DICE Embeddings

Release 0.1.3.2

Caglar Demir

Mar 28, 2024

Contents:

DICE Embeddings[1]: Hardware-agnostic Framework for Large-scale Knowledge Graph Embeddings:

1 Dicee Manual

Version: dicee 0.1.3.2

GitHub repository: https://github.com/dice-group/dice-embeddings

Publisher and maintainer: Caglar Demir[2]

Contact: caglar.demir@upb.de

Dicee is a hardware-agnostic framework for large-scale knowledge graph embeddings.

Knowledge graph embedding research has mainly focused on learning continuous representations of knowledge graphs towards the link prediction problem. Recently developed frameworks can be effectively applied in a wide range of research-related applications. Yet, using these frameworks in real-world applications becomes more challenging as the size of the knowledge graph grows

We developed the DICE Embeddings framework (dicee) to compute embeddings for large-scale knowledge graphs in a hardware-agnostic manner. To achieve this goal, we rely on

1. **Pandas**[3] **& Co.** to use parallelism at preprocessing a large knowledge graph,
2. **PyTorch**[4] **& Co.** to learn knowledge graph embeddings via multi-CPUs, GPUs, TPUs or computing cluster, and
3. **Huggingface**[5] to ease the deployment of pre-trained models.

Why Pandas[6] **& Co. ?** A large knowledge graph can be read and preprocessed (e.g. removing literals) by pandas, modin, or polars in parallel. Through polars, a knowledge graph having more than 1 billion triples can be read in parallel fashion. Importantly, using these frameworks allow us to perform all necessary computations on a single CPU as well as a cluster of computers.

Why PyTorch[7] **& Co. ?** PyTorch is one of the most popular machine learning frameworks available at the time of writing. PytorchLightning facilitates scaling the training procedure of PyTorch without boilerplate. In our framework, we combine PyTorch[8] & PytorchLightning[9]. Users can choose the trainer class (e.g., DDP by Pytorch) to train large knowledge graph embedding models with billions of parameters. PytorchLightning allows us to use state-of-the-art model parallelism techniques (e.g. Fully Sharded Training, FairScale, or DeepSpeed) without extra effort. With our framework, practitioners can directly use PytorchLightning for model parallelism to train gigantic embedding models.

Why Hugging-face Gradio[10]**?** Deploy a pre-trained embedding model without writing a single line of code.

[1] https://github.com/dice-group/dice-embeddings
[2] https://github.com/Demirrr
[3] https://pandas.pydata.org/
[4] https://pytorch.org/
[5] https://huggingface.co/
[6] https://pandas.pydata.org/
[7] https://pytorch.org/
[8] https://pytorch.org/
[9] https://www.pytorchlightning.ai/
[10] https://huggingface.co/gradio

2 Installation

2.1 Installation from Source

```
git clone https://github.com/dice-group/dice-embeddings.git
conda create -n dice python=3.10.13 --no-default-packages && conda activate dice &&␣
↪cd dice-embeddings &&
pip3 install -e .
```

or

```
pip install dicee
```

3 Download Knowledge Graphs

```
wget https://files.dice-research.org/datasets/dice-embeddings/KGs.zip --no-check-
↪certificate && unzip KGs.zip
```

To test the Installation

```
python -m pytest -p no:warnings -x # Runs >114 tests leading to > 15 mins
python -m pytest -p no:warnings --lf # run only the last failed test
python -m pytest -p no:warnings --ff # to run the failures first and then the rest of␣
↪the tests.
```

4 Knowledge Graph Embedding Models

1. TransE, DistMult, ComplEx, ConEx, QMult, OMult, ConvO, ConvQ, Keci
2. All 44 models available in https://github.com/pykeen/pykeen#models

   For more, please refer to `examples`.

5 How to Train

To Train a KGE model (KECI) and evaluate it on the train, validation, and test sets of the UMLS benchmark dataset.

```
from dicee.executer import Execute
from dicee.config import Namespace
args = Namespace()
args.model = 'Keci'
args.scoring_technique = "KvsAll"  # 1vsAll, or AllvsAll, or NegSample
args.dataset_dir = "KGs/UMLS"
args.path_to_store_single_run = "Keci_UMLS"
args.num_epochs = 100
args.embedding_dim = 32
args.batch_size = 1024
reports = Execute(args).start()
print(reports["Train"]["MRR"]) # => 0.9912
```

(continues on next page)

(continued from previous page)

```
print(reports["Test"]["MRR"]) # => 0.8155
# See the Keci_UMLS folder embeddings and all other files
```

where the data is in the following form

```
$ head -3 KGs/UMLS/train.txt
acquired_abnormality    location_of     experimental_model_of_disease
anatomical_abnormality  manifestation_of        physiologic_function
alga    isa     entity
```

A KGE model can also be trained from the command line

```
dicee --dataset_dir "KGs/UMLS" --model Keci --eval_model "train_val_test"
```

dicee automaticaly detects available GPUs and trains a model with distributed data parallels technique. Under the hood, dicee uses lighning as a default trainer.

```
# Train a model by only using the GPU-0
CUDA_VISIBLE_DEVICES=0 dicee --dataset_dir "KGs/UMLS" --model Keci --eval_model
↪"train_val_test"
# Train a model by only using GPU-1
CUDA_VISIBLE_DEVICES=1 dicee --dataset_dir "KGs/UMLS" --model Keci --eval_model
↪"train_val_test"
NCCL_P2P_DISABLE=1 CUDA_VISIBLE_DEVICES=0,1 python dicee/scripts/run.py --trainer PL -
↪-dataset_dir "KGs/UMLS" --model Keci --eval_model "train_val_test"
```

Under the hood, dicee executes run.py script and uses lighning as a default trainer

```
# Two equivalent executions
# (1)
dicee --dataset_dir "KGs/UMLS" --model Keci --eval_model "train_val_test"
# Evaluate Keci on Train set: Evaluate Keci on Train set
# {'H@1': 0.9518788343558282, 'H@3': 0.9988496932515337, 'H@10': 1.0, 'MRR': 0.
↪9753123402351737}
# Evaluate Keci on Validation set: Evaluate Keci on Validation set
# {'H@1': 0.6932515337423313, 'H@3': 0.9041411042944786, 'H@10': 0.9754601226993865,
↪'MRR': 0.8072362996241839}
# Evaluate Keci on Test set: Evaluate Keci on Test set
# {'H@1': 0.6951588502269289, 'H@3': 0.9039334341906202, 'H@10': 0.9750378214826021,
↪'MRR': 0.8064032293278861}

# (2)
CUDA_VISIBLE_DEVICES=0,1 python dicee/scripts/run.py --trainer PL --dataset_dir "KGs/
↪UMLS" --model Keci --eval_model "train_val_test"
# Evaluate Keci on Train set: Evaluate Keci on Train set
# {'H@1': 0.9518788343558282, 'H@3': 0.9988496932515337, 'H@10': 1.0, 'MRR': 0.
↪9753123402351737}
# Evaluate Keci on Train set: Evaluate Keci on Train set
# Evaluate Keci on Validation set: Evaluate Keci on Validation set
# {'H@1': 0.6932515337423313, 'H@3': 0.9041411042944786, 'H@10': 0.9754601226993865,
↪'MRR': 0.8072362996241839}
# Evaluate Keci on Test set: Evaluate Keci on Test set
# {'H@1': 0.6951588502269289, 'H@3': 0.9039334341906202, 'H@10': 0.9750378214826021,
↪'MRR': 0.8064032293278861}
```

Similarly, models can be easily trained with torchrun

```
torchrun --standalone --nnodes=1 --nproc_per_node=gpu dicee/scripts/run.py --trainer␣
↪torchDDP --dataset_dir "KGs/UMLS" --model Keci --eval_model "train_val_test"
# Evaluate Keci on Train set: Evaluate Keci on Train set: Evaluate Keci on Train set
# {'H@1': 0.9518788343558282, 'H@3': 0.9988496932515337, 'H@10': 1.0, 'MRR': 0.
↪9753123402351737}
# Evaluate Keci on Validation set: Evaluate Keci on Validation set
# {'H@1': 0.6932515337423313, 'H@3': 0.9041411042944786, 'H@10': 0.9754601226993865,
↪'MRR': 0.8072499937521418}
# Evaluate Keci on Test set: Evaluate Keci on Test set
{'H@1': 0.6951588502269289, 'H@3': 0.9039334341906202, 'H@10': 0.9750378214826021,
↪'MRR': 0.8064032293278861}
```

You can also train a model in multi-node multi-gpu setting.

```
torchrun --nnodes 2 --nproc_per_node=gpu  --node_rank 0 --rdzv_id 455 --rdzv_backend␣
↪c10d --rdzv_endpoint=nebula  dicee/scripts/run.py --trainer torchDDP --dataset_dir␣
↪KGs/UMLS
torchrun --nnodes 2 --nproc_per_node=gpu  --node_rank 1 --rdzv_id 455 --rdzv_backend␣
↪c10d --rdzv_endpoint=nebula dicee/scripts/run.py --trainer torchDDP --dataset_dir␣
↪KGs/UMLS
```

Train a KGE model by providing the path of a single file and store all parameters under newly created directory called `KeciFamilyRun`.

```
dicee --path_single_kg "KGs/Family/family-benchmark_rich_background.owl" --model Keci␣
↪--path_to_store_single_run KeciFamilyRun --backend rdflib
```

where the data is in the following form

```
$ head -3 KGs/Family/train.txt
_:1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2002/07/owl
↪#Ontology> .
<http://www.benchmark.org/family#hasChild> <http://www.w3.org/1999/02/22-rdf-syntax-ns
↪#type> <http://www.w3.org/2002/07/owl#ObjectProperty> .
<http://www.benchmark.org/family#hasParent> <http://www.w3.org/1999/02/22-rdf-syntax-
↪ns#type> <http://www.w3.org/2002/07/owl#ObjectProperty> .
```

Apart from n-triples or standard link prediction dataset formats, we support [“owl”, “nt”, “turtle”, “rdf/xml”, “n3”]*. Moreover, a KGE model can be also trained by providing **an endpoint of a triple store**.

```
dicee --sparql_endpoint "http://localhost:3030/mutagenesis/" --model Keci
```

For more, please refer to `examples`.

