda function. See multiprocessing-best-practices on more details related to multiprocessing in PyTorch.

Warning: `len(dataloader)` heuristic is based on the length of the sampler used. When `dataset` is an `IterableDataset`, it instead returns an estimate based on `len(dataset) / batch_size`, with proper rounding depending on `drop_last`, regardless of multi-process loading configurations. This represents the best guess PyTorch can make because PyTorch trusts user `dataset` code in correctly handling multi-process loading to avoid duplicate data.

However, if sharding results in multiple workers having incomplete last batches, this estimate can still be inaccurate, because (1) an otherwise complete batch can be broken into multiple ones and (2) more than one batch worth of samples can be dropped when `drop_last` is set. Unfortunately, PyTorch can not detect such cases in general.

See **`Dataset Types`_** for more details on these two types of datasets and how `IterableDataset` interacts with **`Multi-process data loading`_**.

Warning: See reproducibility, and dataloader-workers-random-seed, and data-loading-randomness notes for random seed related questions.

`property mat: coo_matrix`

Sparse COO matrix of `interactions`.

`property num_interactions: int`

Number of interactions in `interactions`.

`property num_items: int`

Number of items in `interactions`.

`property num_negative_samples: int`

Number of negative samples in `interactions`.

property num_users: int

Number of users in `interactions`.

1.2.3 HDF5 Approximate Negative Sampling Interactions DataLoader

class collie.interactions.**HDF5InteractionsDataLoader**(*hdf5_interactions: Optional[HDF5Interactions] = None, hdf5_path: Optional[str] = None, batch_size: int = 1024, shuffle: bool = False, **kwargs*)

Bases: `BaseInteractionsDataLoader`

A light wrapper around a `torch.utils.data.DataLoader` for HDF5 data, with behavior very similar to `ApproximateNegativeSamplingInteractionsDataLoader`.

If not provided, a `HDF5Interactions` dataset will be created as the data for the `DataLoader`. A custom sampler, `HDF5Sampler`, will also be instantiated for the `DataLoader` to use that allows sampling in batches that make for faster HDF5 data reads from disk.

While similar to a standard `DataLoader`, note that when `shuffle is True`, this will only shuffle the order of batches and the data within batches to still make for efficient reading of HDF5 data from disk, rather than shuffling across the entire dataset.

For greater efficiency, we disable automated batching by setting the DataLoader's `batch_size` attribute to `None`. Thus, to access the "true" batch size that the sampler uses, access `HDF5InteractionsDataLoader.hdf5_sampler.batch_size`.

Parameters

- **hdf5_interactions** (`HDF5Interactions`) – If provided, will override input argument for `hdf5_path`
- **hdf5_path** (*str*) – If `hdf5_interactions is None`, the path to the HDF5 dataset
- **batch_size** (*int*) – Number of samples per batch to load
- **shuffle** (*bool*) – Whether to shuffle the order of batches returned or not. This is especially useful for training data to ensure the model does not overfit to a specific order of data. Note that this will not perform a true shuffle of the data, but shuffle the order of batches. While this is an approximation of true sampling, it allows us a greater speed up during model training for a negligible effect on model performance
- ****kwargs** (*keyword arguments*) – Relevant keyword arguments will be passed into `HDF5Interactions` object creation, if `hdf5_interactions is None` and the keyword argument matches one of `HDF5Interactions.__init__.__code__.co_varnames`. All other keyword arguments will be passed into `torch.utils.data.DataLoader`: https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader
- **follows** (*Original torch.utils.data.DataLoader docstring as*) –
- **########** –
- **sampler** (*Data loader. Combines a dataset and a*) –
- **over** (*and provides an iterable*) –
- **dataset.** (*the given*) –

:param The `DataLoader` supports both map-style and: :param iterable-style datasets with single- or multi-process loading: :param customizing: :param loading order and optional automatic batching (collation) and memory pinning.: :param See `torch.utils.data` documentation page for more details.: :param Args: dataset (Dataset): dataset from which to load the data.

batch_size (int, optional): how many samples per batch to load
(default: `1`).

shuffle (bool, optional): set to `True` to have the data reshuffled
at every epoch (default: `False`).

sampler (Sampler or Iterable, optional): defines the strategy to draw
samples from the dataset. Can be any `Iterable` with `__len__` implemented. If specified, `shuffle` must not be specified.

batch_sampler (Sampler or Iterable, optional): like `sampler`, but
returns a batch of indices at a time. Mutually exclusive with `batch_size`, `shuffle`, `sampler`, and `drop_last`.

num_workers (int, optional): how many subprocesses to use for data
loading. `0` means that the data will be loaded in the main process. (default: `0`)

collate_fn (Callable, optional): merges a list of samples to form a
mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.

pin_memory (bool, optional): If `True`, the data loader will copy Tensors
into device/CUDA pinned memory before returning them. If your data elements are a custom type, or your `collate_fn` returns a batch that is a custom type, see the example below.

drop_last (bool, optional): set to `True` to drop the last incomplete batch,
if the dataset size is not divisible by the batch size. If `False` and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: `False`)

timeout (numeric, optional): if positive, the timeout value for collecting a batch
from workers. Should always be non-negative. (default: `0`)

worker_init_fn (Callable, optional): If not `None`, this will be called on each
worker subprocess with the worker id (an int in `[0, num_workers - 1]`) as input, after seeding and before data loading. (default: `None`)

generator (torch.Generator, optional): If not `None`, this RNG will be used
by RandomSampler to generate random indexes and multiprocessing to generate *base_seed* for workers. (default: `None`)

prefetch_factor (int, optional, keyword-only arg): Number of batches loaded
in advance by each worker. `2` means there will be a total of 2 * num_workers batches prefetched across all workers. (default: `2`)

persistent_workers (bool, optional): If `True`, the data loader will not shutdown
the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. (default: `False`)

pin_memory_device (str, optional): the data loader will copy Tensors
into device pinned memory before returning them if pin_memory is set to true.

Warning: If the `spawn` start method is used, `worker_init_fn` cannot be an unpicklable object, e.g., a lambda function. See multiprocessing-best-practices on more details related to multiprocessing in PyTorch.

Warning: `len(dataloader)` heuristic is based on the length of the sampler used. When `dataset` is an `IterableDataset`, it instead returns an estimate based on `len(dataset) / batch_size`, with proper rounding depending on `drop_last`, regardless of multi-process loading configurations. This represents the best guess PyTorch can make because PyTorch trusts user `dataset` code in correctly handling multi-process loading to avoid duplicate data.

However, if sharding results in multiple workers having incomplete last batches, this estimate can still be inaccurate, because (1) an otherwise complete batch can be broken into multiple ones and (2) more than one batch worth of samples can be dropped when `drop_last` is set. Unfortunately, PyTorch can not detect such cases in general.

See **`Dataset Types`_** for more details on these two types of datasets and how `IterableDataset` interacts with **`Multi-process data loading`_**.

Warning: See reproducibility, and dataloader-workers-random-seed, and data-loading-randomness notes for random seed related questions.

`property mat: None`

`mat` attribute is not possible to access in `HDF5InteractionsDataLoader`.

`property num_interactions: int`

Number of interactions in `interactions`.

`property num_items: int`

Number of items in `interactions`.

`property num_negative_samples: int`

Number of negative samples in `interactions`.

`property num_users: int`

Number of users in `interactions`.

CHAPTER

TWO

CROSS VALIDATION

Once data is set up in an `Interactions` dataset, we can perform a data split to evaluate a trained model later. Collie supports the following two data splits below that share a common API, but differ in split strategy and performance.

```
from collie.cross_validation import random_split, stratified_split
from collie.interactions import Interactions
from collie.movielens import read_movielens_df
from collie.utils import convert_to_implicit, Timer


# EXPERIMENT SETUP
# read in MovieLens 100K data
df = read_movielens_df()

# convert the data to implicit
df_imp = convert_to_implicit(df)

# store data as ``Interactions``
interactions = Interactions(users=df_imp['user_id'],
                            items=df_imp['item_id'],
                            allow_missing_ids=True)

t = Timer()

# EXPERIMENT BEGIN
train, test = random_split(interactions)
t.timecheck(message='Random split timecheck')

train, test = stratified_split(interactions)
t.timecheck(message='Stratified split timecheck')
```

```
# as expected, a random split is much faster than a stratified split
Random split timecheck (0.00 min)
Stratified split timecheck (0.04 min)
```

2.1 Random Split

`collie.cross_validation.`**`random_split`**(*interactions: BaseInteractions*, *val_p: float = 0.0*, *test_p: float = 0.2*, *processes: Optional[Any] = None*, *seed: Optional[int] = None*) → Tuple[BaseInteractions, ...]

Randomly split interactions into training, validation, and testing sets.

This split does NOT guarantee that every user will be represented in both the training and testing datasets. While much faster than `stratified_split`, it is not the most representative data split because of this.

Note that this function is not supported for `HDF5Interactions` objects, since this data split implementation requires all data to fit in memory. A data split for large datasets should be done using a big data processing technology, like Spark.

Parameters

- **interactions** (*collie.interactions.BaseInteractions*) –
- **val_p** (*float*) – Proportion of data used for validation
- **test_p** (*float*) – Proportion of data used for testing
- **processes** (*Any*) – Ignored, included only for compatability with `stratified_split` API
- **seed** (*int*) – Random seed for splits

Returns

- **train_interactions** (*collie.interactions.BaseInteractions*) – Training data of size proportional to `1 - val_p - test_p`
- **validate_interactions** (*collie.interactions.BaseInteractions*) – Validation data of size proportional to `val_p`, returned only if `val_p > 0`
- **test_interactions** (*collie.interactions.BaseInteractions*) – Testing data of size proportional to `test_p`

Examples

```
>>> interactions = Interactions(...)
>>> len(interactions)
100000
>>> train, test = random_split(interactions)
>>> len(train), len(test)
(80000, 20000)
```

2.2 Stratified Split

`collie.cross_validation.`**`stratified_split`**(*interactions: BaseInteractions*, *val_p: float = 0.0*, *test_p: float = 0.2*, *processes: int = -1*, *seed: Optional[int] = None*, *force_split: bool = False*) → Tuple[BaseInteractions, ...]

Split an `Interactions` instance into train, validate, and test datasets in a stratified manner such that each user appears at least once in each of the datasets.

This split guarantees that every user will be represented in the training, validation, and testing datasets given they appear in `interactions` at least three times. If `val_p == 0`, they will appear in the training and testing datasets given they appear at least two times. If a user appears fewer than this number of times, a `ValueError` will be raised. To filter users with fewer than `n` points out, use `collie.utils.remove_users_with_fewer_than_n_interactions`.

This is computationally more complex than `random_split`, but produces a more representative data split. Note that when `val_p > 0`, the algorithm will perform the data split twice, once to create the test set and another to create the validation set, essentially doubling the computational time.

Note that this function is not supported for `HDF5Interactions` objects, since this data split implementation requires all data to fit in memory. A data split for large datasets should be done using a big data processing technology, like Spark.

Parameters

- **`interactions`** (*`collie.interactions.BaseInteractions`*) – `Interactions` instance containing the data to split
- **`val_p`** (*`float`*) – Proportion of data used for validation
- **`test_p`** (*`float`*) – Proportion of data used for testing
- **`processes`** (*`int`*) – Number of CPUs to use for parallelization. If `processes == 0`, this will be run sequentially in a single list comprehension, else this function uses `joblib.delayed` and `joblib.Parallel` for parallelization. A value of `-1` means that all available cores will be used
- **`seed`** (*`int`*) – Random seed for splits
- **`force_split`** (*`bool`*) – Ignore error raised when a user in the dataset has only a single interaction. Normally, a `ValueError` is raised when this occurs. When `force_split=True`, however, users with a single interaction will be placed in the training set and an error will NOT be raised

Returns

- **train_interactions** (*collie.interactions.BaseInteractions*) – Training data of size proportional to `1 - val_p - test_p`
- **validate_interactions** (*collie.interactions.BaseInteractions*) – Validation data of size proportional to `val_p`, returned only if `val_p > 0`
- **test_interactions** (*collie.interactions.BaseInteractions*) – Testing data of size proportional to `test_p`

Examples

```
>>> interactions = Interactions(...)
>>> len(interactions)
100000
>>> train, test = stratified_split(interactions)
>>> len(train), len(test)
(80000, 20000)
```

CHAPTER
THREE

LOSSES

Standard Implicit Loss Functions

A Collie model can’t train without a loss function, and Collie comes out-of-the-box with two different standard loss function calculations: **Bayesian Personalized Ranking (BPR) loss** and **Hinge loss**.

In its simplest form, each loss function accepts as input a prediction score for a positive item (an item a user has interacted with), and a prediction score for a negative item (an item a user has not interacted with). While the mathematical details differ between each loss, generally, **all implicit losses will punish a model for ranking a negative item higher than a positive item**. The severity of this punishment differs across loss functions, as shown in the example below.

```python
import torch

from collie.loss import bpr_loss, hinge_loss


# an ideal loss case
positive_score = torch.tensor([3.0])
negative_score = torch.tensor([1.5])

print('BPR Loss:  ', bpr_loss(positive_score, negative_score))
print('Hinge Loss:', hinge_loss(positive_score, negative_score))

print('\n-----\n')

# a less-than-ideal loss case
positive_score = torch.tensor([1.5])
negative_score = torch.tensor([3.0])

print('BPR Loss:  ', bpr_loss(positive_score, negative_score))
print('Hinge Loss:', hinge_loss(positive_score, negative_score))
```

```
BPR Loss:   tensor(0.2157)
Hinge Loss: tensor(0.)

-----

BPR Loss:   tensor(1.4860)
Hinge Loss: tensor(8.7500)
```

Adaptive Implicit Loss Functions

Some losses extend this idea by being “adaptive,” or accepting multiple negative item prediction scores for each positive score. These losses are typically much more punishing than “non-adaptive” losses, since they allow more opportunities for the model to incorrectly rank a negative item higher than a positive one. These losses include **Adaptive Bayesian Personalized Ranking loss**, **Adaptive Hinge loss**, and **Weighted Approximately Ranked Pairwise (WARP) loss**.

```
import torch

from collie.loss import adaptive_bpr_loss, adaptive_hinge_loss, warp_loss


# an ideal loss case
positive_score = torch.tensor([3.0])
many_negative_scores = torch.tensor([[1.5], [0.5], [1.0]])

print('Adaptive BPR Loss:  ', adaptive_bpr_loss(positive_score, many_negative_scores))
print('Adaptive Hinge Loss:', adaptive_hinge_loss(positive_score, many_negative_scores))
print('WARP Loss:          ', warp_loss(positive_score, many_negative_scores, num_
↪items=3))

print('\n-----\n')

# a less-than-ideal loss case
positive_score = torch.tensor([1.5])
many_negative_scores = torch.tensor([[2.0], [3.0], [2.5]])

print('Adaptive BPR Loss:  ', adaptive_bpr_loss(positive_score, many_negative_scores))
print('Adaptive Hinge Loss:', adaptive_hinge_loss(positive_score, many_negative_scores))
print('WARP Loss:          ', warp_loss(positive_score, many_negative_scores, num_
↪items=3))
print('WARP Loss:          ', warp_loss(positive_score, many_negative_scores, num_
↪items=30))

print('\n-----\n')

# a case where multiple negative items gives us greater opportunity to correct the model
positive_score = torch.tensor([1.5])
many_negative_scores = torch.tensor([[1.0], [4.0], [1.49]])

print('Adaptive BPR Loss:  ', adaptive_bpr_loss(positive_score, many_negative_scores))
print('Adaptive Hinge Loss:', adaptive_hinge_loss(positive_score, many_negative_scores))
print('WARP Loss:          ', warp_loss(positive_score, many_negative_scores, num_
↪items=3))
print('WARP Loss:          ', warp_loss(positive_score, many_negative_scores, num_
↪items=30))
```

```
Adaptive BPR Loss:   tensor(0.2157)
Adaptive Hinge Loss: tensor(0.)
WARP Loss:           tensor(0.)

-----

Adaptive BPR Loss:   tensor(1.4860)
Adaptive Hinge Loss: tensor(8.7500)
WARP Loss:           tensor(4.3636)
```

(continues on next page)

(continued from previous page)

```
WARP Loss:           tensor(31.1301)

-----

Adaptive BPR Loss:   tensor(1.7782)
Adaptive Hinge Loss: tensor(15.7500)
WARP Loss:           tensor(0.8510)
WARP Loss:           tensor(4.5926)
```

Partial Credit Loss Functions

If you have item metadata available, you might reason that not all losses should be equal. For example, say you are training a recommendation system on MovieLens data, where users interact with different films, and you are comparing a positive item, *Star Wars*, with two negative items: *Star Trek* and *Legally Blonde*.

Normally, the loss for *Star Wars* compared with *Star Trek*, and *Star Wars* compared with *Legally Blonde* would be equal. But, as humans, we know that *Star Trek* is closer to *Star Wars* (both being space western films) than *Legally Blonde* is (a romantic comedy that does not have space elements), and would want our loss function to account for that[5].

For these scenarios, all loss functions in Collie support partial credit calculations, meaning we can provide metadata to reduce the potential loss for certain items with matching metadata. This is best seen through an example below:

```python
import torch

# we'll just look at ``bpr_loss`` for this, but note that this works with
# all loss functions in Collie
from collie.loss import bpr_loss


# positive item is Star Wars
star_wars_score = torch.tensor([1.0])

# negative items are Star Trek and Legally Blonde
star_trek_score = torch.tensor([3.0])
legally_blonde_score = torch.tensor([3.0])

print('Star Wars vs Star Trek Loss:      ', end='')
print(bpr_loss(positive_scores=star_wars_score, negative_scores=star_trek_score))

print('Star Wars vs Legally Blonde Loss: ', end='')
print(bpr_loss(positive_scores=star_wars_score, negative_scores=legally_blonde_score))

print('\n-----\n')

# now let's apply a partial credit calculation to the loss
metadata_weights = {'genre': 0.25}

# categorically encode Sci-Fi as ``0`` and Comedy as ``1`` and
# order values by Star Wars, Star Trek, Legally Blonde
metadata = {'genre': torch.tensor([0, 0, 1])}
```

(continues on next page)

[5] If it were up to the author of this library, everyone would be recommended *Legally Blonde*. It is a fantastic film.

(continued from previous page)

```
print('Star Wars vs Star Trek Partial Credit Loss:      ', end='')
print(bpr_loss(positive_scores=star_wars_score,
               negative_scores=star_trek_score,
               positive_items=torch.tensor([0]),
               negative_items=torch.tensor([1]),
               metadata=metadata,
               metadata_weights=metadata_weights))

print('Star Wars vs Legally Blonde Partial Credit Loss: ', end='')
print(bpr_loss(positive_scores=star_wars_score,
               negative_scores=legally_blonde_score,
               positive_items=torch.tensor([0]),
               negative_items=torch.tensor([2]),
               metadata=metadata,
               metadata_weights=metadata_weights))
```

```
Star Wars vs Star Trek Loss:      tensor(1.6566)
Star Wars vs Legally Blonde Loss: tensor(1.6566)

-----

Star Wars vs Star Trek Partial Credit Loss:      tensor(1.0287)
Star Wars vs Legally Blonde Partial Credit Loss: tensor(1.6566)
```

See *Tutorials* for a more in-depth example using partial credit loss functions.

3.1 Standard Losses

3.1.1 BPR Loss

`collie.loss.`**`bpr_loss`**(*positive_scores: tensor*, *negative_scores: tensor*, *num_items: Optional[Any] = None*, *positive_items: Optional[tensor] = None*, *negative_items: Optional[tensor] = None*, *metadata: Optional[Dict[str, tensor]] = {}*, *metadata_weights: Optional[Dict[str, float]] = {}*) → tensor

Modified Bayesian Personalised Ranking[1].