6 Creating an Embedding Vector Database

6.1 Learning Embeddings

```
# Train an embedding model
dicee --dataset_dir KGs/Countries-S1 --path_to_store_single_run CountryEmbeddings --
↪model Keci --p 0 --q 1 --embedding_dim 32 --adaptive_swa
```

6.2 Loading Embeddings into Qdrant Vector Database

```
# Ensure that Qdrant available
# docker pull qdrant/qdrant && docker run -p 6333:6333 -p 6334:6334      -v $(pwd)/
↪qdrant_storage:/qdrant/storage:z      qdrant/qdrant
diceeindex --path_model "CountryEmbeddings" --collection_name "dummy" --location
↪"localhost"
```

6.3 Launching Webservice

```
diceeserve --path_model "CountryEmbeddings" --collection_name "dummy" --collection_
↪location "localhost"
```

Retrieve and Search

Get embedding of germany

```
curl -X 'GET' 'http://0.0.0.0:8000/api/get?q=germany' -H 'accept: application/json'
```

Get most similar things to europe

```
curl -X 'GET' 'http://0.0.0.0:8000/api/search?q=europe' -H 'accept: application/json'
{"result":[{"hit":"europe","score":1.0},
{"hit":"northern_europe","score":0.67126536},
{"hit":"western_europe","score":0.6010134},
{"hit":"puerto_rico","score":0.5051694},
{"hit":"southern_europe","score":0.4829831}]}
```

7 Answering Complex Queries

```
# pip install dicee
# wget https://files.dice-research.org/datasets/dice-embeddings/KGs.zip --no-check-
↪certificate & unzip KGs.zip
from dicee.executer import Execute
from dicee.config import Namespace
from dicee.knowledge_graph_embeddings import KGE
# (1) Train a KGE model
args = Namespace()
args.model = 'Keci'
args.p=0
args.q=1
args.optim = 'Adam'
args.scoring_technique = "AllvsAll"
args.path_single_kg = "KGs/Family/family-benchmark_rich_background.owl"
args.backend = "rdflib"
args.num_epochs = 200
args.batch_size = 1024
args.lr = 0.1
args.embedding_dim = 512
result = Execute(args).start()
# (2) Load the pre-trained model
```

(continues on next page)

(continued from previous page)

```
pre_trained_kge = KGE(path=result['path_experiment_folder'])
# (3) Single-hop query answering
# Query: ?E : \exist E.hasSibling(E, F9M167)
# Question: Who are the siblings of F9M167?
# Answer: [F9M157, F9F141], as (F9M167, hasSibling, F9M157) and (F9M167, hasSibling,␣
↪F9F141)
predictions = pre_trained_kge.answer_multi_hop_query(query_type="1p",
                                                     query=('http://www.benchmark.org/
↪family#F9M167',
                                                            ('http://www.benchmark.
↪org/family#hasSibling',)),
                                                     tnorm="min", k=3)
top_entities = [topk_entity for topk_entity, query_score in predictions]
assert "http://www.benchmark.org/family#F9F141" in top_entities
assert "http://www.benchmark.org/family#F9M157" in top_entities
# (2) Two-hop query answering
# Query: ?D : \exist E.Married(D, E) \land hasSibling(E, F9M167)
# Question: To whom a sibling of F9M167 is married to?
# Answer: [F9F158, F9M142] as (F9M157 #married F9F158) and (F9F141 #married F9M142)
predictions = pre_trained_kge.answer_multi_hop_query(query_type="2p",
                                                     query=("http://www.benchmark.org/
↪family#F9M167",
                                                            ("http://www.benchmark.
↪org/family#hasSibling",
                                                             "http://www.benchmark.
↪org/family#married")),
                                                     tnorm="min", k=3)
top_entities = [topk_entity for topk_entity, query_score in predictions]
assert "http://www.benchmark.org/family#F9M142" in top_entities
assert "http://www.benchmark.org/family#F9F158" in top_entities
# (3) Three-hop query answering
# Query: ?T : \exist D.type(D,T) \land Married(D,E) \land hasSibling(E, F9M167)
# Question: What are the type of people who are married to a sibling of F9M167?
# (3) Answer: [Person, Male, Father] since  F9M157 is [Brother Father Grandfather␣
↪Male] and F9M142 is [Male Grandfather Father]

predictions = pre_trained_kge.answer_multi_hop_query(query_type="3p", query=("http://
↪www.benchmark.org/family#F9M167",
                                                                             ("http://
↪www.benchmark.org/family#hasSibling",
                                                                              "http://
↪www.benchmark.org/family#married",
                                                                              "http://
↪www.w3.org/1999/02/22-rdf-syntax-ns#type")),
                                                     tnorm="min", k=5)
top_entities = [topk_entity for topk_entity, query_score in predictions]
print(top_entities)
assert "http://www.benchmark.org/family#Person" in top_entities
assert "http://www.benchmark.org/family#Father" in top_entities
assert "http://www.benchmark.org/family#Male" in top_entities
```

For more, please refer to `examples/multi_hop_query_answering`.

8 Predicting Missing Links

```
from dicee import KGE
# (1) Train a knowledge graph embedding model..
# (2) Load a pretrained model
pre_trained_kge = KGE(path='..')
# (3) Predict missing links through head entity rankings
pre_trained_kge.predict_topk(h=[".."],r=[".."],topk=10)
# (4) Predict missing links through relation rankings
pre_trained_kge.predict_topk(h=[".."],t=[".."],topk=10)
# (5) Predict missing links through tail entity rankings
pre_trained_kge.predict_topk(r=[".."],t=[".."],topk=10)
```

9 Downloading Pretrained Models

```
from dicee import KGE
# (1) Load a pretrained ConEx on DBpedia
model = KGE(url="https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-
↪dim128-epoch256-KvsAll")
```

- For more please look at dice-research.org/projects/DiceEmbeddings/[11]

10 How to Deploy

```
from dicee import KGE
KGE(path='...').deploy(share=True,top_k=10)
```

11 Docker

To build the Docker image:

```
docker build -t dice-embeddings .
```

To test the Docker image:

```
docker run --rm -v ~/.local/share/dicee/KGs:/dicee/KGs dice-embeddings ./main.py --
↪model AConEx --embedding_dim 16
```

[11] https://files.dice-research.org/projects/DiceEmbeddings/

12 How to cite

Currently, we are working on our manuscript describing our framework. If you really like our work and want to cite it now, feel free to chose one :)

```
# Keci
@inproceedings{demir2023clifford,
  title={Clifford Embeddings--A Generalized Approach for Embedding in Normed Algebras}
↪,
  author={Demir, Caglar and Ngonga Ngomo, Axel-Cyrille},
  booktitle={Joint European Conference on Machine Learning and Knowledge Discovery in␣
↪Databases},
  pages={567--582},
  year={2023},
  organization={Springer}
}
# LitCQD
@inproceedings{demir2023litcqd,
  title={LitCQD: Multi-Hop Reasoning in Incomplete Knowledge Graphs with Numeric␣
↪Literals},
  author={Demir, Caglar and Wiebesiek, Michel and Lu, Renzhong and Ngonga Ngomo, Axel-
↪Cyrille and Heindorf, Stefan},
  booktitle={Joint European Conference on Machine Learning and Knowledge Discovery in␣
↪Databases},
  pages={617--633},
  year={2023},
  organization={Springer}
}
# DICE Embedding Framework
@article{demir2022hardware,
  title={Hardware-agnostic computation for large-scale knowledge graph embeddings},
  author={Demir, Caglar and Ngomo, Axel-Cyrille Ngonga},
  journal={Software Impacts},
  year={2022},
  publisher={Elsevier}
}
# KronE
@inproceedings{demir2022kronecker,
  title={Kronecker decomposition for knowledge graph embeddings},
  author={Demir, Caglar and Lienen, Julian and Ngonga Ngomo, Axel-Cyrille},
  booktitle={Proceedings of the 33rd ACM Conference on Hypertext and Social Media},
  pages={1--10},
  year={2022}
}
# QMult, OMult, ConvQ, ConvO
@InProceedings{pmlr-v157-demir21a,
  title =          {Convolutional Hypercomplex Embeddings for Link Prediction},
  author =       {Demir, Caglar and Moussallem, Diego and Heindorf, Stefan and Ngonga␣
↪Ngomo, Axel-Cyrille},
  booktitle =          {Proceedings of The 13th Asian Conference on Machine Learning},
  pages =          {656--671},
  year =          {2021},
  editor =          {Balasubramanian, Vineeth N. and Tsang, Ivor},
  volume =          {157},
  series =          {Proceedings of Machine Learning Research},
  month =          {17--19 Nov},
  publisher =    {PMLR},
```

(continues on next page)

(continued from previous page)

```
  pdf =          {https://proceedings.mlr.press/v157/demir21a/demir21a.pdf},
  url =          {https://proceedings.mlr.press/v157/demir21a.html},
}
# ConEx
@inproceedings{demir2021convolutional,
title={Convolutional Complex Knowledge Graph Embeddings},
author={Caglar Demir and Axel-Cyrille Ngonga Ngomo},
booktitle={Eighteenth Extended Semantic Web Conference - Research Track},
year={2021},
url={https://openreview.net/forum?id=6T45-4TFqaX}}
# Shallom
@inproceedings{demir2021shallow,
  title={A shallow neural model for relation prediction},
  author={Demir, Caglar and Moussallem, Diego and Ngomo, Axel-Cyrille Ngonga},
  booktitle={2021 IEEE 15th International Conference on Semantic Computing (ICSC)},
  pages={179--182},
  year={2021},
  organization={IEEE}
```

For any questions or wishes, please contact: caglar.demir@upb.de

13 dicee

13.1 Subpackages

dicee.models

Submodules

dicee.models.base_model

Module Contents

Classes

BaseKGELightning	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
IdentityClass	A class that represents an identity function.

class dicee.models.base_model.**BaseKGELightning**(**args*, ***kwargs*)

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

mem_of_model() → Dict

Size of model in MB and number of params

training_step(*batch*, *batch_idx=None*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** – The output of your data iterable, normally a `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple dataloaders used)

Returns

- `Tensor` - The loss tensor
- `dict` - A dictionary which can include any keys, but must include the key `'loss'` in the case of automatic optimization.
- `None` - In automatic optimization, this will skip to the next batch (but is not supported for multi-GPU, TPU, or DeepSpeed). For manual optimization, this has no special meaning, as returning the loss is not required.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

To use multiple optimizers, you can switch to 'manual optimization' and control their stepping:

```
def __init__(self):
    super().__init__()
    self.automatic_optimization = False


# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

    # do training_step with encoder
    ...
    opt1.step()
    # do training_step with decoder
    ...
    opt2.step()
```

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

loss_function(*yhat_batch: torch.FloatTensor*, *y_batch: torch.FloatTensor*)

Parameters

- **yhat_batch** –
- **y_batch** –

on_train_epoch_end(**args*, ***kwargs*)

Called in the training loop at the very end of the epoch.