See `ideal_difference_from_metadata` docstring for more info on how metadata is used.

Modified from `torchmf` and `Spotlight`:

- https://github.com/EthanRosenthal/torchmf/blob/master/torchmf.py
- https://github.com/maciejkula/spotlight/blob/master/spotlight/losses.py

Parameters

- **`positive_scores`** (*`torch.tensor, 1-d`*) – Tensor containing predictions for known positive items of shape `1 x batch_size`
- **`negative_scores`** (*`torch.tensor, 1-d`*) – Tensor containing scores for a single sampled negative item of shape `1 x batch_size`

[1] Hildesheim et al. "BPR: Bayesian Personalized Ranking from Implicit Feedback." BPR | Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, 1 June 2009, dl.acm.org/doi/10.5555/1795114.1795167.

- **num_items** (*Any*) – Ignored, included only for compatability with WARP loss
- **positive_items** (*torch.tensor, 1-d*) – Tensor containing ids for known positive items of shape `1 x batch_size`. This is only needed if `metadata` is provided
- **negative_items** (*torch.tensor, 1-d*) – Tensor containing ids for randomly-sampled negative items of shape `1 x batch_size`. This is only needed if `metadata` is provided
- **metadata** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values `<= 1`. e.g. If `metadata_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit

Returns
loss

Return type
torch.tensor

References

3.1.2 Hinge Loss

`collie.loss.`**`hinge_loss`**(*positive_scores: tensor*, *negative_scores: tensor*, *num_items: Optional[Any] = None*, *positive_items: Optional[tensor] = None*, *negative_items: Optional[tensor] = None*, *metadata: Optional[Dict[str, tensor]] = {}*, *metadata_weights: Optional[Dict[str, float]] = {}*) → tensor

Modified hinge pairwise loss function[2].

See `ideal_difference_from_metadata` docstring for more info on how metadata is used.

Modified from `Spotlight`: https://github.com/maciejkula/spotlight/blob/master/spotlight/losses.py

Parameters

- **positive_scores** (*torch.tensor, 1-d*) – Tensor containing scores for known positive items
- **negative_scores** (*torch.tensor, 1-d*) – Tensor containing scores for a single sampled negative item
- **num_items** (*Any*) – Ignored, included only for compatability with WARP loss

[2] "Hinge Loss." Wikipedia, Wikimedia Foundation, 5 Mar. 2021, en.wikipedia.org/wiki/ Hinge_loss.

- **positive_items** (*torch.tensor, 1-d*) – Tensor containing ids for known positive items of shape `1 x batch_size`. This is only needed if `metadata` is provided
- **negative_items** (*torch.tensor, 1-d*) – Tensor containing ids for randomly-sampled negative items of shape `1 x batch_size`. This is only needed if `metadata` is provided
- **metadata** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values <= `1`. e.g. If `metadata_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit

Returns
loss

Return type
torch.tensor

References

3.2 Adaptive Losses

3.2.1 Adaptive BPR Loss

`collie.loss.`**`adaptive_bpr_loss`**(*positive_scores: tensor*, *many_negative_scores: tensor*, *num_items: Optional[Any] = None*, *positive_items: Optional[tensor] = None*, *negative_items: Optional[tensor] = None*, *metadata: Optional[Dict[str, tensor]] = {}*, *metadata_weights: Optional[Dict[str, float]] = {}*) → tensor

Modified adaptive BPR loss function.

Approximates WARP loss by taking the maximum of negative predictions for each user and sending this to BPR loss.

See `ideal_difference_from_metadata` docstring for more info on how metadata is used.

Parameters

- **positive_scores** (*torch.tensor, 1-d*) – Tensor containing scores for known positive items of shape `num_negative_samples x batch_size`
- **many_negative_scores** (*torch.tensor, 2-d*) – Iterable of tensors containing scores for many (n > 1) sampled negative items of shape `num_negative_samples x batch_size`. More tensors increase the likelihood of finding ranking-violating pairs, but risk overfitting

- **num_items** (*Any*) – Ignored, included only for compatability with WARP loss
- **positive_items** (*torch.tensor, 1-d*) – Tensor containing ids for known positive items of shape `num_negative_samples x batch_size`. This is only needed if `metadata` is provided
- **negative_items** (*torch.tensor, 2-d*) – Tensor containing ids for sampled negative items of shape `num_negative_samples x batch_size`. This is only needed if `metadata` is provided
- **metadata** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values `<= 1`. e.g. If `metadata_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit

Returns
loss

Return type
torch.tensor

3.2.2 Adaptive Hinge Loss

`collie.loss.`**`adaptive_hinge_loss`**(*positive_scores: tensor, many_negative_scores: tensor, num_items: Optional[Any] = None, positive_items: Optional[tensor] = None, negative_items: Optional[tensor] = None, metadata: Optional[Dict[str, tensor]] = {}, metadata_weights: Optional[Dict[str, float]] = {}*) → tensor

Modified adaptive hinge pairwise loss function[3].

Approximates WARP loss by taking the maximum of negative predictions for each user and sending this to hinge loss.

See `ideal_difference_from_metadata` docstring for more info on how metadata is used.

Modified from `Spotlight`: https://github.com/maciejkula/spotlight/blob/master/spotlight/losses.py

Parameters

- **positive_scores** (*torch.tensor, 1-d*) – Tensor containing scores for known positive items of shape `num_negative_samples x batch_size`
- **many_negative_scores** (*torch.tensor, 2-d*) – Iterable of tensors containing scores for many (n > 1) sampled negative items of shape `num_negative_samples x`

[3] Kula, Maciej. "Loss Functions." Loss Functions - Spotlight Documentation, maciejkula.github.io/spotlight/losses.html.

batch_size. More tensors increase the likelihood of finding ranking-violating pairs, but risk overfitting

- **num_items** (*Any*) – Ignored, included only for compatability with WARP loss
- **positive_items** (*torch.tensor, 1-d*) – Tensor containing ids for known positive items of shape num_negative_samples x batch_size. This is only needed if metadata is provided
- **negative_items** (*torch.tensor, 2-d*) – Tensor containing ids for sampled negative items of shape num_negative_samples x batch_size. This is only needed if metadata is provided
- **metadata** (*dict*) – Keys should be strings identifying each metadata type that match keys in metadata_weights. Values should be a torch.tensor of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in metadata. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values <= 1. e.g. If metadata_weights = {'genre': .3, 'director': .2}, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit

Returns
loss

Return type
torch.tensor

References

3.2.3 WARP Loss

collie.loss.**warp_loss**(*positive_scores: tensor*, *many_negative_scores: tensor*, *num_items: int*, *positive_items: Optional[tensor] = None*, *negative_items: Optional[tensor] = None*, *metadata: Optional[Dict[str, tensor]] = {}*, *metadata_weights: Optional[Dict[str, float]] = {}*) → tensor

Modified WARP loss function[4].

See http://www.thespermwhale.com/jaseweston/papers/wsabie-ijcai.pdf for loss equation.

See ideal_difference_from_metadata docstring for more info on how metadata is used.

Parameters

- **positive_scores** (*torch.tensor, 1-d*) – Tensor containing scores for known positive items of shape num_negative_samples x batch_size

[4] Weston et al. WSABIE: Scaling Up To Large Vocabulary Image Annotation. www.thespermwhale.com/jaseweston/papers/wsabie-ijcai.pdf.

- **many_negative_scores** (*torch.tensor, 2-d*) – Iterable of tensors containing scores for many (n > 1) sampled negative items of shape `num_negative_samples x batch_size`. More tensors increase the likelihood of finding ranking-violating pairs, but risk overfitting
- **num_items** (*int*) – Total number of items in the dataset
- **positive_items** (*torch.tensor, 1-d*) – Tensor containing ids for known positive items of shape `num_negative_samples x batch_size`. This is only needed if `metadata` is provided
- **negative_items** (*torch.tensor, 2-d*) – Tensor containing ids for sampled negative items of shape `num_negative_samples x batch_size`. This is only needed if `metadata` is provided
- **metadata** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values `<= 1`. e.g. If `metadata_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit

Returns
loss

Return type
torch.tensor

References

CHAPTER

FOUR

MODELS

Instantiating and Training a Collie Model

Collie provides architectures for several state-of-the-art recommendation model architectures for both non-hybrid and hybrid models, depending on if you would like to directly incorporate metadata into the model.

Since Collie utilizes PyTorch Lightning for model training, all models, by default:

- Are compatible with CPU, GPU, multi-GPU, and TPU training
- Allow for 16-bit precision
- Integrate with common external loggers
- Allow for extensive predefined and custom training callbacks
- Are flexible with minimal boilerplate code

While each model's API differs slightly, generally, the training procedure for each model will look like:

```python
from collie.model import CollieTrainer, MatrixFactorizationModel


# assume you have ``interactions`` already defined and ready-to-go

model = MatrixFactorizationModel(interactions)

trainer = CollieTrainer(model)
trainer.fit(model)
model.eval()

# now, ``model`` is ready to be used for inference, evaluation, etc.

model.save_model('model.pkl')
```

When we have side-data about items, this can be incorporated directly into the loss function of the model. For details on this, see *Losses*.

Hybrid Collie models allow incorporating side-data about items and/or users directly into the model. For an in-depth example of this, see *Tutorials*.

Creating a Custom Architecture

Collie not only houses incredible pre-defined architectures, but was built with customization in mind. All Collie recommendation models are built as subclasses of the `BasePipeline` model, inheriting common loss calculation functions and model training boilerplate. This allows for a nice balance between both flexibility and faster iteration.

While any method can be overridden with more architecture-specific implementations, at the bare minimum, each additional model *must* override:

- `_setup_model` - Model architecture initialization
- `forward` - Model step that accepts a batch of data of form `(users, items), negative_items` and outputs a recommendation score for each item

If we wanted to create a custom model that performed a barebones matrix factorization calculation, in Collie, this would be implemented as:

```
import torch

from collie.model import BasePipeline, CollieTrainer, ScaledEmbedding
from collie.utils import get_init_arguments


class SimpleModel(BasePipeline):
    def __init__(self, train, val, embedding_dim):
        """
        Initialize a simple model that is a subclass of ``BasePipeline``.

        Parameters
        ----------
        train: ``collie.interactions`` object
        val: ``collie.interactions`` object
        embedding_dim: int
            Number of latent factors to use for user and item embeddings

        """
        super().__init__(**get_init_arguments())

    def _setup_model(self, **kwargs):
        """Method for building model internals that rely on the data passed in."""
        self.user_embeddings = ScaledEmbedding(num_embeddings=self.hparams.num_users,
                                               embedding_dim=self.hparams.embedding_dim)
        self.item_embeddings = ScaledEmbedding(num_embeddings=self.hparams.num_items,
                                               embedding_dim=self.hparams.embedding_dim)

    def forward(self, users, items):
        """
        Forward pass through the model.

        Parameters
        ----------
        users: tensor, 1-d
            Array of user indices
        items: tensor, 1-d
            Array of item indices

        Returns
        -------
        preds: tensor, 1-d
            Predicted scores

```

(continues on next page)

(continued from previous page)

```
        """
        return torch.mul(
            self.user_embeddings(users), self.item_embeddings(items)
        ).sum(axis=1)


# assume you have ``train`` and ``val`` already defined and ready-to-go

model = SimpleModel(train, val, embedding_dim=10)

trainer = CollieTrainer(model, max_epochs=10)
trainer.fit(model)
model.eval()

# now, ``model`` is ready to be used for inference, evaluation, etc.

model.save_model('model.pkl')
```

See the source code for the `BasePipeline` in *Model Templates* below for the calling order of each class method as well as initialization details for optimizers, schedulers, and more.

4.1 Standard Models

4.1.1 Matrix Factorization Model

class collie.model.**MatrixFactorizationModel**(*train: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.ApproximateNeg ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, val: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.ApproximateNeg ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, embedding_dim: int = 30, dropout_p: float = 0.0, sparse: bool = False, lr: float = 0.001, bias_lr: ~typing.Optional[~typing.Union[float, str]] = 0.01, lr_scheduler_func: ~typing.Optional[~torch.optim.lr_scheduler._LRScheduler] = functools.partial(<class 'torch.optim.lr_scheduler.ReduceLROnPlateau'>, patience=1, verbose=True), weight_decay: float = 0.0, optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', bias_optimizer: ~typing.Optional[~typing.Union[str, ~torch.optim.optimizer.Optimizer]] = 'sgd', loss: ~typing.Union[str, ~typing.Callable[[...], ~torch._VariableFunctionsClass.tensor]] = 'hinge', metadata_for_loss: ~typing.Optional[~typing.Dict[str, ~torch._VariableFunctionsClass.tensor]] = None, metadata_for_loss_weights: ~typing.Optional[~typing.Dict[str, float]] = None, y_range: ~typing.Optional[~typing.Tuple[float, float]] = None, load_model_path: ~typing.Optional[str] = None, map_location: ~typing.Optional[str] = None*)

Bases: `BasePipeline`

Training pipeline for the matrix factorization model.

`MatrixFactorizationModel` models have an embedding layer for both users and items which are dot-producted together to output a single float ranking value.

Collie adds a twist on to this incredibly popular framework by allowing separate optimizers for embeddings and bias terms. With larger datasets and multiple epochs of training, a model might incorrectly learn to only optimize the bias terms for a quicker path towards a local loss minimum, essentially memorizing how popular each item is. By using a separate, slower optimizer for the bias terms (like Stochastic Gradient Descent), the model must prioritize optimizing the embeddings for meaningful, more varied recommendations, leading to a model that is able to achieve a much lower loss. See the documentation below for `bias_lr` and `bias_optimizer` input arguments for implementation details.

All `MatrixFactorizationModel` instances are subclasses of the `LightningModule` class provided by PyTorch Lightning. This means to train a model, you will need a `collie.model.CollieTrainer` object, but the model can be saved and loaded without this `Trainer` instance. Example usage may look like:

```
from collie.model import CollieTrainer, MatrixFactorizationModel

model = MatrixFactorizationModel(train=train)
trainer = CollieTrainer(model)
trainer.fit(model)
model.eval()
```

(continues on next page)

(continued from previous page)

```
# do evaluation as normal with ``model``

model.save_model(filename='model.pth')
new_model = MatrixFactorizationModel(load_model_path='model.pth')

# do evaluation as normal with ``new_model``
```

Parameters

- **train** (collie.interactions object) – Data loader for training data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=True
- **val** (collie.interactions object) – Data loader for validation data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=False
- **embedding_dim** (*int*) – Number of latent factors to use for user and item embeddings
- **dropout_p** (*float*) – Probability of dropout
- **sparse** (*bool*) – Whether or not to treat embeddings as sparse tensors. If True, cannot use weight decay on the optimizer
- **lr** (*float*) – Model learning rate
- **bias_lr** (*float*) – Bias terms learning rate. If 'infer', will set equal to lr
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **optimizer** (*torch.optim or str*) – If a string, one of the following supported optimizers:
  - 'sgd' (for torch.optim.SGD)
  - 'adagrad' (for torch.optim.Adagrad)
  - 'adam' (for torch.optim.Adam)
  - 'sparse_adam' (for torch.optim.SparseAdam)
- **bias_optimizer** (*torch.optim or str*) – Optimizer for the bias terms. This supports the same string options as optimizer, with the addition of infer, which will set the optimizer equal to optimizer. If bias_optimizer is None, only a single optimizer will be created for all model parameters
- **loss** (*function or str*) – If a string, one of the following implemented losses:
  - 'bpr' / 'adaptive_bpr' (implicit data)
  - 'hinge' / 'adaptive_hinge' (implicit data)
  - 'warp' (implicit data)
  - 'mse' (explicit data)
  - 'mae' (explicit data)

For implicit data, if `train.num_negative_samples > 1`, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are `num_items`, `positive_items`, `negative_items`, `metadata`, and `metadata_weights`. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.

- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values <= `1`. e.g. If `metadata_for_loss_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it’s the same item,
  - a 50% match if it’s a different item with the same genre and same director,
  - a 30% match if it’s a different item with the same genre and different director,
  - a 20% match if it’s a different item with a different genre and same director,
  - a 0% match if it’s a different item with a different genre and different director, which is equivalent to the loss without any partial credit
- **y_range** (*tuple*) – Specify as `(min, max)` to apply a sigmoid layer to the output score of the model to get predicted ratings within the range of `min` and `max`
- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of `model.save_model()` method. Note that datasets and optimizers will NOT be restored. If `None`, will initialize model as normal
- **map_location** (*str or torch.device*) – If `load_model_path` is provided, device specifying how to remap storage locations when `torch.load`-ing the state dictionary

forward(*users: tensor*, *items: tensor*) → tensor

Forward pass through the model.

Simple matrix factorization for a single user and item looks like:

``` ``prediction = (user_embedding * item_embedding) + user_bias + item_bias`` ```

If dropout is added, it is applied to the two embeddings and not the biases.

Parameters

- **users** (*tensor, 1-d*) – Array of user indices
- **items** (*tensor, 1-d*) – Array of item indices

Returns

preds – Predicted ratings or rankings

Return type

tensor, 1-d
```

4.1.2 Multilayer Perceptron Matrix Factorization Model

class collie.model.**MLPMatrixFactorizationModel**(train: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.Approximate ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, val: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.Approximate ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, embedding_dim: int = 30, num_layers: int = 3, dropout_p: float = 0.0, lr: float = 0.001, bias_lr: ~typing.Optional[~typing.Union[float, str]] = 0.01, lr_scheduler_func: ~typing.Optional[~torch.optim.lr_scheduler._LRScheduler] = functools.partial(<class 'torch.optim.lr_scheduler.ReduceLROnPlateau'>, patience=1, verbose=True), weight_decay: float = 0.0, optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', bias_optimizer: ~typing.Optional[~typing.Union[str, ~torch.optim.optimizer.Optimizer]] = 'sgd', loss: ~typing.Union[str, ~typing.Callable[[...], ~torch._VariableFunctionsClass.tensor]] = 'hinge', metadata_for_loss: ~typing.Optional[~typing.Dict[str, ~torch._VariableFunctionsClass.tensor]] = None, metadata_for_loss_weights: ~typing.Optional[~typing.Dict[str, float]] = None, y_range: ~typing.Optional[~typing.Tuple[float, float]] = None, load_model_path: ~typing.Optional[str] = None, map_location: ~typing.Optional[str] = None)

Bases: `BasePipeline`

Training pipeline for the matrix factorization model with MLP layers instead of a final dot

product (like in `MatrixFactorizationModel`).

`MLPMatrixFactorizationModel` models have an embedding layer for both users and items which, are concatenated and sent through a MLP to output a single float ranking value.

All `MLPMatrixFactorizationModel` instances are subclasses of the `LightningModule` class provided by PyTorch Lightning. This means to train a model, you will need a `collie.model.CollieTrainer` object, but the model can be saved and loaded without this `Trainer` instance. Example usage may look like:

```python
from collie.model import CollieTrainer, MLPMatrixFactorizationModel

model = MLPMatrixFactorizationModel(train=train)
trainer = CollieTrainer(model)
trainer.fit(model)
model.eval()

# do evaluation as normal with ``model``

model.save_model(filename='model.pth')
new_model = MLPMatrixFactorizationModel(load_model_path='model.pth')
```

(continues on next page)

(continued from previous page)

```
# do evaluation as normal with ``new_model``
```

Parameters

- **train** (collie.interactions object) – Data loader for training data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=True
- **val** (collie.interactions object) – Data loader for validation data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=False
- **embedding_dim** (*int*) – Number of latent factors to use for user and item embeddings
- **num_layers** (*int*) – Number of MLP layers to apply. Each MLP layer will have its input dimension calculated with the formula embedding_dim * (2 ** (``num_layers - current_layer_number))``
- **dropout_p** (*float*) – Probability of dropout on the linear layers
- **lr** (*float*) – Model learning rate
- **bias_lr** (*float*) – Bias terms learning rate. If 'infer', will set equal to lr
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **optimizer** (*torch.optim or str*) – If a string, one of the following supported optimizers:
  - 'sgd' (for torch.optim.SGD)
  - 'adam' (for torch.optim.Adam)
- **bias_optimizer** (*torch.optim or str*) – Optimizer for the bias terms. This supports the same string options as optimizer, with the addition of infer, which will set the optimizer equal to optimizer. If bias_optimizer is None, only a single optimizer will be created for all model parameters
- **loss** (*function or str*) – If a string, one of the following implemented losses:
  - 'bpr' / 'adaptive_bpr' (implicit data)
  - 'hinge' / 'adaptive_hinge' (implicit data)
  - 'warp' (implicit data)
  - 'mse' (explicit data)
  - 'mae' (explicit data)

  For implicit data, if train.num_negative_samples > 1, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

  If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are num_items, positive_items, negative_items, metadata, and metadata_weights. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.

- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values <= `1`. e.g. If `metadata_for_loss_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit
- **y_range** (*tuple*) – Specify as `(min, max)` to apply a sigmoid layer to the output score of the model to get predicted ratings within the range of `min` and `max`
- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of `model.save_model()` method. Note that datasets and optimizers will NOT be restored. If `None`, will initialize model as normal
- **map_location** (*str or torch.device*) – If `load_model_path` is provided, device specifying how to remap storage locations when `torch.load`-ing the state dictionary

forward(*users: tensor*, *items: tensor*) → tensor

Forward pass through the model, roughly:

`` `prediction = MLP(concatenate(user_embedding * item_embedding)) + user_bias + item_bias` ``

If dropout is added, it is applied for the two embeddings and not the biases.

Parameters

- **users** (*tensor, 1-d*) – Array of user indices
- **items** (*tensor, 1-d*) – Array of item indices

Returns
preds – Predicted ratings or rankings

Return type
tensor, 1-d

4.1.3 Nonlinear Embeddings Matrix Factorization Model

class collie.model.**NonlinearMatrixFactorizationModel**(*train: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.Appr ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, val: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.Appr ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, user_embedding_dim: int = 60, item_embedding_dim: int = 60, user_dense_layers_dims: ~typing.List[float] = [48, 32], item_dense_layers_dims: ~typing.List[float] = [48, 32], embedding_dropout_p: float = 0.0, dense_dropout_p: float = 0.0, lr: float = 0.001, bias_lr: ~typing.Optional[~typing.Union[float, str]] = 0.01, lr_scheduler_func: ~typing.Optional[~torch.optim.lr_scheduler._LRScheduler] = functools.partial(<class 'torch.optim.lr_scheduler.ReduceLROnPlateau'>, patience=1, verbose=True), weight_decay: float = 0.0, optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', bias_optimizer: ~typing.Optional[~typing.Union[str, ~torch.optim.optimizer.Optimizer]] = 'sgd', loss: ~typing.Union[str, ~typing.Callable[[...], ~torch._VariableFunctionsClass.tensor]] = 'hinge', metadata_for_loss: ~typing.Optional[~typing.Dict[str, ~torch._VariableFunctionsClass.tensor]] = None, metadata_for_loss_weights: ~typing.Optional[~typing.Dict[str, float]] = None, y_range: ~typing.Optional[~typing.Tuple[float, float]] = None, load_model_path: ~typing.Optional[str] = None, map_location: ~typing.Optional[str] = None*)

Bases: `BasePipeline`

Training pipeline for a nonlinear matrix factorization model.

`NonlinearMatrixFactorizationModel` models have an embedding layer for users and items. These are sent through separate dense networks, which output more refined embeddings, which are then dot producted for a single float ranking / rating.

Collie adds a twist on to this novel framework by allowing separate optimizers for embeddings and bias terms. With larger datasets and multiple epochs of training, a model might incorrectly learn to only optimize the bias terms for a quicker path towards a local loss minimum, essentially memorizing how popular each item is. By using a separate, slower optimizer for the bias terms (like Stochastic Gradient Descent), the model must prioritize optimizing the embeddings for meaningful, more varied recommendations, leading to a model that is able to achieve a much lower loss. See the documentation below for `bias_lr` and `bias_optimizer` input arguments for implementation details.

All NonlinearMatrixFactorizationModel instances are subclasses of the LightningModule class provided by PyTorch Lightning. This means to train a model, you will need a collie.model.CollieTrainer object, but the model can be saved and loaded without this Trainer instance. Example usage may look like:

```
from collie.model import CollieTrainer, NonlinearMatrixFactorizationModel

model = NonlinearMatrixFactorizationModel(train=train)
trainer = CollieTrainer(model)
trainer.fit(model)
model.eval()

# do evaluation as normal with ``model``

model.save_model(filename='model.pth')
new_model = NonlinearMatrixFactorizationModel(load_model_path='model.pth')

# do evaluation as normal with ``new_model``
```

Parameters

- **train** (collie.interactions object) – Data loader for training data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=True
- **val** (collie.interactions object) – Data loader for validation data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=False
- **user_embedding_dim** (*int*) – Number of latent factors to use for user embeddings
- **item_embedding_dim** (*int*) – Number of latent factors to use for item embeddings
- **user_dense_layers_dims** (*list*) – List of linear layer dimensions to apply to the user embedding, starting with the dimension directly following user_embedding_dim
- **item_dense_layers_dims** (*list*) – List of linear layer dimensions to apply to the item embedding, starting with the dimension directly following item_embedding_dim
- **embedding_dropout_p** (*float*) – Probability of dropout on the embedding layers
- **dense_dropout_p** (*float*) – Probability of dropout on the dense layers
- **lr** (*float*) – Model learning rate
- **bias_lr** (*float*) – Bias terms learning rate. If 'infer', will set equal to lr
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **optimizer** (*torch.optim or str*) – If a string, one of the following supported optimizers:
  - 'sgd' (for torch.optim.SGD)
  - 'adam' (for torch.optim.Adam)
- **bias_optimizer** (*torch.optim or str*) – Optimizer for the bias terms. This supports the same string options as optimizer, with the addition of infer, which will set the optimizer equal to optimizer. If bias_optimizer is None, only a single optimizer will be created for all model parameters

- **loss** (*function or str*) – If a string, one of the following implemented losses:
  - `'bpr'` / `'adaptive_bpr'` (implicit data)
  - `'hinge'` / `'adaptive_hinge'` (implicit data)
  - `'warp'` (implicit data)
  - `'mse'` (explicit data)
  - `'mae'` (explicit data)

  For implicit data, if `train.num_negative_samples > 1`, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

  If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are `num_items`, `positive_items`, `negative_items`, `metadata`, and `metadata_weights`. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.

- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)

- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values `<= 1`. e.g. If `metadata_for_loss_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit

- **y_range** (*tuple*) – Specify as `(min, max)` to apply a sigmoid layer to the output score of the model to get predicted ratings within the range of `min` and `max`

- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of `model.save_model()` method. Note that datasets and optimizers will NOT be restored. If `None`, will initialize model as normal

- **map_location** (*str or torch.device*) – If `load_model_path` is provided, device specifying how to remap storage locations when `torch.load`-ing the state dictionary

forward(*users: tensor*, *items: tensor*) → tensor

Forward pass through the model.

Parameters

- **users** (*tensor, 1-d*) – Array of user indices
- **items** (*tensor, 1-d*) – Array of item indices

Returns
preds – Predicted ratings or rankings

Return type
tensor, 1-d

4.1.4 Collaborative Metric Learning Model

class collie.model.**CollaborativeMetricLearningModel**(*train: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.Appro ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, val: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.Appro ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, embedding_dim: int = 30, sparse: bool = False, lr: float = 0.001, lr_scheduler_func: ~typing.Optional[~torch.optim.lr_scheduler._LRScheduler] = functools.partial(<class 'torch.optim.lr_scheduler.ReduceLROnPlateau'>, patience=1, verbose=True), weight_decay: float = 0.0, optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', loss: ~typing.Union[str, ~typing.Callable[[...], ~torch._VariableFunctionsClass.tensor]] = 'hinge', metadata_for_loss: ~typing.Optional[~typing.Dict[str, ~torch._VariableFunctionsClass.tensor]] = None, metadata_for_loss_weights: ~typing.Optional[~typing.Dict[str, float]] = None, y_range: ~typing.Optional[~typing.Tuple[float, float]] = None, load_model_path: ~typing.Optional[str] = None, map_location: ~typing.Optional[str] = None*)

Bases: `BasePipeline`

Training pipeline for the collaborative metric learning model.

`CollaborativeMetricLearningModel` models have an embedding layer for both users and items. A single float, prediction is retrieved by taking the pairwise distance between the two embeddings.

The implementation here is meant to mimic its original implementation as specified here: https://arxiv.org/pdf/1803.00202.pdf[1]

All `CollaborativeMetricLearningModel` instances are subclasses of the `LightningModule` class provided by PyTorch Lightning. This means to train a model, you will need a `collie.model.CollieTrainer` object, but the model can be saved and loaded without this `Trainer` instance. Example usage may look like:

[1] Campo, Miguel, et al. "Collaborative Metric Learning Recommendation System: Application to Theatrical Movie Releases." ArXiv.org, 1 Mar. 2018, arxiv.org/abs/1803.00202.

```
from collie.model import CollaborativeMetricLearningModel, CollieTrainer

model = CollaborativeMetricLearningModel(train=train)
trainer = CollieTrainer(model)
trainer.fit(model)
model.eval()

# do evaluation as normal with ``model``

model.save_model(filename='model.pth')
new_model = CollaborativeMetricLearningModel(load_model_path='model.pth')

# do evaluation as normal with ``new_model``
```

Parameters

- **train** (collie.interactions object) – Data loader for training data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=True
- **val** (collie.interactions object) – Data loader for validation data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=False
- **embedding_dim** (*int*) – Number of latent factors to use for user and item embeddings
- **sparse** (*bool*) – Whether or not to treat embeddings as sparse tensors. If True, cannot use weight decay on the optimizer
- **lr** (*float*) – Model learning rate
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **optimizer** (*torch.optim or str*) – If a string, one of the following supported optimizers:
  - 'sgd' (for torch.optim.SGD)
  - 'adagrad' (for torch.optim.Adagrad)
  - 'adam' (for torch.optim.Adam)
  - 'sparse_adam' (for torch.optim.SparseAdam)
- **loss** (*function or str*) – If a string, one of the following implemented losses:
  - 'bpr' / 'adaptive_bpr' (implicit data)
  - 'hinge' / 'adaptive_hinge' (implicit data)
  - 'warp' (implicit data)
  - 'mse' (explicit data)
  - 'mae' (explicit data)

  For implicit data, if train.num_negative_samples > 1, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are `num_items`, `positive_items`, `negative_items`, `metadata`, and `metadata_weights`. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.

- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values <= 1. e.g. If `metadata_for_loss_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit
- **y_range** (*tuple*) – Specify as `(min, max)` to apply a sigmoid layer to the output score of the model to get predicted ratings within the range of `min` and `max`
- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of `model.save_model()` method. Note that datasets and optimizers will NOT be restored. If `None`, will initialize model as normal
- **map_location** (*str or torch.device*) – If `load_model_path` is provided, device specifying how to remap storage locations when `torch.load`-ing the state dictionary

References

forward(*users: tensor*, *items: tensor*) → tensor

Forward pass through the model, equivalent to:

`` `prediction = pairwise_distance(user_embedding * item_embedding)` ``

Parameters

- **users** (*tensor, 1-d*) – Array of user indices
- **items** (*tensor, 1-d*) – Array of item indices

Returns

preds – Predicted ratings or rankings

Return type

tensor, 1-d

4.1.5 Neural Collaborative Filtering (NeuCF)

class collie.model.**NeuralCollaborativeFiltering**(*train: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.Approxima ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, val: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.Approxima ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, embedding_dim: int = 8, num_layers: int = 3, final_layer: ~typing.Optional[~typing.Union[str, ~typing.Callable[[~torch._VariableFunctionsClass.tensor], ~torch._VariableFunctionsClass.tensor]]] = None, dropout_p: float = 0.0, lr: float = 0.001, lr_scheduler_func: ~typing.Optional[~torch.optim.lr_scheduler._LRScheduler] = functools.partial(<class 'torch.optim.lr_scheduler.ReduceLROnPlateau'>, patience=1, verbose=True), weight_decay: float = 0.0, optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', loss: ~typing.Union[str, ~typing.Callable[[...], ~torch._VariableFunctionsClass.tensor]] = 'hinge', metadata_for_loss: ~typing.Optional[~typing.Dict[str, ~torch._VariableFunctionsClass.tensor]] = None, metadata_for_loss_weights: ~typing.Optional[~typing.Dict[str, float]] = None, load_model_path: ~typing.Optional[str] = None, map_location: ~typing.Optional[str] = None*)

Bases: `BasePipeline`

Training pipeline for a neural matrix factorization model.

`NeuralCollaborativeFiltering` models combine a collaborative filtering and multilayer perceptron network in a single, unified model. The model consists of two sections: the first is a simple matrix factorization that calculates a score by multiplying together user and item embeddings (lookups through an embedding table); the second is a MLP network that feeds embeddings from a second set of embedding tables (one for user, one for item). Both output vectors are combined and sent through a final MLP layer before returning a single recommendation score.

The implementation here is meant to mimic its original implementation as specified here: https://arxiv.org/pdf/1708.05031.pdf[2]

All `NeuralCollaborativeFiltering` instances are subclasses of the `LightningModule` class provided by PyTorch Lightning. This means to train a model, you will need a `collie.model.CollieTrainer` object, but the model can be saved and loaded without this `Trainer` instance. Example usage may look like:

```
from collie.model import CollieTrainer, NeuralCollaborativeFiltering

model = NeuralCollaborativeFiltering(train=train)
trainer = CollieTrainer(model)
```

(continues on next page)

[2] Xiangnan et al. "Neural Collaborative Filtering." Neural Collaborative Filtering | Proceedings of the 26th International Conference on World Wide Web, 1 Apr. 2017, dl.acm.org/doi/10.1145/3038912.3052569.

(continued from previous page)

```
trainer.fit(model)
model.eval()

# do evaluation as normal with ``model``

model.save_model(filename='model.pth')
new_model = NeuralCollaborativeFiltering(load_model_path='model.pth')

# do evaluation as normal with ``new_model``
```

Parameters

- **train** (collie.interactions object) – Data loader for training data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=True
- **val** (collie.interactions object) – Data loader for validation data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=False
- **embedding_dim** (*int*) – Number of latent factors to use for the matrix factorization embedding table. For the MLP embedding table, the dimensionality will be calculated with the formula embedding_dim * (2 ** (num_layers - 1))
- **num_layers** (*int*) – Number of MLP layers to apply. Each MLP layer will have its input dimension calculated with the formula embedding_dim * (2 ** (``num_layers - current_layer_number))``
- **final_layer** (*str or function*) – Final layer activation function. Available string options include:
  - ’sigmoid’
  - ’relu’
  - ’leaky_relu’
- **dropout_p** (*float*) – Probability of dropout on the MLP layers
- **lr** (*float*) – Model learning rate
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **optimizer** (*torch.optim or str*) – If a string, one of the following supported optimizers:
  - 'sgd' (for torch.optim.SGD)
  - 'adam' (for torch.optim.Adam)
- **loss** (*function or str*) – If a string, one of the following implemented losses:
  - 'bpr' / 'adaptive_bpr' (implicit data)
  - 'hinge' / 'adaptive_hinge' (implicit data)
  - 'warp' (implicit data)
  - 'mse' (explicit data)

  - `'mae'` (explicit data)

  For implicit data, if `train.num_negative_samples > 1`, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

  If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are `num_items`, `positive_items`, `negative_items`, `metadata`, and `metadata_weights`. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.

- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values <= 1. e.g. If `metadata_for_loss_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit
- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of `model.save_model()` method. Note that datasets and optimizers will NOT be restored. If `None`, will initialize model as normal
- **map_location** (*str or torch.device*) – If `load_model_path` is provided, device specifying how to remap storage locations when `torch.load`-ing the state dictionary

References

`forward`(*users: tensor*, *items: tensor*) → tensor

Forward pass through the model.

Parameters

- **users** (*tensor, 1-d*) – Array of user indices
- **items** (*tensor, 1-d*) – Array of item indices

Returns

preds – Predicted ratings or rankings

Return type

tensor, 1-d

4.1.6 Deep Factorization Machine (DeepFM)

class collie.model.**DeepFM**(train: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.ApproximateNegativeSamplingInteractio
~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, val: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.ApproximateNegativeSamplingInteractio
~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, embedding_dim: int = 8, num_layers: int = 3, final_layer: ~typing.Optional[~typing.Union[str, ~typing.Callable[[...], ~typing.Any]]] = None, dropout_p: float = 0.0, lr: float = 0.001, bias_lr: ~typing.Optional[~typing.Union[float, str]] = 0.01, lr_scheduler_func: ~typing.Optional[~torch.optim.lr_scheduler._LRScheduler] = functools.partial(<class 'torch.optim.lr_scheduler.ReduceLROnPlateau'>, patience=1, verbose=True), weight_decay: float = 0.0, optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', bias_optimizer: ~typing.Optional[~typing.Union[str, ~torch.optim.optimizer.Optimizer]] = 'sgd', loss: ~typing.Union[str, ~typing.Callable[[...], ~torch._VariableFunctionsClass.tensor]] = 'hinge', metadata_for_loss: ~typing.Optional[~typing.Dict[str, ~torch._VariableFunctionsClass.tensor]] = None, metadata_for_loss_weights: ~typing.Optional[~typing.Dict[str, float]] = None, load_model_path: ~typing.Optional[str] = None, map_location: ~typing.Optional[str] = None)

Bases: `BasePipeline`

Training pipeline for a deep factorization model.

`DeepFM` models combine a shallow factorization machine and a deep multilayer perceptron network in a single, unified model. The model consists of embedding tables for users and items, and model output is the sum of 1) factorization machine output of both embeddings (shallow) and 2) MLP output for the concatenation of both embeddings (deep).

The implementation here is meant to mimic its original implementation as specified here: https://arxiv.org/pdf/1703.04247.pdf[3]

All `DeepFM` instances are subclasses of the `LightningModule` class provided by PyTorch Lightning. This means to train a model, you will need a `collie.model.CollieTrainer` object, but the model can be saved and loaded without this `Trainer` instance. Example usage may look like:

```python
from collie.model import CollieTrainer, DeepFM

model = DeepFM(train=train)
trainer = CollieTrainer(model)
trainer.fit(model)
model.eval()

# do evaluation as normal with ``model``

model.save_model(filename='model.pth')
new_model = DeepFM(load_model_path='model.pth')

# do evaluation as normal with ``new_model``
```

[3] Guo, Huifeng, et al. "DeepFM: A Factorization-Machine Based Neural Network for CTR Prediction." ArXiv.org, 13 Mar. 2017, arxiv.org/abs/1703.04247.

Parameters

- **train** (collie.interactions object) – Data loader for training data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=True
- **val** (collie.interactions object) – Data loader for validation data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=False
- **embedding_dim** (*int*) – Number of latent factors to use for the matrix factorization embedding table. For the MLP embedding table, the dimensionality will be calculated with the formula embedding_dim * (2 ** (num_layers - 1))
- **num_layers** (*int*) – Number of MLP layers to apply. Each MLP layer will have its input dimension calculated with the formula embedding_dim * (2 ** (``num_layers - current_layer_number))``
- **final_layer** (*str or function*) – Final layer activation function. Available string options include:
  - ’sigmoid’
  - ’relu’
  - ’leaky_relu’
- **dropout_p** (*float*) – Probability of dropout
- **lr** (*float*) – Model learning rate
- **bias_lr** (*float*) – Bias terms learning rate. If ‘infer’, will set equal to lr
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **optimizer** (*torch.optim or str*) – If a string, one of the following supported optimizers:
  - 'sgd' (for torch.optim.SGD)
  - 'adam' (for torch.optim.Adam)
- **bias_optimizer** (*torch.optim or str*) – Optimizer for the bias terms. This supports the same string options as optimizer, with the addition of infer, which will set the optimizer equal to optimizer. If bias_optimizer is None, only a single optimizer will be created for all model parameters
- **loss** (*function or str*) – If a string, one of the following implemented losses:
  - 'bpr' / 'adaptive_bpr' (implicit data)
  - 'hinge' / 'adaptive_hinge' (implicit data)
  - 'warp' (implicit data)
  - 'mse' (explicit data)
  - 'mae' (explicit data)

  For implicit data, if train.num_negative_samples > 1, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are `num_items`, `positive_items`, `negative_items`, `metadata`, and `metadata_weights`. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.

- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values <= 1. e.g. If `metadata_for_loss_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit
- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of `model.save_model()` method. Note that datasets and optimizers will NOT be restored. If `None`, will initialize model as normal
- **map_location** (*str or torch.device*) – If `load_model_path` is provided, device specifying how to remap storage locations when `torch.load`-ing the state dictionary

References

forward(*users: tensor*, *items: tensor*) → tensor

Forward pass through the model.

Parameters

- **users** (*tensor, 1-d*) – Array of user indices
- **items** (*tensor, 1-d*) – Array of item indices

Returns

preds – Predicted ratings or rankings

Return type

tensor, 1-d

4.2 Hybrid Models

4.2.1 Hybrid Pretrained Matrix Factorization Model

class collie.model.**HybridPretrainedModel**(train: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.ApproximateNegativ
~collie.interactions.datasets.Interactions,
~collie.interactions.dataloaders.InteractionsDataLoader]] =
None, val: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.ApproximateNegativ
~collie.interactions.datasets.Interactions,
~collie.interactions.dataloaders.InteractionsDataLoader]] =
None, item_metadata: ~typing.Optional[~typing.Union[~torch._VariableFunctionsClass.tensor,
~pandas.core.frame.DataFrame, ~numpy.array]] = None,
user_metadata: ~typing.Optional[~typing.Union[~torch._VariableFunctionsClass.tensor,
~pandas.core.frame.DataFrame, ~numpy.array]] = None,
trained_model: ~typing.Optional[~collie.model.matrix_factorization.MatrixFactorizationModel]
= None, item_metadata_layers_dims:
~typing.Optional[~typing.List[int]] = None,
user_metadata_layers_dims:
~typing.Optional[~typing.List[int]] = None,
combined_layers_dims: ~typing.List[int] = [128, 64, 32],
freeze_embeddings: bool = True, dropout_p: float = 0.0, lr:
float = 0.001, lr_scheduler_func:
~typing.Optional[~torch.optim.lr_scheduler._LRScheduler] =
functools.partial(<class
'torch.optim.lr_scheduler.ReduceLROnPlateau'>, patience=1,
verbose=True), weight_decay: float = 0.0, optimizer:
~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam',
loss: ~typing.Union[str, ~typing.Callable[[...],
~torch._VariableFunctionsClass.tensor]] = 'hinge',
metadata_for_loss: ~typing.Optional[~typing.Dict[str,
~torch._VariableFunctionsClass.tensor]] = None,
metadata_for_loss_weights: ~typing.Optional[~typing.Dict[str,
float]] = None, load_model_path: ~typing.Optional[str] =
None, map_location: ~typing.Optional[str] = None)

Bases: `BasePipeline`

Training pipeline for a hybrid recommendation model using a pre-trained matrix factorization

model as its base.

`HybridPretrainedModel` models contain dense layers that process item and/or user metadata, concatenate this embedding with the user and item embeddings copied from a trained `MatrixFactorizationModel`, and send this concatenated embedding through more dense layers to output a single float ranking / rating. We add both user and item biases to this score before returning. This is the same architecture as the `HybridModel`, but we are using the embeddings from a pre-trained model rather than training them up ourselves.

All `HybridPretrainedModel` instances are subclasses of the `LightningModule` class provided by PyTorch Lightning. This means to train a model, you will need a `collie.model.CollieTrainer` object, but the model can be saved and loaded without this `Trainer` instance. Example usage may look like:

```
from collie.model import CollieTrainer, HybridPretrainedModel, MatrixFactorizationModel

# instantiate and fit a ``MatrixFactorizationModel`` as expected
mf_model = MatrixFactorizationModel(train=train)
mf_trainer = CollieTrainer(mf_model)
mf_trainer.fit(mf_model)

hybrid_model = HybridPretrainedModel(train=train,
                                     item_metadata=item_metadata,
                                     user_metadata=user_metadata,
                                     trained_model=mf_model)
hybrid_trainer = CollieTrainer(hybrid_model)
hybrid_trainer.fit(hybrid_model)
hybrid_model.eval()

# do evaluation as normal with ``hybrid_model``

hybrid_model.save_model(path='model')
new_hybrid_model = HybridPretrainedModel(load_model_path='model')

# do evaluation as normal with ``new_hybrid_model``
```

Parameters

- **train** (collie.interactions object) – Data loader for training data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=True
- **val** (collie.interactions object) – Data loader for validation data. If an Interactions object is supplied, an InteractionsDataLoader will automatically be instantiated with shuffle=False
- **item_metadata** (*torch.tensor, pd.DataFrame, or np.array, 2-dimensional*) – The shape of the item metadata should be (num_items x metadata_features), and each item's metadata should be available when indexing a row by an item ID
- **user_metadata** (*torch.tensor, pd.DataFrame, or np.array, 2-dimensional*) – The shape of the user metadata should be (num_users x metadata_features), and each user's metadata should be available when indexing a row by an item ID
- **trained_model** (collie.model.MatrixFactorizationModel) – Previously trained MatrixFactorizationModel model to extract embeddings from
- **item_metadata_layers_dims** (*list*) – List of linear layer dimensions to apply to the item metadata only, starting with the dimension directly following metadata_features and ending with the dimension to concatenate with the item embeddings
- **user_metadata_layers_dims** (*list*) – List of linear layer dimensions to apply to the user metadata only, starting with the dimension directly following metadata_features and ending with the dimension to concatenate with the user embeddings
- **combined_layers_dims** (*list*) – List of linear layer dimensions to apply to the concatenated item embeddings and item metadata, starting with the dimension directly following the shape of item_embeddings + metadata_features and ending with the dimension before the final linear layer to dimension 1

- **freeze_embeddings** (*bool*) – When initializing the model, whether or not to freeze `trained_model`'s embeddings
- **dropout_p** (*float*) – Probability of dropout
- **lr** (*float*) – Model learning rate
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **optimizer** (*torch.optim or str*) – If a string, one of the following supported optimizers:
  - `'sgd'` (for `torch.optim.SGD`)
  - `'adam'` (for `torch.optim.Adam`)
- **loss** (*function or str*) – If a string, one of the following implemented losses:
  - `'bpr'` / `'adaptive_bpr'` (implicit data)
  - `'hinge'` / `'adaptive_hinge'` (implicit data)
  - `'warp'` (implicit data)
  - `'mse'` (explicit data)
  - `'mae'` (explicit data)

  For implicit data, if `train.num_negative_samples > 1`, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

  If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are `num_items`, `positive_items`, `negative_items`, `metadata`, and `metadata_weights`. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.
- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values `<= 1`. e.g. If `metadata_for_loss_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit

- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of `model.save_model()` method. Note that datasets and optimizers will NOT be restored. If `None`, will initialize model as normal
- **map_location** (*str or torch.device*) – If `load_model_path` is provided, device specifying how to remap storage locations when `torch.load`-ing the state dictionary

forward(*users: tensor*, *items: tensor*) → tensor

Forward pass through the model.

Parameters

- **users** (*tensor, 1-d*) – Array of user indices
- **items** (*tensor, 1-d*) – Array of item indices

Returns

preds – Predicted ratings or rankings

Return type

tensor, 1-d

freeze_embeddings() → None

Remove gradient requirement from the embeddings.

load_from_hybrid_model(*hybrid_model*) → None

Copy hyperparameters and state dictionary from an existing `HybridPretrainedModel` instance.

This is particularly useful for creating another PyTorch Lightning trainer object to fine-tune copied-over embeddings from a `MatrixFactorizationModel` instance.

Parameters

hybrid_model (`collie.model.HybridPretrainedModel`) – HybridPretrainedModel containing hyperparameters and state dictionary to copy over

save_model(*path: Union[str, Path] = 'data/model'*, *overwrite: bool = False*) → None

Save the model's state dictionary, hyperparameters, and user and/or item metadata.

While PyTorch Lightning offers a way to save and load models, there are two main reasons for overriding these:

1) To properly save and load a model requires the `Trainer` object, meaning that all deployed models will require Lightning to run the model, which is not actually needed for inference.
2) In the v0.8.4 release, loading a model back in leads to a `RuntimeError` unable to load in weights.

Parameters

- **path** (*str or Path*) – Directory path to save model and data files
- **overwrite** (*bool*) – Whether or not to overwrite existing data

unfreeze_embeddings() → None

Require gradients for the embeddings.

4.3 Multi-Stage Models

4.3.1 Cold Start Matrix Factorization Model

class collie.model.**ColdStartModel**(*train: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.ApproximateNegativeSamplin ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, val: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.ApproximateNegativeSamplin ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, item_buckets: ~typing.Optional[~typing.Iterable[int]] = None, embedding_dim: int = 30, dropout_p: float = 0.0, sparse: bool = False, item_buckets_stage_lr: float = 0.001, no_buckets_stage_lr: float = 0.001, lr_scheduler_func: ~typing.Optional[~torch.optim.lr_scheduler._LRScheduler] = functools.partial(<class 'torch.optim.lr_scheduler.ReduceLROnPlateau'>, patience=1, verbose=False), weight_decay: float = 0.0, item_buckets_stage_optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', no_buckets_stage_optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', loss: ~typing.Union[str, ~typing.Callable[[...], ~torch._VariableFunctionsClass.tensor]] = 'hinge', metadata_for_loss: ~typing.Optional[~typing.Dict[str, ~torch._VariableFunctionsClass.tensor]] = None, metadata_for_loss_weights: ~typing.Optional[~typing.Dict[str, float]] = None, load_model_path: ~typing.Optional[str] = None, map_location: ~typing.Optional[str] = None*)

Bases: *MultiStagePipeline*

Training pipeline for a matrix factorization model optimized for the cold-start problem.

Many recommendation models suffer from the cold start problem, when a model is unable to provide adequate recommendations for a new item until enough users have interacted with it. But, if users only interact with recommended items, the item will never be recommended, and thus the model will never improve recommendations for this item.

The `ColdStartModel` attempts to bypass this by limiting the item space down to "item buckets", training a model on this as the item space, then expanding out to all items. During this expansion, the learned-embeddings of each bucket is copied over to each corresponding item, providing a smarter initialization than a random one for both existing and new items. Now, when we have a new item, we can use its bucket embedding as an initialization into a model.

The stages in a `ColdStartModel` are, in order:

1. **`item_buckets`**

   Matrix factorization with item embeddings and bias terms bucketed by `item_buckets` argument. Unlike in the next stage, many items may map on to a single bucket, and this will share the same embedding and bias representation. The model should learn user preference for buckets in this stage.

2. **`no_buckets`**

   Standard matrix factorization as we do in `MatrixFactorizationModel`. However, upon advancing to this stage, the item embeddings are initialized with their bucketed embedding value (and same for biases). Not only does this provide better initialization than random, but allows new items to be incorporated into the model without training by using their item bucket embedding and bias terms at prediction time.

Note that the cold start problem exists for new users as well, but this functionality will be added to this model in a future version.

All `ColdStartModel` instances are subclasses of the `LightningModule` class provided by PyTorch Lightning. This means to train a model, you will need a `collie.model.CollieTrainer` object, but the model can be saved and loaded without this `Trainer` instance. Example usage may look like:

```python
from collie.model import ColdStartModel, CollieTrainer

# instantiate and fit a ``ColdStartModel`` as expected
model = ColdStartModel(train=train, item_buckets=item_buckets)
trainer = CollieTrainer(model)
trainer.fit(model)

# train for X more epochs on the next stage, ``no_buckets``
trainer.max_epochs += X
model.advance_stage()
trainer.fit(model)

model.eval()

# do evaluation as normal with ``model``

# get item-item recommendations for a new item by using the bucket ID, Z
similar_items = model.item_bucket_item_similarity(item_bucket_id=Z)

model.save_model(filename='model.pth')
new_model = ColdStartModel(load_model_path='model.pth')

# do evaluation as normal with ``new_model``
```

Parameters

- **`train`** (`collie.interactions` object) – Data loader for training data. If an `Interactions` object is supplied, an `InteractionsDataLoader` will automatically be instantiated with `shuffle=True`
- **`val`** (`collie.interactions` object) – Data loader for validation data. If an `Interactions` object is supplied, an `InteractionsDataLoader` will automatically be instantiated with `shuffle=False`
- **`item_buckets`** (*`torch.tensor, 1-d`*) – An ordered iterable containing the bucket ID for each item ID. For example, if you have five films and are going to bucket by primary genre, and your data looks like:
  - Item ID: 0, Genre ID: 1
  - Item ID: 1, Genre ID: 0
  - Item ID: 2, Genre ID: 2
  - Item ID: 3, Genre ID: 2
  - Item ID: 4, Genre ID: 1

  Then `item_buckets` would be: `[1, 0, 2, 2, 1]`

- **embedding_dim** (*int*) – Number of latent factors to use for user and item embeddings
- **dropout_p** (*float*) – Probability of dropout
- **item_buckets_stage_lr** (*float*) – Optimizer used for user parameters and item bucket parameters optimized during the `item_buckets` stage. If a string, one of the following supported optimizers:
  - `'sgd'` (for `torch.optim.SGD`)
  - `'adam'` (for `torch.optim.Adam`)
- **no_buckets_stage_lr** (*float*) – Optimizer used for user parameters and item parameters optimized during the `no_buckets` stage. If a string, one of the following supported optimizers:
  - `'sgd'` (for `torch.optim.SGD`)
  - `'adam'` (for `torch.optim.Adam`)
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **loss** (*function or str*) – If a string, one of the following implemented losses:
  - `'bpr'` / `'adaptive_bpr'` (implicit data)
  - `'hinge'` / `'adaptive_hinge'` (implicit data)
  - `'warp'` (implicit data)
  - `'mse'` (explicit data)
  - `'mae'` (explicit data)

  For implicit data, if `train.num_negative_samples > 1`, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

  If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are `num_items`, `positive_items`, `negative_items`, `metadata`, and `metadata_weights`. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.
- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values `<= 1`. e.g. If `metadata_for_loss_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,

  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit

- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of `model.save_model()` method. Note that datasets and optimizers will NOT be restored. If `None`, will initialize model as normal
- **map_location** (*str or torch.device*) – If `load_model_path` is provided, device specifying how to remap storage locations when `torch.load`-ing the state dictionary

Notes

The `forward` calculation will be different depending on the stage that is set. Note this when evaluating / saving and loading models in.

forward(*users: tensor*, *items: tensor*) → tensor

Forward pass through the model.

Parameters

- **users** (*tensor, 1-d*) – Array of user indices
- **items** (*tensor, 1-d*) – Array of item indices

Returns

preds – Predicted ratings or rankings

Return type

tensor, 1-d

item_bucket_item_similarity(*item_bucket_id: int*) → Series

Get most similar item indices to a item bucket by cosine similarity.

Cosine similarity is computed with item and item bucket embeddings from a trained model.

Parameters

item_id (*int*) –

Returns

sim_score_idxs – Sorted values as cosine similarity for each item in the dataset with the index being the item ID

Return type

pd.Series

set_stage(*stage: str*) → None

Set the stage for the model.

4.3.2 Hybrid Matrix Factorization Model

class collie.model.**HybridModel**(*train: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.ApproximateNegativeSamplingInt ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, val: ~typing.Optional[~typing.Union[~collie.interactions.dataloaders.ApproximateNegativeSamplingInt ~collie.interactions.datasets.Interactions, ~collie.interactions.dataloaders.InteractionsDataLoader]] = None, item_metadata: ~typing.Optional[~typing.Union[~torch._VariableFunctionsClass.tensor, ~pandas.core.frame.DataFrame, ~numpy.array]] = None, user_metadata: ~typing.Optional[~typing.Union[~torch._VariableFunctionsClass.tensor, ~pandas.core.frame.DataFrame, ~numpy.array]] = None, embedding_dim: int = 30, item_metadata_layers_dims: ~typing.Optional[~typing.List[int]] = None, user_metadata_layers_dims: ~typing.Optional[~typing.List[int]] = None, combined_layers_dims: ~typing.List[int] = [128, 64, 32], dropout_p: float = 0.0, lr: float = 0.001, bias_lr: ~typing.Optional[~typing.Union[float, str]] = 0.01, metadata_only_stage_lr: float = 0.001, all_stage_lr: float = 0.0001, lr_scheduler_func: ~typing.Optional[~torch.optim.lr_scheduler._LRScheduler] = functools.partial(<class 'torch.optim.lr_scheduler.ReduceLROnPlateau'>, patience=1, verbose=False), weight_decay: float = 0.0, optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', bias_optimizer: ~typing.Optional[~typing.Union[str, ~torch.optim.optimizer.Optimizer]] = 'sgd', metadata_only_stage_optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', all_stage_optimizer: ~typing.Union[str, ~torch.optim.optimizer.Optimizer] = 'adam', loss: ~typing.Union[str, ~typing.Callable[[...], ~torch._VariableFunctionsClass.tensor]] = 'hinge', metadata_for_loss: ~typing.Optional[~typing.Dict[str, ~torch._VariableFunctionsClass.tensor]] = None, metadata_for_loss_weights: ~typing.Optional[~typing.Dict[str, float]] = None, load_model_path: ~typing.Optional[str] = None, map_location: ~typing.Optional[str] = None*)

Bases: `MultiStagePipeline`

Training pipeline for a multi-stage hybrid recommendation model.

`HybridModel` models contain dense layers that process item and/or user metadata, concatenate this embedding with user and item embeddings, sending this concatenated embedding through more dense layers to output a single float ranking / rating. We add both user and item biases to this score before returning. This is the same architecture as the `HybridPretrainedModel`, but we are training the embeddings ourselves rather than relying on pulling this from a pre-trained model.

The stages in a `HybridModel` depend on whether both item and user metadata is used. For the full model, they are, in order:

1. **`matrix_factorization`**
   Matrix factorization exactly as we do in `MatrixFactorizationModel`. In this stage, metadata is NOT incorporated into the model.
2. **`metadata_only`**
   User and item embeddings terms are frozen, and the MLP layers for the metadata (if specified) and combined embedding-metadata data are optimized.
3. **`all`**
   Embedding and MLP layers are all optimized together, including those for metadata.