To access all batch outputs at the end of the epoch, you can cache step outputs as an attribute of the `LightningModule` and access them in this hook:

```
class MyLightningModule(L.LightningModule):
    def __init__(self):
        super().__init__()
        self.training_step_outputs = []

    def training_step(self):
        loss = ...
        self.training_step_outputs.append(loss)
        return loss

    def on_train_epoch_end(self):
        # do something with all training_step outputs, for example:
        epoch_mean = torch.stack(self.training_step_outputs).mean()
        self.log("training_epoch_mean", epoch_mean)
        # free up the memory
        self.training_step_outputs.clear()
```

test_epoch_end(*outputs: List[Any]*)

test_dataloader() → None

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `test()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Note: If you don't need a test dataset and a `test_step()`, you don't need to implement this method.

`val_dataloader``()` → None

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:paramref:`~lightning.pytorch.trainer.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `fit()`
- `validate()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Note: If you don't need a validation dataset and a `validation_step()`, you don't need to implement this method.

`predict_dataloader``()` → None

An iterable or collection of iterables specifying prediction samples.

For more information about multiple dataloaders, see this section.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `predict()`

- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Returns

A `torch.utils.data.DataLoader` or a sequence of them specifying prediction samples.

`train_dataloader``()` → None

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:paramref:`~lightning.pytorch.trainer.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

`configure_optimizers`(*parameters=None*)

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple. Optimization with multiple optimizers only works in the manual optimization mode.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.

- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` automatically in case of automatic optimization.
- If a learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizer.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, you will have to switch to 'manual optimization' mode and step them yourself.
- If you need to control how often the optimizer steps, override the `optimizer_step()` hook.

---

class dicee.models.base_model.**BaseKGE**(*args: dict*)

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_byte_pair_encoded_k_vs_all (*x: torch.LongTensor*)

Parameters

x (*B x 2 x T*) –

forward_byte_pair_encoded_triple (*x: Tuple[torch.LongTensor, torch.LongTensor]*)

Perform the forward pass for byte pair encoded triples.

Parameters

x (*Tuple[torch.LongTensor, torch.LongTensor]*) – The input tuple containing byte pair encoded entities and relations.

Returns

The output tensor containing the scores for the byte pair encoded triples.

Return type

torch.Tensor

init_params_with_sanity_checking()

forward (*x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor]*, *y_idx: torch.LongTensor = None*)

Perform the forward pass of the model.

Parameters

- **x** (*Union[torch.LongTensor, Tuple[torch.LongTensor, torch.LongTensor]]*) – The input tensor or a tuple containing the input tensor and target entity indexes.
- **y_idx** (*torch.LongTensor, optional*) – The target entity indexes (default is None).

Returns
The output of the forward pass.

Return type
Any

forward_triples(*x: torch.LongTensor*) → torch.Tensor

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

forward_k_vs_all(**args*, ***kwargs*)

Forward pass for K vs. All.

Raises
ValueError – This function is not implemented in the current model.

forward_k_vs_sample(**args*, ***kwargs*)

Forward pass for K vs. Sample.

Raises
ValueError – This function is not implemented in the current model.

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for the head and relation entities.

Parameters
indexed_triple (*torch.LongTensor*) – The indexes of the head and relation entities.

Returns
The representation for the head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_sentence_representation(*x: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Get the representation for a sentence.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The representation for the input sentence.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

get_bpe_head_and_relation_representation(*x: torch.LongTensor*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for BPE head and relation entities.

Parameters
x (*B x 2 x T*) –

Returns
The representation for BPE head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]

Get the entity and relation embeddings.

Returns
The entity and relation embeddings.

Return type
Tuple[np.ndarray, np.ndarray]

class dicee.models.base_model.**IdentityClass**(*args: Dict | None = None*)

Bases: torch.nn.Module

A class that represents an identity function.

Parameters
args (*dict, optional*) – A dictionary containing arguments (default is None).

__call__(*x*)

static forward(*x: torch.Tensor*) → torch.Tensor

The forward pass of the identity function.

Parameters
x (*torch.Tensor*) – The input tensor.

Returns
The output tensor, which is the same as the input.

Return type
torch.Tensor

dicee.models.clifford

Module Contents

Classes

CMult	The CMult class represents a specific kind of mathematical object used in knowledge graph embeddings,
Keci	The Keci class is a knowledge graph embedding model that incorporates Clifford algebra for embeddings.
KeciBase	The KeciBase class is a variant of the Keci class for knowledge graph embeddings, with the key difference being
DeCaL	Base class for all neural network modules.
KeciBase	The KeciBase class is a variant of the Keci class for knowledge graph embeddings, with the key difference being
DeCaL	Base class for all neural network modules.

class dicee.models.clifford.**CMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

The CMult class represents a specific kind of mathematical object used in knowledge graph embeddings, involving Clifford algebra multiplication. It defines several algebraic structures based on the signature (p, q), such as Real Numbers, Complex Numbers, Quaternions, and others. The class provides functionality for performing Clifford multiplication, a generalization of the geometric product for vectors in a Clifford algebra.

TODO: Add mathematical format for sphinx.

Cl_(0,0) => Real Numbers

Cl_(0,1) =>

A multivector mathbf{a} = a_0 + a_1 e_1 A multivector mathbf{b} = b_0 + b_1 e_1

multiplication is isomorphic to the product of two complex numbers

mathbf{a} imes mathbf{b} = a_0 b_0 + a_0b_1 e1 + a_1 b_1 e_1 e_1
= (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) e_1

Cl_(2,0) =>

A multivector mathbf{a} = a_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 e_2 A multivector mathbf{b} = b_0 + b_1 e_1 + b_2 e_2 + b_{12} e_1 e_2

mathbf{a} imes mathbf{b} = a_0b_0 + a_0b_1 e_1 + a_0b_2e_2 + a_0 b_12 e_1 e_2

- a_1 b_0 e_1 + a_1b_1 e_1_e1 ..

Cl_(0,2) => Quaternions

name

The name identifier for the CMult class.

Type
str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

p

Non-negative integer representing the number of positive square terms in the Clifford algebra.

Type
int

q

Non-negative integer representing the number of negative square terms in the Clifford algebra.

Type
int

clifford_mul (*x: torch.FloatTensor*, *y: torch.FloatTensor*, *p: int*, *q: int*) → tuple

Performs Clifford multiplication based on the given signature (p, q).

score (*head_ent_emb*, *rel_ent_emb*, *tail_ent_emb*) → torch.FloatTensor

Computes a scoring function for a head entity, relation, and tail entity embeddings.

forward_triples (*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples.

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities in the knowledge graph.

clifford_mul (*x: torch.FloatTensor*, *y: torch.FloatTensor*, *p: int*, *q: int*) → tuple

Performs Clifford multiplication in the Clifford algebra Cl_{p,q}. This method generalizes the geometric product of vectors in a Clifford algebra, handling different algebraic structures like real numbers, complex numbers, quaternions, etc., based on the signature (p, q).

Clifford multiplication Cl_{p,q} (mathbb{R})

ei ^2 = +1 for i =< i =< p ej ^2 = -1 for p < j =< p+q ei ej = -eje1 for i

eq j

x
[torch.FloatTensor] The first multivector operand with shape (n, d).

y
[torch.FloatTensor] The second multivector operand with shape (n, d).

p
[int] A non-negative integer representing the number of positive square terms in the Clifford algebra.

q
[int] A non-negative integer representing the number of negative square terms in the Clifford algebra.

tuple
The result of Clifford multiplication, a tuple of tensors representing the components of the resulting multivector.

score (*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes a scoring function for a given triple of head entity, relation, and tail entity embeddings. The method involves Clifford multiplication of the head entity and relation embeddings, followed by a calculation of the score with the tail entity embedding.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – Embedding of the head entity.
- **rel_ent_emb** (*torch.FloatTensor*) – Embedding of the relation.
- **tail_ent_emb** (*torch.FloatTensor*) – Embedding of the tail entity.

Returns
A tensor representing the score of the given triple.

Return type
torch.FloatTensor

forward_triples (*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples. This method is typically used in training or evaluation of knowledge graph embedding models. It applies Clifford multiplication to the embeddings of head entities and relations and then calculates the score with respect to the tail entity embeddings.

Parameters
x (*torch.LongTensor*) – A tensor with shape (n, 3) representing a batch of triples, where each triple consists of indices for a head entity, a relation, and a tail entity.

Returns
A tensor with shape (n,) containing the scores for each triple in the batch.

Return type
torch.FloatTensor

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities in the knowledge graph, often used in KvsAll evaluation. This method retrieves embeddings for heads and relations, performs Clifford multiplication, and then computes the inner product with all entity embeddings to get scores for every possible triple involving the given heads and relations.

Parameters
x (*torch.Tensor*) – A tensor with shape (n, 3) representing a batch of triples, where each triple consists of indices for a head entity and a relation. The tail entity is to be compared against all possible entities.

Returns
A tensor with shape (n,) containing scores for each triple against all possible tail entities.

Return type
torch.FloatTensor

class dicee.models.clifford.**Keci** (*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

The Keci class is a knowledge graph embedding model that incorporates Clifford algebra for embeddings. It supports different dimensions of Clifford algebra by setting the parameters p and q. The class utilizes Clifford multiplication for embedding interactions and computes scores for knowledge graph triples.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model.

name

The name identifier for the Keci class.

Type
str

p

The parameter ‘p’ in Clifford algebra, representing the number of positive square terms.

Type
int

q

The parameter ‘q’ in Clifford algebra, representing the number of negative square terms.

Type
int

r

A derived attribute for dimension scaling based on ‘p’ and ‘q’.

Type
int

p_coefficients

Embedding for scaling coefficients of ‘p’ terms, if ‘p’ > 0.

Type
torch.nn.Embedding (optional)

q_coefficients

Embedding for scaling coefficients of ‘q’ terms, if ‘q’ > 0.

Type
torch.nn.Embedding (optional)

compute_sigma_pp(*hp: torch.Tensor*, *rp: torch.Tensor*) → torch.Tensor

Computes the sigma_pp component in Clifford multiplication.

compute_sigma_qq(*hq: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_qq component in Clifford multiplication.

compute_sigma_pq(*hp: torch.Tensor*, *hq: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_pq component in Clifford multiplication.

apply_coefficients(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → tuple

Applies scaling coefficients to the base vectors in Clifford algebra.

clifford_multiplication(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → tuple

Performs Clifford multiplication of head and relation embeddings.

construct_cl_multivector(*x: torch.FloatTensor*, *r: int*, *p: int*, *q: int*) → tuple

Constructs a multivector in Clifford algebra Cl_{p,q}(mathbb{R}^d).

forward_k_vs_with_explicit(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities using explicit Clifford multiplication.

k_vs_all_score(*bpe_head_ent_emb: torch.Tensor*, *bpe_rel_ent_emb: torch.Tensor*, *E: torch.Tensor*) → torch.FloatTensor

Computes scores for all triples using Clifford multiplication in a K-vs-All setup.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Wrapper function for K-vs-All scoring.

forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Computes scores for a sampled subset of entities.

score(*h: torch.Tensor*, *r: torch.Tensor*, *t: torch.Tensor*) → torch.FloatTensor

Computes the score for a given triple using Clifford multiplication.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples.

Notes

The class is designed to work with embeddings in the context of knowledge graph completion tasks, leveraging the properties of Clifford algebra for embedding interactions.

compute_sigma_pp(*hp: torch.Tensor*, *rp: torch.Tensor*) → torch.Tensor

Computes the sigma_pp component in Clifford multiplication, representing the interactions between the positive square terms in the Clifford algebra.

sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k, TODO: Add mathematical format for sphinx.

sigma_{pp} captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

for k in range(i + 1, p):
results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

Parameters

- **hp** (*torch.Tensor*) – The 'p' part of the head entity embedding in Clifford algebra.
- **rp** (*torch.Tensor*) – The 'p' part of the relation embedding in Clifford algebra.

Returns
sigma_pp – The sigma_pp component of the Clifford multiplication.

Return type
torch.Tensor

compute_sigma_qq(*hq: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_qq component in Clifford multiplication, representing the interactions between the negative square terms in the Clifford algebra.

TODO: Add mathematical format for sphinx.

sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k sigma_{q} captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

for k in range(j + 1, q):
results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

Parameters

- **hq** (*torch.Tensor*) – The 'q' part of the head entity embedding in Clifford algebra.
- **rq** (*torch.Tensor*) – The 'q' part of the relation embedding in Clifford algebra.

Returns
sigma_qq – The sigma_qq component of the Clifford multiplication.

Return type
torch.Tensor

compute_sigma_pq(*, *hp: torch.Tensor*, *hq: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_pq component in Clifford multiplication, representing the interactions between the positive and negative square terms in the Clifford algebra.

TODO: Add mathematical format for sphinx.

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] # sigma_pq = torch.zeros(b, r, p, q) # for i in range(p): # for j in range(q): # sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i] # print(sigma_pq.shape)

Parameters

- **hp** (*torch.Tensor*) – The 'p' part of the head entity embedding in Clifford algebra.
- **hq** (*torch.Tensor*) – The 'q' part of the head entity embedding in Clifford algebra.
- **rp** (*torch.Tensor*) – The 'p' part of the relation embedding in Clifford algebra.
- **rq** (*torch.Tensor*) – The 'q' part of the relation embedding in Clifford algebra.

Returns
sigma_pq – The sigma_pq component of the Clifford multiplication.

Return type
torch.Tensor

apply_coefficients(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTe

Applies scaling coefficients to the base vectors in the Clifford algebra. This method is used for adjusting the contributions of different components in the algebra.

Parameters

- **h0** (*torch.Tensor*) – The scalar part of the head entity embedding.
- **hp** (*torch.Tensor*) – The 'p' part of the head entity embedding.
- **hq** (*torch.Tensor*) – The 'q' part of the head entity embedding.
- **r0** (*torch.Tensor*) – The scalar part of the relation embedding.
- **rp** (*torch.Tensor*) – The 'p' part of the relation embedding.
- **rq** (*torch.Tensor*) – The 'q' part of the relation embedding.

Returns
Tuple containing the scaled components of the head and relation embeddings.

Return type
tuple

clifford_multiplication(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTe

Performs Clifford multiplication of head and relation embeddings. This method computes the various components of the Clifford product, combining the scalar, 'p', and 'q' parts of the embeddings.

TODO: Add mathematical format for sphinx.

h = h_0 + sum_{i=1}^p h_i e_i + sum_{j=p+1}^{p+q} h_j e_j r = r_0 + sum_{i=1}^p r_i e_i + sum_{j=p+1}^{p+q} r_j e_j

ei ^2 = +1 for i =< i =< p ej ^2 = -1 for p < j =< p+q ei ej = -eje1 for i

eq j

h r = sigma_0 + sigma_p + sigma_q + sigma_{pp} + sigma_{q}+ sigma_{pq} where

(1) sigma_0 = h_0 r_0 + sum_{i=1}^p (h_0 r_i) e_i - sum_{j=p+1}^{p+q} (h_j r_j) e_j

(2) sigma_p = sum_{i=1}^p (h_0 r_i + h_i r_0) e_i

(3) sigma_q = sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j

(4) sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k

(5) sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k

(6) sigma_{pq} = sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

h0
[torch.Tensor] The scalar part of the head entity embedding.

hp
[torch.Tensor] The 'p' part of the head entity embedding.

hq
[torch.Tensor] The 'q' part of the head entity embedding.

r0
[torch.Tensor] The scalar part of the relation embedding.

rp
[torch.Tensor] The 'p' part of the relation embedding.

rq
[torch.Tensor] The 'q' part of the relation embedding.

tuple
Tuple containing the components of the Clifford product.

construct_cl_multivector (*x: torch.FloatTensor*, *r: int*, *p: int*, *q: int*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors Cl_{p,q}(mathbb{R}^d)

Parameter

x
[torch.FloatTensor] The embedding vector with shape (n, d).

r
[int] The dimension of the scalar part.

p
[int] The number of positive square terms.

q
[int] The number of negative square terms.

returns

- **a0** (*torch.FloatTensor*) – Tensor with (n,r) shape
- **ap** (*torch.FloatTensor*) – Tensor with (n,r,p) shape
- **aq** (*torch.FloatTensor*) – Tensor with (n,r,q) shape

forward_k_vs_with_explicit (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities using explicit Clifford multiplication. This method is used for K-vs-All training and evaluation.

Parameters
x (*torch.Tensor*) – Tensor representing a batch of head entities and relations.

Returns
A tensor containing scores for each triple against all entities.

Return type
torch.FloatTensor

k_vs_all_score (*bpe_head_ent_emb: torch.Tensor*, *bpe_rel_ent_emb: torch.Tensor*, *E: torch.Tensor*)
→ torch.FloatTensor

Computes scores for all triples using Clifford multiplication in a K-vs-All setup. This method involves constructing multivectors for head entities and relations in Clifford algebra, applying coefficients, and computing interaction scores based on different components of the Clifford algebra.

Parameters

- **bpe_head_ent_emb** (*torch.Tensor*) – Batch of head entity embeddings in BPE (Byte Pair Encoding) format. Tensor shape: (batch_size, embedding_dim).
- **bpe_rel_ent_emb** (*torch.Tensor*) – Batch of relation embeddings in BPE format. Tensor shape: (batch_size, embedding_dim).
- **E** (*torch.Tensor*) – Tensor containing all entity embeddings. Tensor shape: (num_entities, embedding_dim).

Returns
Tensor containing the scores for each triple in the K-vs-All setting. Tensor shape: (batch_size, num_entities).

Return type
torch.FloatTensor

Notes

The method computes scores based on the basis of 1 (scalar part), the bases of 'p' (positive square terms), and the bases of 'q' (negative square terms). Additional computations involve sigma_pp, sigma_qq, and sigma_pq components in Clifford multiplication, corresponding to different interaction terms.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

TODO: Add mathematical format for sphinx. Performs the forward pass for K-vs-All training and evaluation in knowledge graph embeddings. This method involves retrieving real-valued embedding vectors for head entities and relations mathbb{R}^d, constructing Clifford algebra multivectors for these embeddings according to Cl_{p,q}(mathbb{R}^d), performing Clifford multiplication, and computing the inner product with all entity embeddings.

Parameters
x (*torch.Tensor*) – A tensor representing a batch of head entities and relations for the K-vs-All evaluation. Expected tensor shape: (n, 2), where 'n' is the batch size and '2' represents head entity and relation pairs.

Returns
A tensor containing the scores for each head entity and relation pair against all possible tail entities in the knowledge graph. Tensor shape: (n, |E|), where '|E|' is the number of entities in the knowledge graph.

Return type
torch.FloatTensor

Notes

This method is similar to the 'forward_k_vs_with_explicit' function in functionality. It is typically used in scenarios where every possible combination of a head entity and a relation is scored against all tail entities, commonly used in knowledge graph completion tasks.

forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*)
→ torch.FloatTensor

TODO: Add mathematical format for sphinx.