All `HybridModel` instances are subclasses of the `LightningModule` class provided by PyTorch Lightning. This means to train a model, you will need a `collie.model.CollieTrainer` object, but the model can be saved and loaded without this `Trainer` instance. Example usage may look like:

```
from collie.model import CollieTrainer, HybridModel

# instantiate and fit a ``HybridModel`` as expected
model = HybridModel(train=train,
                    item_metadata=item_metadata,
                    user_metadata=user_metadata)
trainer = CollieTrainer(model)
trainer.fit(model)

# train for X more epochs on the next stage, ``metadata_only``
trainer.max_epochs += X
model.advance_stage()
trainer.fit(model)

# train for Y more epochs on the next stage, ``all``
trainer.max_epochs += Y
model.advance_stage()
trainer.fit(model)

model.eval()

# do evaluation as normal with ``model``

model.save_model(path='model')
new_model = HybridModel(load_model_path='model')

# do evaluation as normal with ``new_model``
```

Parameters

- **train** (`collie.interactions` object) – Data loader for training data. If an `Interactions` object is supplied, an `InteractionsDataLoader` will automatically be instantiated with `shuffle=True`
- **val** (`collie.interactions` object) – Data loader for validation data. If an `Interactions` object is supplied, an `InteractionsDataLoader` will automatically be instantiated with `shuffle=False`
- **item_metadata** (*torch.tensor, pd.DataFrame, or np.array, 2-dimensional*) – The shape of the item metadata should be (num_items x metadata_features), and each item's metadata should be available when indexing a row by an item ID
- **user_metadata** (*torch.tensor, pd.DataFrame, or np.array, 2-dimensional*) – The shape of the user metadata should be (num_users x metadata_features), and each user's metadata should be available when indexing a row by a user ID
- **embedding_dim** (*int*) – Number of latent factors to use for user and item embeddings
- **item_metadata_layers_dims** (*list*) – List of linear layer dimensions to apply to the item metadata only, starting with the dimension directly following `item_metadata_features` and ending with the dimension to concatenate with the item embeddings
- **user_metadata_layers_dims** (*list*) – List of linear layer dimensions to apply to the user

metadata only, starting with the dimension directly following `user_metadata_features` and ending with the dimension to concatenate with the user embeddings

- **combined_layers_dims** (*list*) – List of linear layer dimensions to apply to the concatenated item embeddings and item metadata, starting with the dimension directly following the shape of `item_embeddings + metadata_features` and ending with the dimension before the final linear layer to dimension 1
- **dropout_p** (*float*) – Probability of dropout
- **metadata_only_stage_lr** (*float*) – Learning rate for metadata and combined layers optimized during the `metadata_only` stage
- **all_stage_lr** (*float*) – Learning rate for all model parameters optimized during the `all` stage
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **optimizer** (*torch.optim or str*) – Optimizer used for embeddings and bias terms (if `bias_optimizer` is `None`) during the `matrix_factorization` stage. If a string, one of the following supported optimizers:
  - `'sgd'` (for `torch.optim.SGD`)
  - `'adam'` (for `torch.optim.Adam`)
- **metadata_only_stage_optimizer** (*torch.optim or str*) – Optimizer used for metadata and combined layers during the `metadata_only` stage. If a string, one of the following supported optimizers:
  - `'sgd'` (for `torch.optim.SGD`)
  - `'adam'` (for `torch.optim.Adam`)
- **all_stage_optimizer** (*torch.optim or str*) – Optimizer used for all model parameters during the `all` stage. If a string, one of the following supported optimizers:
  - `'sgd'` (for `torch.optim.SGD`)
  - `'adam'` (for `torch.optim.Adam`)
- **loss** (*function or str*) – If a string, one of the following implemented losses:
  - `'bpr'` / `'adaptive_bpr'` (implicit data)
  - `'hinge'` / `'adaptive_hinge'` (implicit data)
  - `'warp'` (implicit data)
  - `'mse'` (explicit data)
  - `'mae'` (explicit data)

  For implicit data, if `train.num_negative_samples > 1`, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

  If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are `num_items`, `positive_items`, `negative_items`, `metadata`, and `metadata_weights`. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.

- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values `<= 1`. e.g. If `metadata_for_loss_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit
- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of `model.save_model()` method. Note that datasets and optimizers will NOT be restored. If `None`, will initialize model as normal
- **map_location** (*str or torch.device*) – If `load_model_path` is provided, device specifying how to remap storage locations when `torch.load`-ing the state dictionary

Notes

The `forward` calculation will be different depending on the stage that is set. Note this when evaluating / saving and loading models in.

forward(*users: tensor*, *items: tensor*) → tensor

Forward pass through the model.

Parameters

- **users** (*tensor, 1-d*) – Array of user indices
- **items** (*tensor, 1-d*) – Array of item indices

Returns

preds – Predicted ratings or rankings

Return type

tensor, 1-d

save_model(*path: Union[str, Path] = 'data/model'*, *overwrite: bool = False*) → None

Save the model's state dictionary, hyperparameters, and user and/or item metadata.

While PyTorch Lightning offers a way to save and load models, there are two main reasons for overriding these:

1) To properly save and load a model requires the `Trainer` object, meaning that all deployed models will require Lightning to run the model, which is not actually needed for inference.
2) In the v0.8.4 release, loading a model back in leads to a `RuntimeError` unable to load in weights.

Parameters

- **path** (*str or Path*) – Directory path to save model and data files
- **overwrite** (*bool*) – Whether or not to overwrite existing data

4.4 Trainers

4.4.1 PyTorch Lightning Trainer

class collie.model.**CollieTrainer**(*model: Module*, *max_epochs: int = 10*, *benchmark: bool = True*, *deterministic: bool = True*, ***kwargs*)

Bases: `Trainer`

Helper wrapper class around PyTorch Lightning's `Trainer` class.

Specifically, this wrapper:

- Checks if a model has a validation dataset passed in (under the `val_loader` attribute) and, if not, sets `num_sanity_val_steps` to 0 and `check_val_every_n_epoch` to `sys.maxint`.
- Checks if a GPU is available and, if `gpus is None`, sets `gpus = -1`.

See `pytorch_lightning.Trainer` documentation for more details at: https://pytorch-lightning.readthedocs.io/en/latest/common/trainer.html#trainer-class-api

Compared with `CollieMinimalTrainer`, PyTorch Lightning's `Trainer` offers more flexibility and room for exploration, at the cost of a higher training time (which is especially true for larger models). We recommend starting all model exploration with this `CollieTrainer` (callbacks, automatic Lightning optimizations, etc.), finding a set of hyperparameters that work for your training job, then using this in the simpler but faster `CollieMinimalTrainer`.

Parameters

- **model** (`collie.model.BasePipeline`) – Initialized Collie model
- **max_epochs** (*int*) – Stop training once this number of epochs is reached
- **benchmark** (*bool*) – If set to `True`, enables `cudnn.benchmark`
- **deterministic** (*bool*) – If set to `True`, enables `cudnn.deterministic`
- ****kwargs** (*keyword arguments*) – Additional keyword arguments to be sent to the `Trainer` class: https://pytorch-lightning.readthedocs.io/en/latest/common/trainer.html#trainer-class-api
- **follows** (*Original pytorch_lightning.Trainer docstring as*) –
- **########** – Customize every aspect of training via flags.

  Args:

  accelerator: Supports passing different accelerator types ("cpu", "gpu", "tpu", "ipu", "hpu", "mps", "auto")
  as well as custom accelerator instances.

  accumulate_grad_batches: Accumulates grads every k batches or as set up in the dict.
  Default: `None`.

  amp_backend: The mixed precision backend to use ("native" or "apex").
  Default: `'native''`.

Deprecated since version v1.9: Setting `amp_backend` inside the `Trainer` is deprecated in v1.8.0 and will be removed in v2.0.0. This argument was only relevant for apex which is being removed.

amp_level: The optimization level to use (O1, O2, etc…). By default it will be set to "O2"
if `amp_backend` is set to "apex".

Deprecated since version v1.8: Setting `amp_level` inside the `Trainer` is deprecated in v1.8.0 and will be removed in v2.0.0.

auto_lr_find: If set to True, will make trainer.tune() run a learning rate finder,
trying to optimize initial learning for faster convergence. trainer.tune() method will set the suggested learning rate in self.lr or self.learning_rate in the LightningModule. To use a different key set a string instead of True with the key name. Default: `False`.

auto_scale_batch_size: If set to True, will *initially* run a batch size
finder trying to find the largest batch size that fits into memory. The result will be stored in self.batch_size in the LightningModule or LightningDataModule depending on your setup. Additionally, can be set to either *power* that estimates the batch size through a power search or *binsearch* that estimates the batch size through a binary search. Default: `False`.

auto_select_gpus: If enabled and `gpus` or `devices` is an integer, pick available
gpus automatically. This is especially useful when GPUs are configured to be in "exclusive mode", such that only one process at a time can access them. Default: `False`.

Deprecated since version v1.9: `auto_select_gpus` has been deprecated in v1.9.0 and will be removed in v2.0.0. Please use the function `find_usable_cuda_devices()` instead.

benchmark: The value (`True` or `False`) to set `torch.backends.cudnn.benchmark` to.
The value for `torch.backends.cudnn.benchmark` set in the current session will be used (`False` if not manually set). If **:paramref:`~pytorch_lightning.trainer.Trainer.deterministic`** is set to `True`, this will default to `False`. Override to manually set a different value. Default: `None`.

callbacks: Add a callback or list of callbacks.
Default: `None`.

enable_checkpointing: If `True`, enable checkpointing.
It will configure a default ModelCheckpoint callback if there is no user-defined ModelCheckpoint in **:paramref:`~pytorch_lightning.trainer.trainer.Trainer.callbacks`**. Default: `True`.

check_val_every_n_epoch: Perform a validation loop every after every *N* training epochs. If `None`,
validation will be done solely based on the number of training batches, requiring `val_check_interval` to be an integer value. Default: `1`.

default_root_dir: Default path for logs and weights when no logger/ckpt_callback passed.
Default: `os.getcwd()`. Can be remote file paths such as *s3://mybucket/path* or 'hdfs://path/'

detect_anomaly: Enable anomaly detection for the autograd engine.
Default: `False`.

deterministic: If `True`, sets whether PyTorch operations must use deterministic algorithms.
Set to `"warn"` to use deterministic algorithms whenever possible, throwing warnings on operations that don't support deterministic mode (requires PyTorch 1.11+). If not set, defaults to `False`. Default: `None`.

devices: Will be mapped to either *gpus*, *tpu_cores*, *num_processes* or *ipus*,
based on the accelerator type.

fast_dev_run: Runs n if set to `n` (int) else 1 if set to `True` batch(es)
of train, val and test to find any bugs (ie: a sort of unit test). Default: `False`.

gpus: Number of GPUs to train on (int) or which GPUs to train on (list or str) applied per node
Default: `None`.

Deprecated since version v1.7: `gpus` has been deprecated in v1.7 and will be removed in v2.0. Please use `accelerator='gpu'` and `devices=x` instead.

gradient_clip_val: The value at which to clip gradients. Passing `gradient_clip_val=None` disables
gradient clipping. If using Automatic Mixed Precision (AMP), the gradients will be unscaled before. Default: `None`.

gradient_clip_algorithm: The gradient clipping algorithm to use. Pass `gradient_clip_algorithm="value"`
to clip by value, and `gradient_clip_algorithm="norm"` to clip by norm. By default it will be set to `"norm"`.

limit_train_batches: How much of training dataset to check (float = fraction, int = num_batches).
Default: `1.0`.

limit_val_batches: How much of validation dataset to check (float = fraction, int = num_batches).
Default: `1.0`.

limit_test_batches: How much of test dataset to check (float = fraction, int = num_batches).
Default: `1.0`.

limit_predict_batches: How much of prediction dataset to check (float = fraction, int = num_batches).
Default: `1.0`.

logger: Logger (or iterable collection of loggers) for experiment tracking. A `True` value uses
the default `TensorBoardLogger` if it is installed, otherwise `CSVLogger`. `False` will disable logging. If multiple loggers are provided, local files (checkpoints, profiler traces, etc.) are saved in the `log_dir` of he first logger. Default: `True`.

log_every_n_steps: How often to log within steps.
Default: `50`.

enable_progress_bar: Whether to enable to progress bar by default.
Default: `True`.

profiler: To profile individual steps during training and assist in identifying bottlenecks.
Default: `None`.

overfit_batches: Overfit a fraction of training/validation data (float) or a set number of batches (int).
Default: `0.0`.

plugins: Plugins allow modification of core behavior like ddp and amp, and enable custom lightning plugins.
Default: `None`.

precision: Double precision (64), full precision (32), half precision (16) or bfloat16 precision (bf16).
Can be used on CPU, GPU, TPUs, HPUs or IPUs. Default: `32`.

max_epochs: Stop training once this number of epochs is reached. Disabled by default (None).
If both max_epochs and max_steps are not specified, defaults to `max_epochs = 1000`. To enable infinite training, set `max_epochs = -1`.

min_epochs: Force training for at least these many epochs. Disabled by default (None).

max_steps: Stop training after this number of steps. Disabled by default (-1). If `max_steps = -1`
and `max_epochs = None`, will default to `max_epochs = 1000`. To enable infinite training, set `max_epochs` to `-1`.

min_steps: Force training for at least these number of steps. Disabled by default (`None`).

max_time: Stop training after this amount of time has passed. Disabled by default (`None`).
The time duration can be specified in the format DD:HH:MM:SS (days, hours, minutes seconds), as a `datetime.timedelta`, or a dictionary with keys that will be passed to `datetime.timedelta`.

num_nodes: Number of GPU nodes for distributed training.
Default: `1`.

num_processes: Number of processes for distributed training with `accelerator="cpu"`.
Default: `1`.

Deprecated since version v1.7: `num_processes` has been deprecated in v1.7 and will be removed in v2.0. Please use `accelerator='cpu'` and `devices=x` instead.

num_sanity_val_steps: Sanity check runs n validation batches before starting the training routine.
Set it to *-1* to run all batches in all validation dataloaders. Default: `2`.

reload_dataloaders_every_n_epochs: Set to a non-negative integer to reload dataloaders every n epochs.
Default: `0`.

replace_sampler_ddp: Explicitly enables or disables sampler replacement. If not specified this
will toggled automatically when DDP is used. By default it will add `shuffle=True` for train sampler and `shuffle=False` for val/test sampler. If you want to customize it, you can set `replace_sampler_ddp=False` and add your own distributed sampler.

resume_from_checkpoint: Path/URL of the checkpoint from which training is resumed. If there is
no checkpoint file at the path, an exception is raised. If resuming from mid-epoch checkpoint, training will start from the beginning of the next epoch.

Deprecated since version v1.5: `resume_from_checkpoint` is deprecated in v1.5 and will be removed in v2.0. Please pass the path to `Trainer.fit(..., ckpt_path=...)` instead.

strategy: Supports different training strategies with aliases
as well custom strategies. Default: `None`.

sync_batchnorm: Synchronize batch norm layers between process groups/whole world.
Default: `False`.

tpu_cores: How many TPU cores to train on (1 or 8) / Single TPU to train on (1)
Default: `None`.

Deprecated since version v1.7: `tpu_cores` has been deprecated in v1.7 and will be removed in v2.0. Please use `accelerator='tpu'` and `devices=x` instead.

ipus: How many IPUs to train on.
Default: `None`.

Deprecated since version v1.7: `ipus` has been deprecated in v1.7 and will be removed in v2.0. Please use `accelerator='ipu'` and `devices=x` instead.

track_grad_norm: -1 no tracking. Otherwise tracks that p-norm. May be set to 'inf' infinity-norm. If using
Automatic Mixed Precision (AMP), the gradients will be unscaled before logging them. Default: `-1`.

val_check_interval: How often to check the validation set. Pass a `float` in the range [0.0, 1.0] to check
after a fraction of the training epoch. Pass an `int` to check after a fixed number of training batches. An `int` value can only be higher than the number of training batches when `check_val_every_n_epoch=None`, which validates after every `N` training batches across epochs or during iteration-based training. Default: `1.0`.

enable_model_summary: Whether to enable model summarization by default.
Default: `True`.

move_metrics_to_cpu: Whether to force internal logged metrics to be moved to cpu.
This can save some gpu memory, but can make training slower. Use with attention. Default: `False`.

multiple_trainloader_mode: How to loop over the datasets when there are multiple train loaders.
In 'max_size_cycle' mode, the trainer ends one epoch when the largest dataset is traversed, and smaller datasets reload when running out of their data. In 'min_size' mode, all the datasets reload when reaching the minimum length of datasets. Default: `"max_size_cycle"`.

inference_mode: Whether to use `torch.inference_mode()` or `torch.no_grad()` during
evaluation (`validate`/`test`/`predict`).

`property checkpoint_callback: Optional[Checkpoint]`

The first `ModelCheckpoint` callback in the Trainer.callbacks list, or `None` if it doesn't exist.

`property checkpoint_callbacks: List[Checkpoint]`

A list of all instances of `ModelCheckpoint` found in the Trainer.callbacks list.

`property ckpt_path: Optional[str]`

Set to the path/URL of a checkpoint loaded via `fit()`, `validate()`, `test()`, or `predict()`. `None` otherwise.

`property current_epoch: int`

The current epoch, updated after the epoch end hooks are run.

property default_root_dir: str

The default location to save artifacts of loggers, checkpoints etc.

It is used as a fallback if logger or checkpoint callback do not define specific save paths.

property device_ids: List[int]

List of device indexes per node.

property early_stopping_callback: Optional[EarlyStopping]

The first `EarlyStopping` callback in the Trainer.callbacks list, or `None` if it doesn't exist.

property early_stopping_callbacks: List[EarlyStopping]

A list of all instances of `EarlyStopping` found in the Trainer.callbacks list.

property enable_validation: bool

Check if we should run validation during training.

property estimated_stepping_batches: Union[int, float]

Estimated stepping batches for the complete training inferred from DataLoaders, gradient accumulation factor and distributed setup.

Examples:

```
def configure_optimizers(self):
    optimizer = ...
    scheduler = torch.optim.lr_scheduler.OneCycleLR(
        optimizer, max_lr=1e-3, total_steps=self.trainer.estimated_stepping_
↪batches
    )
    return [optimizer], [scheduler]
```

fit(*model: LightningModule*, *train_dataloaders: Optional[Union[DataLoader, Sequence[DataLoader], Sequence[Sequence[DataLoader]], Sequence[Dict[str, DataLoader]], Dict[str, DataLoader], Dict[str, Dict[str, DataLoader]], Dict[str, Sequence[DataLoader]], LightningDataModule]] = None*, *val_dataloaders: Optional[Union[DataLoader, Sequence[DataLoader]]] = None*, *datamodule: Optional[LightningDataModule] = None*, *ckpt_path: Optional[str] = None*) → None

Runs the full optimization routine.

Parameters

- **model** – Model to fit.
- **train_dataloaders** – A collection of `torch.utils.data.DataLoader` or a `LightningDataModule` specifying training samples. In the case of multiple dataloaders, please see this section.
- **val_dataloaders** – A `torch.utils.data.DataLoader` or a sequence of them specifying validation samples.
- **ckpt_path** – Path/URL of the checkpoint from which training is resumed. Could also be one of two special keywords `"last"` and `"hpc"`. If there is no checkpoint file at the path, an exception is raised. If resuming from mid-epoch checkpoint, training will start from the beginning of the next epoch.
- **datamodule** – An instance of `LightningDataModule`.

property global_step: int

The number of optimizer steps taken (does not reset each epoch).

This includes multiple optimizers and TBPTT steps (if enabled).

`property is_last_batch: bool`

Whether trainer is executing the last batch.

`property max_epochs`

Property that just returns `max_epochs`, included only so we can have a setter for it without an `AttributeError`.

`property model: Optional[Module]`

The LightningModule, but possibly wrapped into DataParallel or DistributedDataParallel.

To access the pure LightningModule, use `lightning_module()` instead.

`property num_devices: int`

Number of devices the trainer uses per node.

`predict`(*model: Optional[LightningModule] = None*, *dataloaders: Optional[Union[DataLoader, Sequence[DataLoader], LightningDataModule]] = None*, *datamodule: Optional[LightningDataModule] = None*, *return_predictions: Optional[bool] = None*, *ckpt_path: Optional[str] = None*) → Optional[Union[List[Any], List[List[Any]]]]

Run inference on your data. This will call the model forward function to compute predictions. Useful to perform distributed and batched predictions. Logging is disabled in the predict hooks.