Performs the forward pass for K-vs-Sample training in knowledge graph embeddings. This method involves retrieving real-valued embedding vectors for head entities and relations mathbb{R}^d, constructing Clifford algebra multivectors for these embeddings according to Cl_{p,q}(mathbb{R}^d), performing Clifford multiplication, and computing the inner product with a sampled subset of entity embeddings.

Parameters

- **x** (*torch.LongTensor*) – A tensor representing a batch of head entities and relations for the K-vs-Sample evaluation. Expected tensor shape: (n, 2), where 'n' is the batch size and '2' represents head entity and relation pairs.
- **target_entity_idx** (*torch.LongTensor*) – A tensor of target entity indices for sampling in the K-vs-Sample evaluation. Tensor shape: (n, sample_size), where 'sample_size' is the number of entities sampled.

Returns

A tensor containing the scores for each head entity and relation pair against the sampled subset of tail entities. Tensor shape: (n, sample_size).

Return type

torch.FloatTensor

Notes

This method is used in scenarios where every possible combination of a head entity and a relation is scored against a sampled subset of tail entities, commonly used in knowledge graph completion tasks with a large number of entities.

score(*h: torch.Tensor*, *r: torch.Tensor*, *t: torch.Tensor*) → torch.FloatTensor

Computes the score for a given triple using Clifford multiplication in the context of knowledge graph embeddings. This method involves constructing Clifford algebra multivectors for head entities, relations, and tail entities, applying coefficients, and computing interaction scores based on different components of the Clifford algebra.

Parameters

- **h** (*torch.Tensor*) – Tensor representing the embeddings of head entities. Expected shape: (n, d), where 'n' is the number of triples and 'd' is the embedding dimension.
- **r** (*torch.Tensor*) – Tensor representing the embeddings of relations. Expected shape: (n, d).
- **t** (*torch.Tensor*) – Tensor representing the embeddings of tail entities. Expected shape: (n, d).

Returns

Tensor containing the scores for each triple. Tensor shape: (n,).

Return type

torch.FloatTensor

Notes

The method computes scores based on the scalar part, the bases of 'p' (positive square terms), and the bases of 'q' (negative square terms) in Clifford algebra. It includes additional computations involving sigma_pp, sigma_qq, and sigma_pq components, which correspond to different interaction terms in the Clifford product.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using Clifford multiplication. This method is involved in the forward pass of the model during training or evaluation. It retrieves embeddings for head entities, relations, and tail entities, constructs Clifford algebra multivectors, applies coefficients, and computes interaction scores based on different components of Clifford algebra.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of triples. Each triple consists of indices for a head entity, a relation, and a tail entity. Expected tensor shape: (n, 3), where 'n' is the number of triples.

Returns

A tensor containing the scores for each triple in the batch. Tensor shape: (n,), where 'n' is the number of triples.

Return type

torch.FloatTensor

Notes

The method computes scores based on the scalar part, the bases of 'p' (positive square terms), and the bases of 'q' (negative square terms) in Clifford algebra. It includes additional computations involving sigma_pp, sigma_qq, and sigma_pq components, corresponding to different interaction terms in the Clifford product.

class dicee.models.clifford.**KeciBase**(*args*)

Bases: *Keci*

The KeciBase class is a variant of the Keci class for knowledge graph embeddings, with the key difference being the lack of learning for dimension scaling. It inherits the core functionality from the Keci class but sets the gradient requirement for interaction coefficients to False, indicating these coefficients are not updated during training.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, including 'p', 'q', and embedding dimensions.

name

The name identifier for the KeciBase class.

Type

str

requires_grad_for_interactions

Flag to indicate if the interaction coefficients require gradients. In KeciBase, this is set to False.

Type

bool

p_coefficients

Embedding for scaling coefficients of 'p' terms, initialized to ones if 'p' > 0.

Type

torch.nn.Embedding (optional)

q_coefficients

Embedding for scaling coefficients of 'q' terms, initialized to ones if 'q' > 0.

Type

torch.nn.Embedding (optional)

Notes

KeciBase is designed for scenarios where fixed coefficients are preferred over learnable parameters for dimension scaling in the Clifford algebra-based embedding interactions.

class dicee.models.clifford.**DeCaL**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables
training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,3) shape

rtype
torch.FloatTensor with (n) shape

class dicee.models.clifford.**KeciBase**(*args*)

Bases: *Keci*

Without learning dimension scaling

class dicee.models.clifford.**DeCaL**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training` (*`bool`*) – Boolean represents whether this module is in training or evaluation mode.

`forward_triples`(*x: torch.Tensor*) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,3) shape

rtype

torch.FloatTensor with (n) shape

`cl_pqr`(*a*)

Input: tensor(batch_size, emb_dim) —-> output: tensor with 1+p+q+r components with size (batch_size, emb_dim/(1+p+q+r)) each.

1) takes a tensor of size (batch_size, emb_dim), split it into 1 + p + q +r components, hence 1+p+q+r must be a divisor of the emb_dim. 2) Return a list of the 1+p+q+r components vectors, each are tensors of size (batch_size, emb_dim/(1+p+q+r))

`compute_sigmas_single`(*list_h_emb*, *list_r_emb*, *list_t_emb*)

here we compute all the sums with no others vectors interaction taken with the scalar product with t, that is, 1) s0 = h_0r_0t_0 2) s1 = sum_{i=1}^{p}h_ir_it_0 3) s2 = sum_{j=p+1}^{p+q}h_jr_jt_0 4) s3 = sum_{i=1}^{q}(h_0r_it_i + h_ir_0t_i) 5) s4 = sum_{i=p+1}^{p+q}(h_0r_it_i + h_ir_0t_i) 5) s5 = sum_{i=p+q+1}^{p+q+r}(h_0r_it_i + h_ir_0t_i)

and return:

*) sigma_0t = sigma_0 cdot t_0 = s0 + s1 -s2 *) s3, s4 and s5

`compute_sigmas_multivect`(*list_h_emb*, *list_r_emb*)

Here we compute and return all the sums with vectors interaction for the same and different bases.

For same bases vectors interaction we have

1) sigma_pp = sum_{i=1}^{p-1}sum_{i'=i+1}^{p}(h_ir_{i'}-h_{i'}r_i) (models the interactions between e_i and e_i' for 1 <= i, i' <= p)

2) sigma_qq = sum_{j=p+1}^{p+q-1}sum_{j'=j+1}^{p+q}(h_jr_{j'}-h_{j'} (models the interactions between e_j and e_j' for p+1 <= j, j' <= p+q)

3) sigma_rr = sum_{k=p+q+1}^{p+q+r-1}sum_{k'=k+1}^{p}(h_kr_{k'}-h_{k'}r_k) (models the interactions between e_k and e_k' for p+q+1 <= k, k' <= p+q+r)

For different base vector interactions, we have

4) sigma_pq = sum_{i=1}^{p}sum_{j=p+1}^{p+q}(h_ir_j - h_jr_i) (interactionsn between e_i and e_j for 1<=i <=p and p+1<= j <= p+q)

5) sigma_pr = sum_{i=1}^{p}sum_{k=p+q+1}^{p+q+r}(h_ir_k - h_kr_i) (interactionsn between e_i and e_k for 1<=i <=p and p+q+1<= k <= p+q+r)

6) sigma_qr = sum_{j=p+1}^{p+q}sum_{j=p+q+1}^{p+q+r}(h_jr_k - h_kr_j) (interactionsn between e_j and e_k for p+1 <= j <=p+q and p+q+1<= j <= p+q+r)

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Kvsall training

(1) Retrieve real-valued embedding vectors for heads and relations mathbb{R}^d .

(2) Construct head entity and relation embeddings according to Cl_{p,q}(mathbb{R}^d) .

(3) Perform Cl multiplication

(4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this funcitons are identical Parameter ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, |E|) shape

apply_coefficients (*h0*, *hp*, *hq*, *hk*, *r0*, *rp*, *rq*, *rk*)

Multiplying a base vector with its scalar coefficient

construct_cl_multivector (*x: torch.FloatTensor*, *re: int*, *p: int*, *q: int*, *r: int*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors Cl_{p,q,r}(mathbb{R}^d)

Parameter

x: torch.FloatTensor with (n,d) shape

returns

- **a0** (*torch.FloatTensor*)
- **ap** (*torch.FloatTensor*)
- **aq** (*torch.FloatTensor*)
- **ar** (*torch.FloatTensor*)

compute_sigma_pp (*hp*, *rp*)

sigma_{p,p}^* = sum_{i=1}^{p-1}sum_{i'=i+1}^{p}(x_iy_{i'}-x_{i'}y_i)

sigma_{pp} captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

for k in range(i + 1, p):
results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

compute_sigma_qq(*hq*, *rq*)

Compute sigma_{q,q}^* = sum_{j=p+1}^{p+q-1}sum_{j'=j+1}^{p+q}(x_jy_{j'}-x_{j'}y_j) Eq. 16 sigma_{q} captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

for k in range(j + 1, q):
results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

compute_sigma_rr(*hk*, *rk*)

sigma_{r,r}^* = sum_{k=p+q+1}^{p+q+r-1}sum_{k'=k+1}^{p}(x_ky_{k'}-x_{k'}y_k)

compute_sigma_pq(*, *hp*, *hq*, *rp*, *rq*)

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):
sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

print(sigma_pq.shape)

compute_sigma_pr(*, *hp*, *hk*, *rp*, *rk*)

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):
sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

print(sigma_pq.shape)

compute_sigma_qr(*, *hq*, *hk*, *rq*, *rk*)

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):
sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

print(sigma_pq.shape)

dicee.models.complex

Module Contents

Classes

ConEx	ConEx (Convolutional ComplEx) is a Knowledge Graph Embedding model that extends ComplEx embeddings with convolutional layers.
AConEx	AConEx (Additive Convolutional ComplEx) extends the ConEx model by incorporating
ComplEx	ComplEx (Complex Embeddings for Knowledge Graphs) is a model that extends

class dicee.models.complex.**ConEx**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

ConEx (Convolutional ComplEx) is a Knowledge Graph Embedding model that extends ComplEx embeddings with convolutional layers. It integrates convolutional neural networks into the embedding process to capture complex patterns in the data.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, kernel size, number of output channels, and dropout rates.

name

The name identifier for the ConEx model.

Type

str

conv2d

A 2D convolutional layer used for processing complex-valued embeddings.

Type

torch.nn.Conv2d

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type

torch.nn.Linear

norm_fc1

Normalization layer applied after the fully connected layer.

Type

Normalizer

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type

torch.nn.BatchNorm2d

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(*C_1: Tuple[torch.Tensor, torch.Tensor]*, *C_2: Tuple[torch.Tensor, torch.Tensor]*) → Tuple[torch.Tensor, torch.Tensor]

Performs a residual convolution operation on two complex-valued embeddings.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional operations on embeddings.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*) → torch.Tensor

Computes scores against a sampled subset of entities using convolutional operations.

Notes

ConEx combines complex-valued embeddings with convolutional neural networks to capture intricate patterns and interactions in the knowledge graph, potentially leading to improved performance on tasks like link prediction.

residual_convolution(*C_1: Tuple[torch.Tensor, torch.Tensor]*, *C_2: Tuple[torch.Tensor, torch.Tensor]*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Computes the residual score of two complex-valued embeddings by applying convolutional operations. This method is a key component of the ConEx model, combining complex embeddings with convolutional neural networks.

Parameters

- **C_1** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple consisting of two PyTorch tensors representing the real and imaginary components of the first complex-valued embedding.
- **C_2** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple consisting of two PyTorch tensors representing the real and imaginary components of the second complex-valued embedding.

Returns
A tuple of two tensors, representing the real and imaginary parts of the convolutionally transformed embeddings.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

Notes

The method involves concatenating the real and imaginary components of the embeddings, applying a 2D convolution, followed by batch normalization, ReLU activation, dropout, and a fully connected layer. This process is intended to capture complex interactions between the embeddings in a convolutional manner.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional operations on complex-valued embeddings. This method is used for evaluating the performance of the model by computing scores for each head entity and relation pair against all possible tail entities.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of head entities and relations. Expected tensor shape: (n, 2), where 'n' is the batch size and '2' represents head entity and relation pairs.

Returns

A tensor containing the scores for each head entity and relation pair against all possible tail entities. Tensor shape: (n, |E|), where '|E|' is the number of entities in the knowledge graph.

Return type

torch.FloatTensor

Notes

The method retrieves embeddings for head entities and relations, splits them into real and imaginary parts, and applies a convolution operation. It then computes the Hermitian product of the transformed embeddings with all tail entity embeddings to generate scores. This approach allows for capturing complex relational patterns in the knowledge graph.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on complex-valued embeddings. This method is crucial for evaluating the performance of the model on individual triples in the knowledge graph.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of triples. Each triple consists of indices for a head entity, a relation, and a tail entity. Expected tensor shape: (n, 3), where 'n' is the number of triples.

Returns

A tensor containing the scores for each triple in the batch. Tensor shape: (n,), where 'n' is the number of triples.

Return type

torch.FloatTensor

Notes

The method retrieves embeddings for head entities, relations, and tail entities, and splits them into real and imaginary parts. It then applies a convolution operation on these embeddings and computes the Hermitian inner product, which involves a combination of real and imaginary parts of the embeddings. This process is designed to capture complex relational patterns and interactions within the knowledge graph, leveraging the power of convolutional neural networks.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*) → torch.Tensor

Computes scores against a sampled subset of entities using convolutional operations on complex-valued embeddings. This method is particularly useful for large knowledge graphs where computing scores against all entities is computationally expensive.

Parameters

- **x** (*torch.Tensor*) – A tensor representing a batch of head entities and relations. Expected tensor shape: (batch_size, 2), where 'batch_size' is the number of head entity and relation pairs.
- **target_entity_idx** (*torch.Tensor*) – A tensor of target entity indices for sampling. Tensor shape: (batch_size, num_selected_entities).

Returns
A tensor containing the scores for each head entity and relation pair against the sampled subset of tail entities. Tensor shape: (batch_size, num_selected_entities).

Return type
torch.Tensor

Notes

The method first retrieves and processes the embeddings for head entities and relations. It then applies a convolution operation and computes the Hermitian inner product with the embeddings of the sampled tail entities. This process enables capturing complex relational patterns in a computationally efficient manner.

class dicee.models.complex.**AConEx**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

AConEx (Additive Convolutional ComplEx) extends the ConEx model by incorporating additive connections in the convolutional operations. This model integrates convolutional neural networks with complex-valued embeddings, emphasizing additive feature interactions for knowledge graph embeddings.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, kernel size, number of output channels, and dropout rates.

name

The name identifier for the AConEx model.

Type
str

conv2d

A 2D convolutional layer used for processing complex-valued embeddings.

Type
torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

norm_fc1

Normalization layer applied after the fully connected layer.

Type
Normalizer

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(C_1: Tuple[torch.Tensor, torch.Tensor],

C_2: Tuple[torch.Tensor, torch.Tensor]) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Performs a residual convolution operation on two complex-valued embeddings.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional operations on embeddings.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*)

Computes scores against a sampled subset of entities using convolutional operations.

Notes

AConEx aims to enhance the modeling capabilities of knowledge graph embeddings by adding more complex interaction patterns through convolutional layers, potentially improving performance on tasks like link prediction.

residual_convolution(*C_1: Tuple[torch.Tensor, torch.Tensor]*,
C_2: Tuple[torch.Tensor, torch.Tensor])
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Computes the residual convolution of two complex-valued embeddings. This method is a core part of the AConEx model, applying convolutional neural network techniques to complex-valued embeddings to capture intricate relationships in the data.

Parameters

- **C_1** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple of two PyTorch tensors representing the real and imaginary components of the first complex-valued embedding.
- **C_2** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple of two PyTorch tensors representing the real and imaginary components of the second complex-valued embedding.

Returns
A tuple of four tensors, each representing a component of the convolutionally transformed embeddings. These components correspond to the modified real and imaginary parts of the input embeddings.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Notes

The method concatenates the real and imaginary components of the embeddings and applies a 2D convolution, followed by batch normalization, ReLU activation, dropout, and a fully connected layer. This convolutional process is designed to enhance the model's ability to capture complex patterns in knowledge graph embeddings.

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional and additive operations on complex-valued embeddings. This method evaluates the performance of the model by computing scores for each head entity and relation pair against all possible tail entities.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of head entities and relations. Expected tensor shape: (batch_size, 2), where 'batch_size' is the number of head entity and relation pairs.

Returns

A tensor containing the scores for each head entity and relation pair against all possible tail entities. Tensor shape: (batch_size, |E|), where '|E|' is the number of entities in the knowledge graph.

Return type

torch.FloatTensor

Notes

The method first retrieves embeddings for head entities and relations, splits them into real and imaginary parts, and applies a convolutional operation. It then computes the Hermitian inner product with all tail entity embeddings, using an additive approach that combines the convolutional results with the original embeddings. This technique aims to capture complex relational patterns in the knowledge graph.

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations and additive connections on complex-valued embeddings. This method is key for evaluating the model's performance on individual triples within the knowledge graph.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of triples. Each triple consists of indices for a head entity, a relation, and a tail entity. Expected tensor shape: (n, 3), where 'n' is the number of triples.

Returns

A tensor containing the scores for each triple in the batch. Tensor shape: (n,), where 'n' is the number of triples.

Return type

torch.FloatTensor

Notes

The method retrieves embeddings for head entities, relations, and tail entities, and splits them into real and imaginary parts. It then applies a convolution operation on these embeddings and computes the Hermitian inner product, enhanced with an additive connection. This approach allows the model to capture complex relational patterns within the knowledge graph, potentially improving prediction accuracy and interpretability.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of samples (entity pairs) given a batch of queries. This method is used to predict the scores for different tail entities for a set of query triples.

Parameters

- **x** (*torch.Tensor*) – A tensor representing a batch of query triples. Each triple consists of indices for a head entity, a relation, and a dummy tail entity (used for scoring). Expected tensor shape: (n, 3), where 'n' is the number of query triples.
- **target_entity_idx** (*torch.Tensor*) – A tensor containing the indices of the target tail entities for which scores are to be predicted. Expected tensor shape: (n, m), where 'n' is the number of queries and 'm' is the number of target entities.

Returns

A tensor containing the scores for each query-triple and target-entity pair. Tensor shape: (n, m), where 'n' is the number of queries and 'm' is the number of target entities.

Return type

torch.FloatTensor

Notes

This method retrieves embeddings for the head entities and relations in the query triples, splits them into real and imaginary parts, and applies convolutional operations with additive connections to capture complex patterns. It also retrieves embeddings for the target tail entities and computes Hermitian inner products to obtain scores, allowing the model to rank the tail entities based on their relevance to the queries.

class dicee.models.complex.**ComplEx**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

ComplEx (Complex Embeddings for Knowledge Graphs) is a model that extends the base knowledge graph embedding approach by using complex-valued embeddings. It emphasizes the interaction of real and imaginary components of embeddings to capture the asymmetric relationships often found in knowledge graphs.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, learning rate, and regularization methods.

name

The name identifier for the ComplEx model.

Type

str

score(head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor,

tail_ent_emb: torch.FloatTensor) -> torch.FloatTensor

Computes the score of a triple using the ComplEx scoring function.

k_vs_all_score(emb_h: torch.FloatTensor, emb_r: torch.FloatTensor,

emb_E: torch.FloatTensor) -> torch.FloatTensor

Computes scores in a K-vs-All setting using complex-valued embeddings.

forward_k_vs_all (*x: torch.LongTensor*) → torch.FloatTensor

Performs a forward pass for K-vs-All scoring, returning scores for all entities.

Notes

ComplEx is particularly suited for modeling asymmetric relations and has been shown to perform well on various knowledge graph benchmarks. The use of complex numbers allows the model to encode additional information compared to real-valued models.

static score (*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Compute the scoring function for a given triple using complex-valued embeddings.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – The complex embedding of the head entity.
- **rel_ent_emb** (*torch.FloatTensor*) – The complex embedding of the relation.
- **tail_ent_emb** (*torch.FloatTensor*) – The complex embedding of the tail entity.