Parameters

- `model` – The model to predict with.
- `dataloaders` – A `torch.utils.data.DataLoader` or a sequence of them, or a `LightningDataModule` specifying prediction samples.
- `datamodule` – The datamodule with a predict_dataloader method that returns one or more dataloaders.
- `return_predictions` – Whether to return predictions. `True` by default except when an accelerator that spawns processes is used (not supported).
- `ckpt_path` – Either `"best"`, `"last"`, `"hpc"` or path to the checkpoint you wish to predict. If `None` and the model instance was passed, use the current weights. Otherwise, the best model checkpoint from the previous `trainer.fit` call will be loaded if a checkpoint callback is configured.

Returns

Returns a list of dictionaries, one for each provided dataloader containing their respective predictions.

See Lightning inference section for more.

`property prediction_writer_callbacks: List[BasePredictionWriter]`

A list of all instances of `BasePredictionWriter` found in the Trainer.callbacks list.

`property progress_bar_callback: Optional[ProgressBarBase]`

An instance of `ProgressBarBase` found in the Trainer.callbacks list, or `None` if one doesn't exist.

`reset_predict_dataloader`(*model: Optional[LightningModule] = None*) → None

Resets the predict dataloader and determines the number of batches.

Parameters

`model` – The `LightningModule` if called outside of the trainer scope.

`reset_test_dataloader`(*model: Optional[LightningModule] = None*) → None

Resets the test dataloader and determines the number of batches.

Parameters
`model` – The `LightningModule` if called outside of the trainer scope.

`reset_train_dataloader`(*model: Optional[LightningModule] = None*) → None

Resets the train dataloader and initialises required variables (number of batches, when to validate, etc.).

Parameters
`model` – The `LightningModule` if calling this outside of the trainer scope.

`reset_val_dataloader`(*model: Optional[LightningModule] = None*) → None

Resets the validation dataloader and determines the number of batches.

Parameters
`model` – The `LightningModule` if called outside of the trainer scope.

`save_checkpoint`(*filepath: Union[str, Path]*, *weights_only: bool = False*, *storage_options: Optional[Any] = None*) → None

Runs routine to create a checkpoint.

Parameters

- `filepath` – Path where checkpoint is saved.
- `weights_only` – If `True`, will only save the model weights.
- `storage_options` – parameter for how to save to storage, passed to `CheckpointIO` plugin

`test`(*model: Optional[LightningModule] = None*, *dataloaders: Optional[Union[DataLoader, Sequence[DataLoader], LightningDataModule]] = None*, *ckpt_path: Optional[str] = None*, *verbose: bool = True*, *datamodule: Optional[LightningDataModule] = None*) → List[Dict[str, float]]

Perform one evaluation epoch over the test set. It's separated from fit to make sure you never run on your test set until you want to.

Parameters

- `model` – The model to test.
- `dataloaders` – A `torch.utils.data.DataLoader` or a sequence of them, or a `LightningDataModule` specifying test samples.
- `ckpt_path` – Either `"best"`, `"last"`, `"hpc"` or path to the checkpoint you wish to test. If `None` and the model instance was passed, use the current weights. Otherwise, the best model checkpoint from the previous `trainer.fit` call will be loaded if a checkpoint callback is configured.
- `verbose` – If True, prints the test results.
- `datamodule` – An instance of `LightningDataModule`.

Returns
List of dictionaries with metrics logged during the test phase, e.g., in model- or callback hooks like `test_step()`, `test_epoch_end()`, etc. The length of the list corresponds to the number of test dataloaders used.

`tune`(*model: LightningModule*, *train_dataloaders: Optional[Union[DataLoader, Sequence[DataLoader], Sequence[Sequence[DataLoader]], Sequence[Dict[str, DataLoader]], Dict[str, DataLoader], Dict[str, Dict[str, DataLoader]], Dict[str, Sequence[DataLoader]], LightningDataModule]] = None*, *val_dataloaders: Optional[Union[DataLoader, Sequence[DataLoader]]] = None*, *dataloaders: Optional[Union[DataLoader, Sequence[DataLoader]]] = None*, *datamodule: Optional[LightningDataModule] = None*, *scale_batch_size_kwargs: Optional[Dict[str, Any]] = None*, *lr_find_kwargs: Optional[Dict[str, Any]] = None*, *method: Literal['fit', 'validate', 'test', 'predict'] = 'fit'*) → _TunerResult

Runs routines to tune hyperparameters before training.

Parameters

- **model** – Model to tune.
- **train_dataloaders** – A collection of `torch.utils.data.DataLoader` or a `LightningDataModule` specifying training samples. In the case of multiple dataloaders, please see this section.
- **val_dataloaders** – A `torch.utils.data.DataLoader` or a sequence of them specifying validation samples.
- **dataloaders** – A `torch.utils.data.DataLoader` or a sequence of them specifying val/test/predict samples used for running tuner on validation/testing/prediction.
- **datamodule** – An instance of `LightningDataModule`.
- **scale_batch_size_kwargs** – Arguments for `scale_batch_size()`
- **lr_find_kwargs** – Arguments for `lr_find()`
- **method** – Method to run tuner on. It can be any of `("fit", "validate", "test", "predict")`.

`validate`(*model: Optional[LightningModule] = None*, *dataloaders: Optional[Union[DataLoader, Sequence[DataLoader], LightningDataModule]] = None*, *ckpt_path: Optional[str] = None*, *verbose: bool = True*, *datamodule: Optional[LightningDataModule] = None*) → List[Dict[str, float]]

Perform one evaluation epoch over the validation set.

Parameters

- **model** – The model to validate.
- **dataloaders** – A `torch.utils.data.DataLoader` or a sequence of them, or a `LightningDataModule` specifying validation samples.
- **ckpt_path** – Either `"best"`, `"last"`, `"hpc"` or path to the checkpoint you wish to validate. If `None` and the model instance was passed, use the current weights. Otherwise, the best model checkpoint from the previous `trainer.fit` call will be loaded if a checkpoint callback is configured.
- **verbose** – If True, prints the validation results.
- **datamodule** – An instance of `LightningDataModule`.

Returns

List of dictionaries with metrics logged during the validation phase, e.g., in model- or callback hooks like `validation_step()`, `validation_epoch_end()`, etc. The length of the list corresponds to the number of validation dataloaders used.

4.4.2 Non- PyTorch Lightning Trainer

class `collie.model.`**`CollieMinimalTrainer`**(*model: BasePipeline*, *max_epochs: int = 10*, *gpus: Optional[Union[bool, int]] = None*, *logger: Optional[Logger] = None*, *early_stopping_patience: Optional[int] = 3*, *log_every_n_steps: int = 50*, *flush_logs_every_n_steps: int = 100*, *enable_model_summary: bool = True*, *weights_summary: Optional[str] = None*, *detect_anomaly: bool = False*, *terminate_on_nan: Optional[bool] = None*, *benchmark: bool = True*, *deterministic: bool = True*, *progress_bar_refresh_rate: Optional[int] = None*, *verbosity: Union[bool, int] = True*)

A more manual implementation of PyTorch Lightning's `Trainer` class, attempting to port over the most commonly used `Trainer` arguments into a training loop with more transparency and faster training times.

Through extensive experimentation, we found that PyTorch Lightning's `Trainer` was training Collie models about 25% slower than the more manual, typical PyTorch training loop boilerplate. Thus, we created the `CollieMinimalTrainer`, which shares a similar API to PyTorch Lightning's `Trainer` object (both in instantiation and in usage), with a standard PyTorch training loop in its place.

While PyTorch Lightning's `Trainer` offers more flexibility and customization through the addition of the additional `Trainer` arguments and `callbacks`, we designed this class as a way to train a model in production, where we might be more focused on faster training times and less on hyperparameter tuning and R&D, where one might instead opt to use PyTorch Lightning's `Trainer` class.

Note that the arguments the `CollieMinimalTrainer` trainer accepts will be slightly different than the ones that the `CollieTrainer` accept, and defaults are also not guaranteed to be equal as the two libraries evolve. Notable changes are:

- If `gpus > 1`, only a single GPU will be used and any other GPUs will remain unused. Multi- GPU training is not supported in `CollieMinimalTrainer` at this time.
- `logger == True` has no meaning in `CollieMinimalTrainer` - a default logger will NOT be created if set to `True`.
- There is no way to pass in `callbacks` at this time. Instead, we will implement the most used ones during training here, manually, in favor of greater speed over customization. To use early stopping, set the `early_stopping_patience` to an integer other than `None`.

```
from collie.model import CollieMinimalTrainer, MatrixFactorizationModel


# notice how similar the usage is to the standard ``CollieTrainer``
model = MatrixFactorizationModel(train=train)
trainer = CollieMinimalTrainer(model)
trainer.fit(model)
```

Model results should NOT be significantly different whether trained with `CollieTrainer` or `CollieMinimalTrainer`.

If there's an argument you would like to see added to `CollieMinimalTrainer` that is present in `CollieTrainer` used during productionalized model training, make an Issue or a PR in GitHub!

Parameters

- **`model`** (`collie.model.BasePipeline`) – Initialized Collie model
- **`max_epochs`** (*`int`*) – Stop training once this number of epochs is reached

- **gpus** (*bool or int*) – Whether to train on the GPU (`gpus == True` or `gpus > 0`) or the CPU
- **logger** (*LightningLoggerBase*) – Logger for experiment tracking. Set `logger = None` or `logger = False` to disable logging
- **early_stopping_patience** (*int*) – Number of epochs of patience to have without any improvement in loss before stopping training early. Validation epoch loss will be used if there is a validation DataLoader present, else training epoch loss will be used. Set `early_stopping_patience = None` or `early_stopping_patience = False` to disable early stopping
- **log_every_n_steps** (*int*) – How often to log within steps, if `logger` is enabled
- **flush_logs_every_n_steps** (*int*) – How often to flush logs to disk, if `logger` is enabled
- **enable_model_summary** (*bool*) – Whether to enable or disable the model summarization
- **weights_summary** (*str*) – Deprecated, replaced with `enable_model_summary`. Prints summary of the weights when training begins
- **detect_anomaly** (*bool*) – Context-manager that enable anomaly detection for the autograd engine. This does two things:
  - Running the forward pass with detection enabled will allow the backward pass to print the traceback of the forward operation that created the failing backward function.
  - Any backward computation that generate "nan" value will raise an error.

  Warning: This mode should be enabled only for debugging as the different tests will slow down your program execution.
- **terminate_on_nan** (*bool*) – Deprecated, replaced with `detect_anomaly`. If set to `True`, will terminate training (by raising a `ValueError`) at the end of each training batch, if any of the parameters or the loss are NaN or +/- infinity
- **benchmark** (*bool*) – If set to `True`, enables `cudnn.benchmark`
- **deterministic** (*bool*) – If set to `True`, enables `cudnn.deterministic`
- **progress_bar_refresh_rate** (*int*) – How often to refresh progress bar (in steps), if `verbosity > 0`
- **verbosity** (*Union[bool, int]*) – How verbose to be in training.
  - `0` disables all printouts, including `weights_summary`
  - 1 prints `weights_summary` (if applicable) and epoch losses
  - 2 prints `weights_summary` (if applicable), epoch losses, and progress bars

fit(*model:* BasePipeline) → None

Runs the full optimization routine.

Parameters

model (`collie.model.BasePipeline`) – Initialized Collie model

property max_epochs

Property that just returns `max_epochs`, included only so we can have a setter for it without an `AttributeError`.

4.5 Model Templates

4.5.1 Base Collie Pipeline Template

class collie.model.**BasePipeline**(*train: Optional[Union[*ApproximateNegativeSamplingInteractionsDataLoader, ExplicitInteractions, Interactions, InteractionsDataLoader*]] = None*, *val: Optional[Union[*ApproximateNegativeSamplingInteractionsDataLoader, ExplicitInteractions, Interactions, InteractionsDataLoader*]] = None*, *lr: float = 0.001*, *lr_scheduler_func: Optional[_LRScheduler] = None*, *weight_decay: float = 0.0*, *optimizer: Union[str, Optimizer] = 'adam'*, *loss: Union[str, Callable[[...], tensor]] = 'hinge'*, *metadata_for_loss: Optional[Dict[str, tensor]] = None*, *metadata_for_loss_weights: Optional[Dict[str, float]] = None*, *load_model_path: Optional[str] = None*, *map_location: Optional[str] = None*, ***kwargs*)

Bases: `LightningModule`

Base Pipeline model architectures to inherit from.

All subclasses MUST at least override the following methods:

- `_setup_model` - Set up the model architecture
- `forward` - Forward pass through a model

For `item_item_similarity` to work properly, all subclasses are should also implement:

- `_get_item_embeddings` - Returns item embeddings from the model on the device

For `user_user_similarity` to work properly, all subclasses are should also implement:

- `_get_user_embeddings` - Returns user embeddings from the model on the device

Parameters

- **train** (`collie.interactions` object) – Data loader for training data. If an `Interactions` object is supplied, an `InteractionsDataLoader` will automatically be instantiated with `shuffle=True`
- **val** (`collie.interactions` object) – Data loader for validation data. If an `Interactions` object is supplied, an `InteractionsDataLoader` will automatically be instantiated with `shuffle=False`
- **lr** (*float*) – Model learning rate
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **optimizer** (*torch.optim or str*) – If a string, one of the following supported optimizers:
  - `'sgd'` (for `torch.optim.SGD`)
  - `'adagrad'` (for `torch.optim.Adagrad`)
  - `'adam'` (for `torch.optim.Adam`)
  - `'sparse_adam'` (for `torch.optim.SparseAdam`)
- **loss** (*function or str*) – If a string, one of the following implemented losses:

- `'bpr'` / `'adaptive_bpr'` (implicit data)
- `'hinge'` / `'adaptive_hinge'` (implicit data)
- `'warp'` (implicit data)
- `'mse'` (explicit data)
- `'mae'` (explicit data)

For implicit data, if `train.num_negative_samples > 1`, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are `num_items`, `positive_items`, `negative_items`, `metadata`, and `metadata_weights`. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.

- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata_weights`. Values should be a `torch.tensor` of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)
- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in `metadata`. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values <= 1. e.g. If `metadata_for_loss_weights = {'genre': .3, 'director': .2}`, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit
- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of `model.save_model()` method. Note that datasets and optimizers will NOT be restored. If `None`, will initialize model as normal
- **map_location** (*str or torch.device*) – If `load_model_path` is provided, device specifying how to remap storage locations when `torch.load`-ing the state dictionary
- ****kwargs** (*keyword arguments*) – All keyword arguments will be saved to `self.hparams` by default

calculate_loss(*batch: Union[Tuple[Tuple[tensor, tensor], tensor], Tuple[tensor, tensor, tensor]]*) → tensor

Given a batch of data, calculate the loss value.

Note that the data type (implicit or explicit) will be determined by the structure of the batch sent to this method. See the table below for expected data types:

`__getitem__` Format	Expected Meaning	Model Type
`((X, Y), Z)`	`((user IDs, item IDs), negative item IDs)`	**Implicit**
`(X, Y, Z)`	`(user IDs, item IDs, ratings)`	**Explicit**

configure_optimizers() → Union[Tuple[List[Optimizer], List[Optimizer]], Tuple[Optimizer, Optimizer], Optimizer]

Configure optimizers and learning rate schedulers to use in optimization.

This method will be called after `setup`.

If `self.bias_optimizer` is None, only a single optimizer will be returned. If there is a non-`None` class attribute for `bias_optimizer`, two optimizers will be created: one for all layers with the name 'bias' in them, and another for all other model parameters. The bias optimizer will be set with the same parameters as `optimizer` with the exception of the learning rate, which will be set to `self.hparams.bias_lr`.

abstract forward(*users: tensor*, *items: tensor*) → tensor

`forward` should be implemented in all subclasses.

get_item_predictions(*user_id: int = 0*, *unseen_items_only: bool = False*, *sort_values: bool = True*) → Series

Get predicted rankings/ratings for all items for a given `user_id`.

This method cannot be called for datasets stored in `HDF5InteractionsDataLoader` since data in this `DataLoader` is read in dynamically.

Parameters

- **user_id** (*int*) –
- **unseen_items_only** (*bool*) – Filter `preds` to only show predictions of unseen items not present in the training or validation datasets for that `user_id`. Note this requires both `train_loader` and `val_loader` to be 1) class-level attributes in the model and 2) DataLoaders with `Interactions` at its core (not `HDF5Interactions`). If you are loading in a model, these two attributes will need to be set manually, since datasets are NOT saved when saving the model
- **sort_values** (*bool*) – Whether to sort recommendations by descending prediction probability or not

Returns

preds – Sorted values as predicted ratings for each item in the dataset with the index being the item ID

Return type

pd.Series

get_user_predictions(*item_id: int = 0*, *unseen_users_only: bool = False*, *sort_values: bool = True*) → Series

User counterpart to `get_item_predictions` method.

Get predicted rankings/ratings for all users for a given `item_id`.

This method cannot be called for datasets stored in `HDF5InteractionsDataLoader` since data in this `DataLoader` is read in dynamically.

Parameters

- **item_id** (*int*) –
- **unseen_users_only** (*bool*) – Filter `preds` to only show predictions of unseen users not present in the training or validation datasets for that `item_id`. Note this requires both `train_loader` and `val_loader` to be 1) class-level attributes in the model and 2) DataLoaders with `Interactions` at its core (not `HDF5Interactions`). If you are loading in a model, these two attributes will need to be set manually, since datasets are NOT saved when saving the model

- **sort_values** (*bool*) – Whether to sort recommendations by descending prediction probability

Returns
preds – Sorted values as predicted ratings for each user in the dataset with the index being the user ID

Return type
pd.Series

item_item_similarity(*item_id: int*) → Series

Get most similar item indices by cosine similarity.

Cosine similarity is computed with item embeddings from a trained model.

Parameters
item_id (*int*) –

Returns
sim_score_idxs – Sorted values as cosine similarity for each item in the dataset with the index being the item ID

Return type
pd.Series

Note: Returned array is unfiltered, so the first element, being the most similar item, will always be the item itself.

on_fit_start() → None

Method that runs at the very beginning of the fit process.

This method will be called after `configure_optimizers`.

save_model(*filename: Union[str, Path] = 'model.pth'*) → None

Save the model's state dictionary and hyperparameters.

While PyTorch Lightning offers a way to save and load models, there are two main reasons for overriding these:

1) We only want to save the underlying PyTorch model (and not the `Trainer` object) so we don't have to require PyTorch Lightning as a dependency when deploying a model.

2) In the v0.8.4 release, loading a model back in leads to a `RuntimeError` unable to load in weights.

Parameters
filepath (*str or Path*) – Filepath for state dictionary to be saved at ending in '.pth'

train_dataloader() → Union[*ApproximateNegativeSamplingInteractionsDataLoader*, *InteractionsDataLoader*]

Method that sets up training data as a PyTorch DataLoader.

This method will be called after `on_fit_start`.

training_epoch_end(*outputs: Union[List[float], List[List[float]]]*) → None

Method that contains a callback for logic to run after the training epoch ends.

This method will be called after `training_step`.

`training_step`(*batch: Tuple[Tuple[tensor, tensor], tensor]*, *batch_idx: int*, *optimizer_idx: Optional[int] = None*) → tensor

Method that contains logic for what happens inside the training loop.

This method will be called after `train_dataloader`.

`user_user_similarity`(*user_id: int*) → Series

User counterpart to `item_item_similarity` method.

Get most similar user indices by cosine similarity.

Cosine similarity is computed with user embeddings from a trained model.

Parameters
`user_id` (*int*) –

Returns
sim_score_idxs – Sorted values as cosine similarity for each user in the dataset with the index being the user ID

Return type
pd.Series

Note: Returned array is unfiltered, so the first element, being the most similar user, will always be the seed user themself.

`val_dataloader`() → Union[*ApproximateNegativeSamplingInteractionsDataLoader*, *InteractionsDataLoader*]

Method that sets up validation data as a PyTorch DataLoader.