Returns
The score of the triple calculated using the Hermitian dot product of complex embeddings.

Return type
torch.FloatTensor

Notes

The scoring function exploits the complex vector space to model the interactions between entities and relations. It involves element-wise multiplication and summation of real and imaginary parts.

static k_vs_all_score (*emb_h: torch.FloatTensor*, *emb_r: torch.FloatTensor*, *emb_E: torch.FloatTensor*) → torch.FloatTensor

Compute scores for a head entity and relation against all entities in a K-vs-All scenario.

Parameters

- **emb_h** (*torch.FloatTensor*) – The complex embedding of the head entity.
- **emb_r** (*torch.FloatTensor*) – The complex embedding of the relation.
- **emb_E** (*torch.FloatTensor*) – The complex embeddings of all possible tail entities.

Returns
Scores for all possible triples formed with the given head entity and relation.

Return type
torch.FloatTensor

Notes

This method is useful for tasks like link prediction where the model predicts the likelihood of a relation between a given entity pair.

forward_k_vs_all(*x: torch.LongTensor*) → torch.FloatTensor

Perform a forward pass for K-vs-all scoring using complex-valued embeddings.

Parameters
x (*torch.LongTensor*) – Tensor containing indices for head entities and relations.

Returns
Scores for all triples formed with the given head entities and relations against all entities.

Return type
torch.FloatTensor

Notes

This method is typically used in training and evaluation of the model in a link prediction setting, where the goal is to rank all possible tail entities for a given head entity and relation.

dicee.models.function_space

Module Contents

Classes

FMult	FMult is a model for learning neural networks on knowledge graphs. It extends
GFMult	GFMult (Graph Function Multiplication) extends the base knowledge graph embedding
FMult2	FMult2 is a model for learning neural networks on knowledge graphs, offering
LFMult1	Embedding with trigonometric functions. We represent all entities and relations in the complex number space as:
LFMult	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:

class dicee.models.function_space.**FMult**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

FMult is a model for learning neural networks on knowledge graphs. It extends the base knowledge graph embedding model by integrating neural network computations with entity and relation embeddings. The model is designed to work with complex embeddings and utilizes a neural network-based approach for embedding interactions.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions and other model-specific parameters.

name

The name identifier for the FMult model.

Type
str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

k

Dimension size for reshaping weights in neural network layers.

Type
int

num_sample

The number of samples to consider in the model computations.

Type
int

gamma

Randomly initialized weights for the neural network layers.

Type
torch.Tensor

roots

Precomputed roots for Legendre polynomials.

Type
torch.Tensor

weights

Precomputed weights for Legendre polynomials.

Type
torch.Tensor

compute_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.FloatTensor

Computes the output of a two-layer neural network for given weights and input.

chain_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.Tensor

Chains two linear neural network layers for a given input.

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Performs a forward pass for a batch of triples and computes the embedding interactions.

compute_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.FloatTensor

Compute the output of a two-layer neural network.

Parameters

- **weights** (*torch.FloatTensor*) – The weights of the neural network, split into two sets for two layers.

- **x** (*torch.Tensor*) – The input tensor for the neural network.

Returns
The output tensor after passing through the two-layer neural network.

Return type
torch.FloatTensor

chain_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.Tensor

Chain two linear layers of a neural network for given weights and input.

Parameters

- **weights** (*torch.FloatTensor*) – The weights of the neural network, split into two sets for two layers.
- **x** (*torch.Tensor*) – The input tensor for the neural network.

Returns
The output tensor after chaining the two linear layers.

Return type
torch.Tensor

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Forward pass for a batch of triples to compute embedding interactions.

Parameters
idx_triple (*torch.Tensor*) – Tensor containing indices of triples.

Returns
The computed scores for the batch of triples.

Return type
torch.Tensor

class dicee.models.function_space.**GFMult**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

GFMult (Graph Function Multiplication) extends the base knowledge graph embedding model by integrating neural network computations with entity and relation embeddings. This model is designed to leverage the strengths of neural networks in capturing complex interactions within knowledge graphs.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, learning rate, and other model-specific parameters.

name

The name identifier for the GFMult model.

Type
str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

k

The dimension size for reshaping weights in neural network layers.

Type
int

num_sample

The number of samples to use in the model computations.

Type
int

roots

Precomputed roots for Legendre polynomials, repeated for each dimension.

Type
torch.Tensor

weights

Precomputed weights for Legendre polynomials.

Type
torch.Tensor

compute_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.FloatTensor

Computes the output of a two-layer neural network for given weights and input.

chain_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.Tensor

Chains two linear neural network layers for a given input.

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Performs a forward pass for a batch of triples and computes the embedding interactions.

compute_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.FloatTensor

Compute the output of a two-layer neural network.

Parameters

- **weights** (*torch.FloatTensor*) – The weights of the neural network, split into two sets for two layers.
- **x** (*torch.Tensor*) – The input tensor for the neural network.

Returns
The output tensor after passing through the two-layer neural network.

Return type
torch.FloatTensor

chain_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.Tensor

Chain two linear layers of a neural network for given weights and input.

Parameters

- **weights** (*torch.FloatTensor*) – The weights of the neural network, split into two sets for two layers.
- **x** (*torch.Tensor*) – The input tensor for the neural network.

Returns
The output tensor after chaining the two linear layers.

Return type
torch.Tensor

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Forward pass for a batch of triples to compute embedding interactions.

Parameters
idx_triple (*torch.Tensor*) – Tensor containing indices of triples.

Returns
The computed scores for the batch of triples.

Return type
torch.Tensor

class dicee.models.function_space.**FMult2**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

FMult2 is a model for learning neural networks on knowledge graphs, offering enhanced capabilities for capturing complex interactions in the graph. It extends the base knowledge graph embedding model by integrating multi-layer neural network computations with entity and relation embeddings.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, learning rate, number of layers, and other model-specific parameters.

name

The name identifier for the FMult2 model.

Type
str

n_layers

Number of layers in the neural network.

Type
int

k

Dimension size for reshaping weights in neural network layers.

Type
int

n

The number of discrete points for computations.

Type
int

a

Lower bound of the range for discrete points.

Type
float

b

Upper bound of the range for discrete points.

Type
float

score_func

The scoring function used in the model.

Type
str

discrete_points

Tensor of discrete points used in the computations.

Type
torch.Tensor

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

build_func(*Vec: torch.Tensor*) → Tuple[List[torch.Tensor], torch.Tensor]

Constructs a multi-layer neural network from a vector representation.

build_chain_funcs(*list_Vec: List[torch.Tensor]*) → Tuple[List[torch.Tensor], torch.Tensor]

Builds chained functions from a list of vector representations.

compute_func(*W: List[torch.Tensor]*, *b: torch.Tensor*, *x: torch.Tensor*) → torch.FloatTensor

Computes the output of a multi-layer neural network.

function(*list_W: List[List[torch.Tensor]]*, *list_b: List[torch.Tensor]*)
→ Callable[[torch.Tensor], torch.Tensor]

Defines a function for neural network computation based on weights and biases.

trapezoid(*list_W: List[List[torch.Tensor]]*, *list_b: List[torch.Tensor]*) → torch.Tensor

Applies the trapezoidal rule for integration on the function output.

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Performs a forward pass for a batch of triples and computes the embedding interactions.

build_func(*Vec: torch.Tensor*) → Tuple[List[torch.Tensor], torch.Tensor]

Constructs a multi-layer neural network from a vector representation.

Parameters
Vec (*torch.Tensor*) – The vector representation from which the neural network is constructed.

Returns
A tuple containing the list of weight matrices for each layer and the bias vector.

Return type
Tuple[List[torch.Tensor], torch.Tensor]

build_chain_funcs (*list_Vec: List[torch.Tensor]*) → Tuple[List[torch.Tensor], torch.Tensor]

Builds chained functions from a list of vector representations. This method constructs a sequence of neural network layers and their corresponding biases based on the provided vector representations.

Each vector representation in the list is first transformed into a set of weights and biases for a neural network layer using the *build_func* method. The method then computes a chained multiplication of these weights, adjusted by biases, to form a composite neural network function.

Parameters

list_Vec (*List[torch.Tensor]*) – A list of vector representations, each corresponding to a set of parameters for constructing a neural network layer.

Returns

A tuple where the first element is a list of weight tensors for each layer of the composite neural network, and the second element is the bias tensor for the last layer in the list.

Return type

Tuple[List[torch.Tensor], torch.Tensor]

Notes

This method is specifically designed to work with the neural network architecture defined in the FMult2 model. It assumes that each vector in *list_Vec* can be decomposed into weights and biases suitable for a layer in a neural network.

compute_func (*W: List[torch.Tensor]*, *b: torch.Tensor*, *x: torch.Tensor*) → torch.FloatTensor

Computes the output of a multi-layer neural network defined by the given weights and bias.

This method sequentially applies a series of matrix multiplications and non-linear transformations to an input tensor *x*, using the provided weights *W*. The method alternates between applying a non-linear function (tanh) and a linear transformation to the intermediate outputs. The final output is adjusted with a bias term *b*.

Parameters

- **W** (*List[torch.Tensor]*) – A list of weight tensors for each layer in the neural network. Each tensor in the list represents the weights of a layer.
- **b** (*torch.Tensor*) – The bias tensor to be added to the output of the final layer.
- **x** (*torch.Tensor*) – The input tensor to be processed by the neural network.

Returns

The output tensor after processing by the multi-layer neural network.

Return type

torch.FloatTensor

Notes

The method assumes an odd-indexed layer applies a non-linearity (tanh), while even-indexed layers apply linear transformations. This design choice is based on empirical observations for better performance in the context of the FMult2 model.

function (*list_W: List[List[torch.Tensor]]*, *list_b: List[torch.Tensor]*)
→ Callable[[torch.Tensor], torch.Tensor]

Defines a function that computes the output of a composite neural network. This higher-order function returns a callable that applies a sequence of transformations defined by the provided weights and biases.

The returned function (f) takes an input tensor x and applies a series of neural network computations on it. If only one set of weights and biases is provided, it directly computes the output using *compute_func*. Otherwise, it sequentially multiplies the outputs of multiple calls to *compute_func*, each using a different set of weights and biases from *list_W* and *list_b*.