This method will be called after `training_step`.

`validation_epoch_end`(*outputs: List[float]*) → None

Method that contains a callback for logic to run after the validation epoch ends.

This method will be called after `validation_step`.

`validation_step`(*batch: Tuple[Tuple[tensor, tensor], tensor]*, *batch_idx: int*, *optimizer_idx: Optional[int] = None*) → tensor

Method that contains logic for what happens inside the validation loop.

This method will be called after `val_dataloader`.

4.5.2 Base Collie Multi-Stage Pipeline Template

class `collie.model.MultiStagePipeline`(*train: Optional[Union[*ApproximateNegativeSamplingInteractionsDataLoader*, *Interactions*, *InteractionsDataLoader*]] = None*, *val: Optional[Union[*ApproximateNegativeSamplingInteractionsDataLoader*, *Interactions*, *InteractionsDataLoader*]] = None*, *lr_scheduler_func: Optional[_LRScheduler] = None*, *weight_decay: float = 0.0*, *optimizer_config_list: Optional[List[Dict[str, Union[float, List[str], str]]]] = None*, *loss: Union[str, Callable[[...], tensor]] = 'hinge'*, *metadata_for_loss: Optional[Dict[str, tensor]] = None*, *metadata_for_loss_weights: Optional[Dict[str, float]] = None*, *load_model_path: Optional[str] = None*, *map_location: Optional[str] = None*, ***kwargs*)

Bases: *BasePipeline*

Multi-stage pipeline model architectures to inherit from.

This model template is intended for models that train in distinct stages, with a different optimizer optimizing each step. This allows model components to be optimized with a set order in mind, rather than all at once, such as with the `BasePipeline`.

Generally, multi-stage models will have a training protocol like:

```
from collie.model import CollieTrainer, SomeMultiStageModel

model = SomeMultiStageModel(train=train)
trainer = CollieTrainer(model)

# fit stage 1
trainer.fit(model)

# fit stage 2
trainer.max_epochs += 10
model.advance_stage()
trainer.fit(model)

# fit stage 3
trainer.max_epochs += 10
model.advance_stage()
trainer.fit(model)

# ... and so on, until...

model.eval()
```

Just like with `BasePipeline`, all subclasses MUST at least override the following methods:

- `_setup_model` - Set up the model architecture
- `forward` - Forward pass through a model

For `item_item_similarity` to work properly, all subclasses are should also implement:

- `_get_item_embeddings` - Returns item embeddings from the model

Parameters

- **train** (`collie.interactions` object) – Data loader for training data. If an `Interactions` object is supplied, an `InteractionsDataLoader` will automatically be instantiated with `shuffle=True`
- **val** (`collie.interactions` object) – Data loader for validation data. If an `Interactions` object is supplied, an `InteractionsDataLoader` will automatically be instantiated with `shuffle=False`
- **lr_scheduler_func** (*torch.optim.lr_scheduler*) – Learning rate scheduler to use during fitting
- **weight_decay** (*float*) – Weight decay passed to the optimizer, if optimizer permits
- **optimizer_config_list** (*list of dict*) – List of dictionaries containing the optimizer configurations for each stage's optimizer(s). Each dictionary must contain the following keys:

  - **lr: str**
    Learning rate for the optimizer
  - optimizer: torch.optim or str
  - **parameter_prefix_list: List[str]**
    List of string prefixes corressponding to the model components that should be optimized with this optimizer
  - **stage: str**
    Name of stage

  This must be ordered with the intended progression of stages.

- **loss** (*function or str*) – If a string, one of the following implemented losses:
  - 'bpr' / 'adaptive_bpr' (implicit data)
  - 'hinge' / 'adaptive_hinge' (implicit data)
  - 'warp' (implicit data)
  - 'mse' (explicit data)
  - 'mae' (explicit data)

  For implicit data, if train.num_negative_samples > 1, the adaptive loss version will automatically be used of the losses above (except for WARP loss, which is only adaptive by nature).

  If a callable is passed, that function will be used for calculating the loss. For implicit models, the first two arguments passed will be the positive and negative predictions, respectively. Additional keyword arguments passed in order are num_items, positive_items, negative_items, metadata, and metadata_weights. For explicit models, the only two arguments passed in will be the prediction and actual rating values, in order.

- **metadata_for_loss** (*dict*) – Keys should be strings identifying each metadata type that match keys in metadata_weights. Values should be a torch.tensor of shape (num_items x 1). Each tensor should contain categorical metadata information about items (e.g. a number representing the genre of the item)

- **metadata_for_loss_weights** (*dict*) – Keys should be strings identifying each metadata type that match keys in metadata. Values should be the amount of weight to place on a match of that type of metadata, with the sum of all values <= 1. e.g. If metadata_for_loss_weights = {'genre': .3, 'director': .2}, then an item is:
  - a 100% match if it's the same item,
  - a 50% match if it's a different item with the same genre and same director,
  - a 30% match if it's a different item with the same genre and different director,
  - a 20% match if it's a different item with a different genre and same director,
  - a 0% match if it's a different item with a different genre and different director, which is equivalent to the loss without any partial credit

- **load_model_path** (*str or Path*) – To load a previously-saved model for inference, pass in path to output of model.save_model() method. Note that datasets and optimizers will NOT be restored. If None, will initialize model as normal

- **map_location** (*str or torch.device*) – If load_model_path is provided, device specifying how to remap storage locations when torch.load-ing the state dictionary

- **`**kwargs`** (*`keyword arguments`*) – All keyword arguments will be saved to `self.hparams` by default

Notes

- With each call of `trainer.fit`, the optimizer and learning rate scheduler state will reset.
- When loading a multi-stage model in, the state will be set to the last possible state. This state may have a different `forward` calculation than other states.

`advance_stage``()` → None

Advance the stage to the next one in `self.hparams.stage_list`.

`configure_optimizers``()` → Union[Tuple[List[Optimizer], List[Optimizer]], Tuple[Optimizer, Optimizer], Optimizer]

Configure optimizers and learning rate schedulers to use in optimization.

This method will be called after *setup*.

Creates an optimizer and learning rate scheduler for each configuration dictionary in `self.hparams.optimizer_config_list`.

`optimizer_step``(`*epoch: Optional[int] = None*, *batch_idx: Optional[int] = None*, *optimizer: Optional[Optimizer] = None*, *optimizer_idx: Optional[int] = None*, *optimizer_closure: Optional[Callable[[...], Any]] = None*, ***kwargs*`)` → None

Overriding Lightning's optimizer step function to only step the optimizer associated with the relevant stage.

See here for more details: https://pytorch-lightning.readthedocs.io/en/stable/common/lightning_module.html#optimizer-step

Parameters

- **`epoch`** (*`int`*) – Current epoch
- **`batch_idx`** (*`int`*) – Index of current batch
- **`optimizer`** (*`torch.optim.Optimizer`*) – A PyTorch optimizer
- **`optimizer_idx`** (*`int`*) – If you used multiple optimizers, this indexes into that list
- **`optimizer_closure`** (*`Callable`*) – Closure for all optimizers

`set_stage``(`*stage: str*`)` → None

Set the model to the desired stage.

4.6 Layers

4.6.1 Scaled Embedding

`class collie.model.`**`ScaledEmbedding`**`(`*num_embeddings: int*, *embedding_dim: int*, *padding_idx: Optional[int] = None*, *max_norm: Optional[float] = None*, *norm_type: float = 2.0*, *scale_grad_by_freq: bool = False*, *sparse: bool = False*, *_weight: Optional[Tensor] = None*, *device=None*, *dtype=None*`)`

Bases: `Embedding`

Embedding layer that initializes its values to use a truncated normal distribution.

reset_parameters() → None

Overriding default `reset_parameters` method.

4.6.2 Zero Embedding

class collie.model.**ZeroEmbedding**(*num_embeddings: int*, *embedding_dim: int*, *padding_idx: Optional[int] = None*, *max_norm: Optional[float] = None*, *norm_type: float = 2.0*, *scale_grad_by_freq: bool = False*, *sparse: bool = False*, *_weight: Optional[Tensor] = None*, *device=None*, *dtype=None*)

Bases: `Embedding`

Embedding layer with weights zeroed-out.

reset_parameters() → None

Overriding default `reset_parameters` method.

CHAPTER

FIVE

EVALUATION METRICS

The Collie library supports evaluating both implicit and explicit models.

Three common implicit recommendation evaluation metrics come out-of-the-box with Collie. These include Area Under the ROC Curve (AUC), Mean Reciprocal Rank (MRR), and Mean Average Precision at K (MAP@K). Each metric is optimized to be as efficient as possible by having all calculations done in batch, tensor form on the GPU (if available). We provide a standard helper function, `evaluate_in_batches`, to evaluate a model on many metrics in a single pass.

Explicit evaluation of recommendation systems is luckily much more straightforward, allowing us to utilize the TorchMetrics library for flexible, optimized metric calculations on the GPU accessed through a standard helper function, `explicit_evaluate_in_batches`, whose API is very similar to its implicit counterpart.

5.1 Evaluate in Batches

5.1.1 Implicit Evaluate in Batches

`collie.metrics.`**`evaluate_in_batches`**(*metric_list: Iterable[Callable[[csr_matrix, Union[array, tensor], Union[array, tensor], Optional[int]], float]]*, *test_interactions:* Interactions, *model:* BasePipeline, *k: int = 10*, *batch_size: int = 20*, *logger: Optional[Logger] = None*, *verbose: bool = True*) → List[float]

Evaluate a model with potentially several different metrics.

Memory constraints require that most test sets will need to be evaluated in batches. This function handles the looping and batching boilerplate needed to properly evaluate the model without running out of memory.

Parameters

- **`metric_list`** (*`list of functions`*) – List of evaluation functions to apply. Each function must accept keyword arguments:
  - `targets`
  - `user_ids`
  - `preds`
  - `k`
- **`test_interactions`** (`collie.interactions.Interactions`) – Interactions to use as labels
- **`model`** (`collie.model.BasePipeline`) – Model that can take a (user_id, item_id) pair as input and return a recommendation score
- **`k`** (*`int`*) – Number of recommendations to consider per user. This is ignored by some metrics

- **batch_size** (*int*) – Number of users to score in a single batch. For best efficiency, this number should be as high as possible without running out of memory
- **logger** (*pytorch_lightning.loggers.base.LightningLoggerBase*) – If provided, will log outputted metrics dictionary using the `log_metrics` method with keys being the string representation of `metric_list` and values being `evaluation_results`. Additionally, if `model.hparams.num_epochs_completed` exists, this will be logged as well, making it possible to track metrics progress over the course of model training
- **verbose** (*bool*) – Display progress bar and print statements during function execution

Returns

evaluation_results – List of floats, with each metric value corresponding to the respective function passed in `metric_list`

Return type

list

Examples

```
from collie.metrics import auc, evaluate_in_batches, mapk, mrr

map_10_score, mrr_score, auc_score = evaluate_in_batches(
    metric_list=[mapk, mrr, auc],
    test_interactions=test,
    model=model,
)

print(map_10_score, mrr_score, auc_score)
```

5.1.2 Explicit Evaluate in Batches

`collie.metrics.`**`explicit_evaluate_in_batches`**(*metric_list: Iterable[Metric]*, *test_interactions:* ExplicitInteractions, *model:* BasePipeline, *logger: Optional[Logger] = None*, *verbose: bool = True*, ***kwargs*) → List[float]

Evaluate a model with potentially several different metrics.

Memory constraints require that most test sets will need to be evaluated in batches. This function handles the looping and batching boilerplate needed to properly evaluate the model without running out of memory.

Parameters

- **metric_list** (list of `torchmetrics.Metric`) – List of evaluation functions to apply. Each function must accept arguments for predictions and targets, in order
- **test_interactions** (`collie.interactions.ExplicitInteractions`) –
- **model** (`collie.model.BasePipeline`) – Model that can take a (user_id, item_id) pair as input and return a recommendation score
- **batch_size** (*int*) – Number of users to score in a single batch. For best efficiency, this number should be as high as possible without running out of memory
- **logger** (*pytorch_lightning.loggers.base.LightningLoggerBase*) – If provided, will log outputted metrics dictionary using the `log_metrics` method with keys being the

string representation of `metric_list` and values being `evaluation_results`. Additionally, if `model.hparams.num_epochs_completed` exists, this will be logged as well, making it possible to track metrics progress over the course of model training

- **verbose** (*bool*) – Display progress bar and print statements during function execution
- ****kwargs** (*keyword arguments*) – Additional arguments sent to the `InteractionsDataLoader`

Returns

evaluation_results – List of floats, with each metric value corresponding to the respective function passed in `metric_list`

Return type

list

Examples

```
import torchmetrics

from collie.metrics import explicit_evaluate_in_batches

mse_score, mae_score = evaluate_in_batches(
    metric_list=[torchmetrics.MeanSquaredError(), torchmetrics.MeanAbsoluteError()],
    test_interactions=test,
    model=model,
)

print(mse_score, mae_score)
```

5.2 Implicit Metrics

5.2.1 AUC

`collie.metrics.`**`auc`**(*targets: csr_matrix*, *user_ids: Union[array, tensor]*, *preds: Union[array, tensor]*, *k: Optional[Any] = None*) → float

Calculate the area under the ROC curve (AUC) for each user and average the results.

Parameters

- **targets** (*scipy.sparse.csr_matrix*) – Interaction matrix containing user and item IDs
- **user_ids** (*np.array or torch.tensor*) – Users corresponding to the recommendations in the top k predictions
- **preds** (*torch.tensor*) – Tensor of shape (n_users x n_items) with each user's scores for each item
- **k** (*Any*) – Ignored, included only for compatibility with `mapk`

Returns

auc_score

Return type

float

5.2.2 MAP@K

`collie.metrics.`**`mapk`**`(`*targets: csr_matrix*, *user_ids: Union[array, tensor]*, *preds: Union[array, tensor]*, *k: int = 10*`)` → float

Calculate the mean average precision at K (MAP@K) score for each user.

Parameters

- **`targets`** (*`scipy.sparse.csr_matrix`*) – Interaction matrix containing user and item IDs
- **`user_ids`** (*`np.array or torch.tensor`*) – Users corresponding to the recommendations in the top k predictions
- **`preds`** (*`torch.tensor`*) – Tensor of shape (n_users x n_items) with each user's scores for each item
- **`k`** (*`int`*) – Number of recommendations to consider per user

Returns
mapk_score

Return type
float

5.2.3 MRR

`collie.metrics.`**`mrr`**`(`*targets: csr_matrix*, *user_ids: Union[array, tensor]*, *preds: Union[array, tensor]*, *k: Optional[Any] = None*`)` → float

Calculate the mean reciprocal rank (MRR) of the input predictions.

Parameters

- **`targets`** (*`scipy.sparse.csr_matrix`*) – Interaction matrix containing user and item IDs
- **`user_ids`** (*`np.array or torch.tensor`*) – Users corresponding to the recommendations in the top k predictions
- **`preds`** (*`torch.tensor`*) – Tensor of shape (n_users x n_items) with each user's scores for each item
- **`k`** (*`Any`*) – Ignored, included only for compatibility with `mapk`

Returns
mrr_score

Return type
float

CHAPTER

SIX

UTILITY FUNCTIONS

6.1 Create Ratings Matrix

`class collie.utils.create_ratings_matrix(df: DataFrame, user_col: str = 'user_id', item_col: str = 'item_id', ratings_col: str = 'rating', sparse: bool = False)`

Helper function to convert a Pandas DataFrame to 2-dimensional matrix.

Parameters

- **df** (*pd.DataFrame*) – Dataframe with columns for user IDs, item IDs, and ratings
- **user_col** (*str*) – Column name for the user IDs
- **item_col** (*str*) – Column name for the item IDs
- **ratings_col** (*str*) – Column name for the ratings column
- **sparse** (*bool*) – Whether to return data as a sparse `coo_matrix` (True) or np.array (False)

Returns

ratings_matrix – Data with users as rows, items as columns, and ratings as values

Return type

np.array or scipy.sparse.coo_matrix, 2-d

6.2 DataFrame to `Interactions`

`class collie.utils.df_to_interactions(df: DataFrame, user_col: str = 'user_id', item_col: str = 'item_id', ratings_col: Optional[str] = 'rating', **kwargs)`

Helper function to convert a DataFrame to an `Interactions` object.

Parameters

- **df** (*pd.DataFrame*) – Dataframe with columns for user IDs, item IDs, and (optionally) ratings
- **user_col** (*str*) – Column name for the user IDs
- **item_col** (*str*) – Column name for the item IDs
- **ratings_col** (*str*) – Column name for the ratings column. If `None`, will default to ratings of all 1s
- ****kwargs** – Keyword arguments to pass to `Interactions`

Returns
interactions

Return type
collie.interactions.Interactions

6.3 Convert to Implicit Ratings

class collie.utils.**convert_to_implicit**(*explicit_df: DataFrame*, *min_rating_to_keep: Optional[float] = 4*, *user_col: str = 'user_id'*, *item_col: str = 'item_id'*, *ratings_col: str = 'rating'*)

Convert explicit interactions data to implicit data.

Duplicate user ID and item ID pairs will be dropped, as well as all scores that are < `min_rating_to_keep`. All remaining interactions will have a rating of `1`.

Parameters

- **explicit_df** (*pd.DataFrame*) – Dataframe with explicit ratings in the rating column
- **min_rating_to_keep** (*int*) – Minimum rating to be considered a valid interaction
- **ratings_col** (*str*) – Column name for the ratings column

Returns
implicit_df – Dataframe that converts all `ratings >= min_rating_to_keep` to 1 and drops the rest with a reset index. Note that the order of `implicit_df` will not be equal to `explicit_df`

Return type
pd.DataFrame

6.4 Remove Users With Fewer Than *n* Interactions

class collie.utils.**remove_users_with_fewer_than_n_interactions**(*df: DataFrame*, *min_num_of_interactions: int = 3*, *user_col: str = 'user_id'*)

Remove DataFrame rows with users who appear fewer than `min_num_of_interactions` times.

Parameters

- **df** (*pd.DataFrame*) –
- **min_num_of_interactions** (*int*) – Minimum number of interactions a user can have while remaining in `filtered_df`
- **user_col** (*str*) – Column name for the user IDs

Returns
filtered_df

Return type
pd.DataFrame

6.5 Pandas DataFrame to HDF5 Format

class collie.utils.**pandas_df_to_hdf5**(*df: DataFrame*, *out_path: Union[str, Path]*, *key: str = 'interactions'*)

Append a Pandas DataFrame to HDF5 using a `table` format and `blosc` compression.

6.6 DataFrame to HTML

class collie.utils.**df_to_html**(*df: DataFrame*, *image_cols: List[str] = []*, *hyperlink_cols: List[str] = []*, *html_tags: Dict[str, Union[str, List[str]]] = {}*, *transpose: bool = False*, *image_width: Optional[int] = None*, *max_num_rows: int = 200*, ***kwargs*)

Convert a Pandas DataFrame to HTML.

Parameters

- **df** (*DataFrame*) – DataFrame to convert to HTML
- **image_cols** (*str or list*) – Column names that contain image urls or file paths. Columns specified as images will make all other transformations to those columns be ignored. Local files will display correctly in Jupyter if specified using relative paths but not if specified using absolute paths (see https://github.com/jupyter/notebook/issues/3810).
- **hyperlink_cols** (*str or list*) – Column names that contain hyperlinks to open in a new tab
- **html_tags** (*dictionary*) – A transformation to be inserted directly into the HTML tag.

  Ex: `{'col_name_1': 'strong'}` becomes <strong>`col_name_1`</strong>

  Ex: `{'col_name_2': 'mark'}` becomes <mark>`col_name_2`</mark>

  Ex: `{'col_name_3': 'h2'}` becomes <h2>`col_name_3`</h2>

  Ex: `{'col_name_4': ['em', 'strong']}` becomes <em><strong>`col_name_4`</strong></em>
- **transpose** (*bool*) – Transpose the DataFrame before converting to HTML
- **image_width** (*int*) – Set image width for each image generated
- **max_num_rows** (*int*) – Maximum number of rows to display
- ****kwargs** (*keyword arguments*) – Additional arguments sent to `pandas.DataFrame.to_html`, as listed in: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_html.html

Returns

df_html – DataFrame converted to a HTML string, ready for displaying

Return type

HTML

Examples

In a Jupyter notebook:

```
from IPython.core.display import display, HTML
import pandas as pd

df = pd.DataFrame({
    'item': ['Beefy Fritos® Burrito'],
    'price': ['1.00'],
    'image_url': ['https://www.tacobell.com/images/22480_beefy_fritos_burrito_
↪269x269.jpg'],
})
display(
    HTML(
        df_to_html(
            df,
            image_cols='image_url',
            html_tags={'item': 'strong', 'price': 'em'},
            image_width=200,
        )
    )
)
```

Note: Converted table will have CSS class 'dataframe', unless otherwise specified.

6.7 Timer Class

class collie.utils.**Timer**

Class to manage timing different sections of a job.

time_since_start(*message: str = 'Total time'*) → float

Get time since timer was instantiated.

timecheck(*message: str = 'Finished'*) → float

Get time since last timecheck.

6.8 Truncated Normal Initialization

class collie.utils.**trunc_normal**(*embedding_weight: tensor*, *mean: float = 0.0*, *std: float = 1.0*)

Truncated normal initialization (approximation).

Taken from FastAI: https://github.com/fastai/fastai/blob/master/fastai/layers.py

CHAPTER

SEVEN

MOVIELENS FUNCTIONS

The following functions under `collie.movielens` read and prepare MovieLens 100K data, train and evaluate a model on this data, and visualize recommendation results.

7.1 Get MovieLens 100K Data

7.1.1 Read MovieLens 100K Interactions Data

`collie.movielens.`**`read_movielens_df`**`(`*decrement_ids: bool = True*`)` → DataFrame

Read `u.data` from the MovieLens 100K dataset.

If there is not a directory at `$DATA_PATH/ml-100k`, this function creates that directory and downloads the entire dataset there.

See the MovieLens 100K README for additional information on the dataset: https://files.grouplens.org/datasets/movielens/ml-100k-README.txt

Parameters

`decrement_ids` (*`bool`*) – Decrement user and item IDs by 1 before returning, which is required for Collie's `Interactions` dataset

Returns

df –

MovieLens 100K `u.data` comprising of columns:

- user_id
- item_id
- rating
- timestamp

Return type

pd.DataFrame

Side Effects

Creates directory at `$DATA_PATH/ml-100k` and downloads data files if data does not exist.

7.1.2 Read MovieLens 100K Item Data

`collie.movielens.`**`read_movielens_df_item`**`()` → DataFrame

Read `u.item` from the MovieLens 100K dataset.

If there is not a directory at `$DATA_PATH/ml-100k`, this function creates that directory and downloads the entire dataset there.

See the MovieLens 100K README for additional information on the dataset: https://files.grouplens.org/datasets/movielens/ml-100k-README.txt

Returns

df_item –

MovieLens 100K `u.item` containing columns:

- item_id
- movie_title
- release_date
- video_release_date
- IMDb_URL
- unknown
- Action
- Adventure
- Animation
- Children
- Comedy
- Crime
- Documentary
- Drama
- Fantasy
- Film_Noir
- Horror
- Musical
- Mystery
- Romance', 'Sci_Fi
- Thriller
- War
- Wester

Return type
pd.DataFrame

Side Effects

Creates directory at `$DATA_PATH/ml-100k` and downloads data files if data does not exist.

7.1.3 Read MovieLens 100K Posters Data

`collie.movielens.`**`read_movielens_posters_df`**`()` → DataFrame

Read in data containing the item ID and poster URL for visualization purposes of MovieLens 100K data.

This function will attempt to read the file at `data/movielens_posters.csv` if it exists and, if not, will read the CSV from the origin GitHub repo at https://raw.githubusercontent.com/ShopRunner/collie/main/data/movielens_posters.csv.

Returns

posters_df –

DataFrame comprising columns:

- item_id
- url

Return type
pd.DataFrame

7.1.4 Format MovieLens 100K Item Metadata Data

`collie.movielens.`**`get_movielens_metadata`**`(`*df_item: Optional[DataFrame] = None*`)` → DataFrame

Return MovieLens 100K metadata as a DataFrame.

DataFrame returned has the following column order:

```
[
    'genre_action', 'genre_adventure', 'genre_animation', 'genre_children', 'genre_
↪comedy',
    'genre_crime', 'genre_documentary', 'genre_drama', 'genre_fantasy', 'genre_film_
↪noir',
    'genre_horror', 'genre_musical', 'genre_mystery', 'genre_romance', 'genre_sci_fi
↪',
    'genre_thriller', 'genre_war', 'genre_western', 'genre_unknown', 'decade_unknown
↪',
    'decade_20', 'decade_30', 'decade_40', 'decade_50', 'decade_60',
    'decade_70', 'decade_80', 'decade_90',
]
```

See the MovieLens 100K README for additional information on the dataset: https://files.grouplens.org/datasets/movielens/ml-100k-README.txt

Parameters
`df_item` (*`pd.DataFrame`*) – DataFrame of MovieLens 100K `u.item` containing binary columns of movie names and metadata. If `None`, will automatically read the output of `read_movielens_df_item()`

Returns
metadata_df

Return type
pd.DataFrame

7.1.5 Format MovieLens 100K User Metadata Data

`collie.movielens.`**`get_user_metadata`**`(`*df_user: Optional[DataFrame] = None*`)` → DataFrame

Return MovieLens 100K user metadata as a DataFrame.

DataFrame returned has the following column order:

```
[
    'age', 'gender', 'occupation_administrator', 'occupation_artist'
    'occupation_doctor', 'occupation_educator', 'occupation_engineer'
    'occupation_entertainment', 'occupation_executive'
    'occupation_healthcare', 'occupation_homemaker'
    'occupation_lawyer', 'occupation_librarian', 'occupation_marketing'
    'occupation_none', 'occupation_other', 'occupation_programmer'
    'occupation_retired', 'occupation_salesman', 'occupation_scientist'
    'occupation_student', 'occupation_technician', 'occupation_writer',
]
```

See the MovieLens 100K README for additional information on the dataset: https://files.grouplens.org/datasets/movielens/ml-100k-README.txt

Parameters
df_user (*pd.DataFrame*) – DataFrame of MovieLens 100K `u.user` containing columns of user metadata. If `None`, will automatically read the output of `read_movielens_df_user()`

Returns
metadata_df

Return type
pd.DataFrame

7.2 MovieLens Model Training Pipeline

`collie.movielens.`**`run_movielens_example`**`(`*epochs: int = 20*, *gpus: int = 0*`)` → None

Retrieve and split data, train and evaluate a model, and save it.

From the terminal, you can run this script with:

```
python collie/movielens/run.py  --epochs 20
```

Parameters

- **epochs** (*int*) – Number of epochs for model training
- **gpus** (*int*) – Number of gpus to train on

7.3 Visualize MovieLens Predictions

`collie.movielens.`**`get_recommendation_visualizations`**(*model: BasePipeline*, *user_id: int*, *df_user: Optional[DataFrame] = None*, *df_item: Optional[DataFrame] = None*, *movielens_posters_df: Optional[DataFrame] = None*, *num_user_movies_to_display: int = 10*, *num_similar_movies: int = 10*, *filter_films: bool = True*, *shuffle: bool = True*, *detailed: bool = False*, *image_width: int = 500*) → str

Visualize Movielens 100K recommendations for a given user.

Parameters

- **model** (`collie.model.BasePipeline`) –
- **user_id** (*int*) – User ID to retrieve recommendations for
- **df_user** (*DataFrame*) – `u.data` from MovieLens data. This DataFrame must have columns:
  - `user_id` (starting at 1)
  - `item_id` (starting at 1)
  - `rating` (explicit ratings)

  If `None`, will set to the output of `read_movielens_df(decrement_ids=False)`.
- **df_item** (*DataFrame*) – `u.item` from MovieLens data. This DataFrame must have columns:
  - `item_id` (starting at 1)
  - `movie_title`

  If `None`, will set to the output of `read_movielens_df_item()`
- **movielens_posters_df** (*DataFrame*) – DataFrame containing item_ids from MovieLens data and the poster url. This DataFrame must have columns:
  - `item_id` (starting at 1)
  - `url`

  If `None`, will set to the output of `read_movielens_posters_df()`
- **num_user_movies_to_display** (*int*) – Number of movies rated `4` or `5` to display for the user
- **num_similar_movies** (*int*) – Number of movies recommendations to display
- **filter_films** (*bool*) – Filter films out of recommendations if the user has already interacted with them
- **shuffle** (*bool*) – Shuffle order of `num_user_movies_to_display` films
- **detailed** (*bool*) – Of the top `N` unfiltered recommendations, display how many movies the user gave a positive and negative rating to
- **image_width** (*int*) – Image width for HTML images

Returns

html – HTML string of movies a user loved and the model recommended for a given user, ready for displaying

Return type

str

CHAPTER

EIGHT

TUTORIALS

Tutorial	Colab Link
Tutorial 01 - Prepare Data	
Tutorial 02 - Matrix Factorization Model	
Tutorial 03 - Advanced Matrix Factorization Model	
Tutorial 04 - Partial Credit Loss - Matrix Factorization Model	
Tutorial 05 - Hybrid Pretrained Matrix Factorization Model	
Tutorial 06 - Multi-Stage Models	
Tutorial 07 - Explicit Data Support in Collie	

CHAPTER

NINE

CONTRIBUTING AND MAKING PRS

9.1 How to Contribute

We welcome contributions in the form of issues or pull requests!

We want this to be a place where all are welcome to discuss and contribute, so please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms. Find the Code of Conduct in the `CONDUCT.md` file on GitHub or in the Code of Conduct section of read the docs.

If you have a problem using Collie or see a possible improvement, open an issue in the GitHub issue tracker. Please be as specific as you can.

If you see an open issue you'd like to be fixed, take a stab at it and open a PR!

9.2 Pull Requests

To create a PR against this library, please fork the project and work from there.

9.2.1 Steps

1. Fork the project via the `Fork` button on GitHub.
2. Clone the repo to your local disk.
3. Create a new branch for your PR.

   ```
   git checkout -b my-awesome-new-feature
   ```

4. Install requirements (either in a virtual environment like below or the Docker container).

   ```
   virtualenv venv
   source venv/bin/activate
   pip install -r requirements-dev.txt
   pip install -r requirements.txt
   ```

5. Develop your feature
6. Submit a PR to `main`! Someone will review your code and merge your code into `main` when it is approved.

9.2.2 PR Checklist

- Ensure your code has followed the Style Guidelines listed below.
- Run the `flake8` linter on your code.

```
source venv/bin/activate
flake8 collie tests
```

- Make sure you have written unit-tests where appropriate.
- Make sure the unit-tests all pass.

```
source venv/bin/activate
pytest -v
```

- Update the docs where appropriate. You can rebuild them with the commands below.

```
cd docs
make html
```

- Update the `CHANGELOG.md` and `version.py` files.

9.2.3 Style Guidelines

For the most part, this library follows PEP8 with a couple of exceptions.

- Lines can be up to 100 characters long.
- Docstrings should be numpy style docstrings.
- Your code should be Python 3 compatible.
- We prefer single quotes for one-line strings unless using double quotes allows us to avoid escaping internal single quotes.
- When in doubt, follow the style of the existing code.

CHAPTER

TEN

CONTRIBUTOR COVENANT CODE OF CONDUCT

10.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

10.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

10.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

10.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

10.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement by submitting this anonymous form or by sending an email to opensource@shoprunner.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

10.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

10.6.1 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

10.6.2 2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

10.6.3 3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

10.6.4 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the project community.

10.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

CHAPTER ELEVEN

COLLIE

Collie is a library for preparing, training, and evaluating implicit deep learning hybrid recommender systems, named after the Border Collie dog breed.

Collie offers a collection of simple APIs for preparing and splitting datasets, incorporating item metadata directly into a model architecture or loss, efficiently evaluating a model's performance on the GPU, and so much more. Above all else though, Collie is built with flexibility and customization in mind, allowing for faster prototyping and experimentation.

See the documentation for more details.

> "We adopted 2 Border Collies a year ago and they are about 3 years old. They are completely obsessed with fetch and tennis balls and it's getting out of hand. They live in the fenced back yard and when anyone goes out there they instantly run around frantically looking for a tennis ball. If there is no ball they will just keep looking and will not let you pet them. When you do have a ball, they are 100% focused on it and will not notice anything else going on around them, like it's their whole world."
>
> —*A Reddit thread on r/DogTraining*

11.1 Installation

```
pip install collie
```

Through July 2021, this library used to be under the name `collie_recs`. While this version is still available on PyPI, it is no longer supported or maintained. All users of the library should use `collie` for the latest and greatest version of the code!

11.2 Quick Start

11.2.1 Implicit Data

Creating and evaluating a matrix factorization model with *implicit* MovieLens 100K data is simple with Collie:

```
from collie.cross_validation import stratified_split
from collie.interactions import Interactions
from collie.metrics import auc, evaluate_in_batches, mapk, mrr
from collie.model import MatrixFactorizationModel, CollieTrainer
from collie.movielens import read_movielens_df
from collie.utils import convert_to_implicit


# read in explicit MovieLens 100K data
df = read_movielens_df()

# convert the data to implicit
df_imp = convert_to_implicit(df)

# store data as ``Interactions``
interactions = Interactions(users=df_imp['user_id'],
                            items=df_imp['item_id'],
                            allow_missing_ids=True)

# perform a data split
train, val = stratified_split(interactions)

# train an implicit ``MatrixFactorization`` model
model = MatrixFactorizationModel(train=train,
                                 val=val,
                                 embedding_dim=10,
                                 lr=1e-1,
                                 loss='adaptive',
                                 optimizer='adam')
trainer = CollieTrainer(model, max_epochs=10)
trainer.fit(model)
model.eval()

# evaluate the model
auc_score, mrr_score, mapk_score = evaluate_in_batches(metric_list=[auc, mrr, mapk],
```

(continues on next page)

(continued from previous page)

```
                                                       test_interactions=val,
                                                       model=model)

print(f'AUC:          {auc_score}')
print(f'MRR:          {mrr_score}')
print(f'MAP@10:       {mapk_score}')
```

More complicated examples of implicit pipelines can be viewed for MovieLens 100K data here, in notebooks here, and documentation here.

11.2.2 Explicit Data

Collie also handles the situation when you instead have *explicit* data, such as star ratings. Note how similar the pipeline and APIs are compared to the implicit example above:

```
from collie.cross_validation import stratified_split
from collie.interactions import ExplicitInteractions
from collie.metrics import explicit_evaluate_in_batches
from collie.model import MatrixFactorizationModel, CollieTrainer
from collie.movielens import read_movielens_df

from torchmetrics import MeanAbsoluteError, MeanSquaredError


# read in explicit MovieLens 100K data
df = read_movielens_df()

# store data as ``Interactions``
interactions = ExplicitInteractions(users=df['user_id'],
                                    items=df['item_id'],
                                    ratings=df['rating'])

# perform a data split
train, val = stratified_split(interactions)

# train an implicit ``MatrixFactorization`` model
model = MatrixFactorizationModel(train=train,
                                 val=val,
                                 embedding_dim=10,
                                 lr=1e-2,
                                 loss='mse',
                                 optimizer='adam')
trainer = CollieTrainer(model, max_epochs=10)
trainer.fit(model)
model.eval()

# evaluate the model
mae_score, mse_score = explicit_evaluate_in_batches(metric_list=[MeanAbsoluteError(),
                                                                 MeanSquaredError()],
                                                    test_interactions=val,
```

(continues on next page)

(continued from previous page)

```
                                            model=model)

print(f'MAE: {mae_score}')
print(f'MSE: {mse_score}')
```

11.3 Comparison With Other Open-Source Recommendation Libraries

On some smaller screens, you might have to scroll right to see the full table.

Aspect Included in Library							Collie
Implicit data support for when we only know when a user interacts with an item or not, not the explicit rating the user gave the item		✓		✓	✓	✓	✓
Explicit data support for when we know the explicit rating the user gave the item	✓	✓	✓	✓	✓	✓	✓
Support for **side-data** incorporated directly into the models		✓			✓	✓	✓
Support a **flexible framework for new model architectures** and experimentation			✓	✓	✓	✓	✓
Deep learning libraries utilizing speed-ups with a GPU and able to implement new, cutting-edge deep learning algorithms			✓	✓	✓	✓	✓
Automatic support for multi-GPU training							✓
Actively supported and maintained	✓	✓	✓		✓	✓	✓
Type annotations for classes, methods, and functions						✓	✓
Scalable for larger, out-of-memory datasets						✓	✓
Includes **model zoo** with two or more model architectures implemented				✓	✓		✓
Includes **implicit loss functions** for training and **metric functions** for model evaluation		✓		✓	✓		✓
Includes **adaptive loss functions** for multiple negative examples		✓		✓			✓
Includes **loss functions with partial credit for side-data**							✓

The following table notes shows the results of an experiment training and evaluating recommendation models in some popular implicit recommendation model frameworks on a common MovieLens 10M dataset. The data was split via a 90/5/5 stratified data split. Each model was trained for a maximum of 40 epochs using an embedding dimension of 32. For each model, we used default hyperparameters (unless otherwise noted below).

Model	MAP@10 Score	Notes
Randomly initialized, untrained model	0.0001	
Logistic MF	0.0128	Using the CUDA implementation.
LightFM with BPR Loss	0.0180	
ALS	0.0189	Using the CUDA implementation.
BPR	0.0301	Using the CUDA implementation.
Spotlight	0.0376	Using adaptive hinge loss.
LightFM with WARP Loss	0.0412	
Collie `MatrixFactorizationModel`	**0.0425**	Using a separate SGD bias optimizer.

At ShopRunner, we have found Collie models outperform comparable LightFM models with up to **64% improved MAP@10 scores**.

11.4 Development

To run locally, begin by creating a data path environment variable:

```
# Define where on your local hard drive you want to store data. It is best if this
# location is not inside the repo itself. An example is below
export DATA_PATH=$HOME/data/collie
```

Run development from within the Docker container:

```
docker build -t collie .

# run the container in interactive mode, leaving port ``8888`` open for Jupyter
docker run \
    -it \
    --rm \
    -v "${DATA_PATH}:/collie/data/" \
    -v "${PWD}:/collie" \
    -p 8888:8888 \
    collie /bin/bash
```

11.4.1 Run on a GPU:

```
docker build -t collie .

# run the container in interactive mode, leaving port ``8888`` open for Jupyter
docker run \
    -it \
    --rm \
    --gpus all \
    -v "${DATA_PATH}:/collie/data/" \
    -v "${PWD}:/collie" \
    -p 8888:8888 \
    collie /bin/bash
```

11.4.2 Start JupyterLab

To run JupyterLab, start the container and execute the following:

```
jupyter lab --ip 0.0.0.0 --no-browser --allow-root
```

Connect to JupyterLab here: http://localhost:8888/lab

11.4.3 Unit Tests

Library unit tests in this repo are to be run in the Docker container:

```
# execute unit tests
pytest --cov-report term --cov=collie
```

Note that a handful of tests require the MovieLens 100K dataset to be downloaded (~5MB in size), meaning that either before or during test time, there will need to be an internet connection. This dataset only needs to be downloaded a single time for use in both unit tests and tutorials.

11.4.4 Docs

The Collie library supports Read the Docs documentation. To compile locally,

```
cd docs
make html

# open local docs
open build/html/index.html
```

INDEX