Parameters

- **list_W** (*List[List[torch.Tensor]]*) – A list where each element is a list of weight tensors for a neural network.
- **list_b** (*List[torch.Tensor]*) – A list of bias tensors corresponding to each set of weights in *list_W*.

Returns

A function that takes an input tensor and returns the output of the composite neural network.

Return type

Callable[[torch.Tensor], torch.Tensor]

Notes

This method is part of the FMult2 model's approach to construct complex scoring functions for knowledge graph embeddings. The flexibility in combining multiple neural network layers enables capturing intricate patterns in the data.

trapezoid(*list_W: List[List[torch.Tensor]]*, *list_b: List[torch.Tensor]*) → torch.Tensor

Computes the integral of the output of a composite neural network function over a range of discrete points using the trapezoidal rule.

This method first constructs a composite neural network function using the *function* method with the provided weights *list_W* and biases *list_b*. It then evaluates this function at a series of discrete points (*self.discrete_points*) and applies the trapezoidal rule to approximate the integral of the function over these points. The sum of the integral approximations across all dimensions is returned.

Parameters

- **list_W** (*List[List[torch.Tensor]]*) – A list where each element is a list of weight tensors for a neural network.
- **list_b** (*List[torch.Tensor]*) – A list of bias tensors corresponding to each set of weights in *list_W*.

Returns

The sum of the integral of the composite function's output over the range of discrete points, computed using the trapezoidal rule.

Return type

torch.Tensor

Notes

The trapezoidal rule is a numerical method to approximate definite integrals. In the context of the FMult2 model, this method is used to integrate the output of the neural network over a range of inputs, which is crucial for certain types of calculations in knowledge graph embeddings.

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Forward pass for a batch of triples to compute embedding interactions.

Parameters
idx_triple (*torch.Tensor*) – Tensor containing indices of triples.

Returns
The computed scores for the batch of triples.

Return type
torch.Tensor

class dicee.models.function_space.**LFMult1**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Embedding with trigonometric functions. We represent all entities and relations in the complex number space as: f(x) = sum_{k=0}^{k=d-1}wk e^{kix}. and use the three differents scoring function as in the paper to evaluate the score

forward_triples(*idx_triple*)

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

tri_score(*h*, *r*, *t*)

vtp_score(*h*, *r*, *t*)

class dicee.models.function_space.**LFMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Embedding with polynomial functions. We represent all entities and relations in the polynomial space as: f(x) = sum_{i=0}^{d-1} a_k x^{i%d} and use the three differents scoring function as in the paper to evaluate the score. We also consider combining with Neural Networks.

forward_triples(*idx_triple*)

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

construct_multi_coeff(*x*)

poly_NN(*x*, *coefh*, *coefr*, *coeft*)

Constructing a 2 layers NN to represent the embeddings. h = sigma(wh^T x + bh), r = sigma(wr^T x + br), t = sigma(wt^T x + bt)

linear(*x*, *w*, *b*)

scalar_batch_NN(*a*, *b*, *c*)

element wise multiplication between a,b and c: Inputs : a, b, c ====> torch.tensor of size batch_size x m x d Output : a tensor of size batch_size x d

tri_score(*coeff_h*, *coeff_r*, *coeff_t*)

this part implement the trilinear scoring techniques:

score(h,r,t) = int_{0}{1} h(x)r(x)t(x) dx = sum_{i,j,k = 0}^{d-1} dfrac{a_i*b_j*c_k}{1+(i+j+k)%d}

1. generate the range for i,j and k from [0 d-1]
2. perform dfrac{a_i*b_j*c_k}{1+(i+j+k)%d} in parallel for every batch
3. take the sum over each batch

vtp_score(*h*, *r*, *t*)

this part implement the vector triple product scoring techniques:

score(h,r,t) = int_{0}{1} h(x)r(x)t(x) dx = sum_{i,j,k = 0}^{d-1} dfrac{a_i*c_j*b_k - b_i*c_j*a_k}{(1+(i+j)%d)(1+k)}

1. generate the range for i,j and k from [0 d-1]
2. Compute the first and second terms of the sum
3. Multiply with then denominator and take the sum
4. take the sum over each batch

comp_func(*h*, *r*, *t*)

this part implement the function composition scoring techniques: i.e. score = <hor, t>

polynomial(*coeff*, *x*, *degree*)

This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,…d] and return a vector tensor (coeff[0][0] + coeff[0][1]x +…+ coeff[0][d]x^d,

coeff[1][0] + coeff[1][1]x +…+ coeff[1][d]x^d)

pop(*coeff*, *x*, *degree*)

This function allow us to evaluate the composition of two polynomes without for loops :) it takes a matrix tensor of coefficients (coeff), a matrix tensor of points x and range of integer [0,1,…d]

and return a tensor (coeff[0][0] + coeff[0][1]x +…+ coeff[0][d]x^d,

coeff[1][0] + coeff[1][1]x +…+ coeff[1][d]x^d)

dicee.models.octonion

Module Contents

Classes

OMult	OMult extends the base knowledge graph embedding model by integrating octonion
ConvO	ConvO extends the base knowledge graph embedding model by integrating convolutional
AConvO	Additive Convolutional Octonion(AConvO) extends the base knowledge graph embedding model by integrating additive convolutional

Functions

octonion_mul(→ Tuple[float, float, float, float, ...)	Performs the multiplication of two octonions.
octonion_mul_norm(→ Tuple[float, float, float, float, ...)	Performs the normalized multiplication of two octonions.

dicee.models.octonion.**octonion_mul**(*,
O_1: Tuple[float, float, float, float, float, float, float, float],
O_2: Tuple[float, float, float, float, float, float, float, float])
→ Tuple[float, float, float, float, float, float, float, float]

Performs the multiplication of two octonions.

Octonions are an extension of quaternions and are represented here as 8-tuples of floats. This function computes the product of two octonions using their components.

Parameters

- **O_1** (*Tuple[float, float, float, float, float, float, float, float]*) – The first octonion, represented as an 8-tuple of float components.
- **O_2** (*Tuple[float, float, float, float, float, float, float, float]*) – The second octonion, represented as an 8-tuple of float components.

Returns

The product of the two octonions, represented as an 8-tuple of float components.

Return type

Tuple[float, float, float, float, float, float, float, float]

dicee.models.octonion.**octonion_mul_norm**(*,
O_1: Tuple[float, float, float, float, float, float, float, float],
O_2: Tuple[float, float, float, float, float, float, float, float])
→ Tuple[float, float, float, float, float, float, float, float]

Performs the normalized multiplication of two octonions.

This function first normalizes the second octonion to unit length to eliminate the scaling effect and then computes the product of two octonions using their components.

Parameters

- **O_1** (*Tuple[float, float, float, float, float, float, float, float]*) – The first octonion, represented as an 8-tuple of float components.
- **O_2** (*Tuple[float, float, float, float, float, float, float, float]*) – The second octonion, represented as an 8-tuple of float components.

Returns
The product of the two octonions, represented as an 8-tuple of float components.

Return type
Tuple[float, float, float, float, float, float, float, float]

Notes

Normalization may cause NaNs due to floating-point precision issues, especially if the second octonion's magnitude is very small.

class dicee.models.octonion.**OMult**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

OMult extends the base knowledge graph embedding model by integrating octonion algebra. This model leverages the properties of octonions to represent and process the embeddings of entities and relations in a knowledge graph, aiming to capture complex interactions and patterns.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions and learning rate.

name
The name identifier for the OMult model.

Type
str

octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, ..., *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, ...]
Normalizes octonion components to unit length.

score(*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor
Computes the score of a triple using octonion multiplication.

k_vs_all_score(*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*) → torch.FloatTensor
Computes scores in a K-vs-All setting using octonion embeddings.

forward_k_vs_all(*x*) → torch.FloatTensor
Performs a forward pass for K-vs-All scoring, returning scores for all entities.

static octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, *emb_rel_e2: torch.Tensor*, *emb_rel_e3: torch.Tensor*, *emb_rel_e4: torch.Tensor*, *emb_rel_e5: torch.Tensor*, *emb_rel_e6: torch.Tensor*, *emb_rel_e7: torch.Tensor*)
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
Normalizes the components of an octonion.

Each component of the octonion is divided by the square root of the sum of the squares of all components, normalizing it to unit length.

Parameters
- **emb_rel_e0** (*torch.Tensor*) – The eight components of an octonion.

- **emb_rel_e1** (*torch.Tensor*) – The eight components of an octonion.
- **...** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e7** (*torch.Tensor*) – The eight components of an octonion.

Returns
The normalized components of the octonion.

Return type
Tuple[torch.Tensor, …]

score (*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes the score of a triple using octonion multiplication.

The method involves splitting the embeddings into real and imaginary parts, normalizing the relation embeddings, performing octonion multiplication, and then calculating the score based on the inner product.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – Embedding of the head entity.
- **rel_ent_emb** (*torch.FloatTensor*) – Embedding of the relation.
- **tail_ent_emb** (*torch.FloatTensor*) – Embedding of the tail entity.

Returns
The score of the triple.

Return type
torch.FloatTensor

k_vs_all_score (*bpe_head_ent_emb: torch.FloatTensor*, *bpe_rel_ent_emb: torch.FloatTensor*, *E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using octonion embeddings for a batch of head entities and relations.

This method splits the head entity and relation embeddings into their octonion components, normalizes the relation embeddings if necessary, and then applies octonion multiplication. It computes the score by performing an inner product with all tail entity embeddings.

Parameters

- **bpe_head_ent_emb** (*torch.FloatTensor*) – Batched embeddings of head entities, each represented as an octonion.
- **bpe_rel_ent_emb** (*torch.FloatTensor*) – Batched embeddings of relations, each represented as an octonion.
- **E** (*torch.FloatTensor*) – Embeddings of all possible tail entities.

Returns
Scores for all possible triples formed with the given head entities and relations against all entities. The shape of the output is (size of batch, number of entities).

Return type
torch.FloatTensor

Notes

The method is particularly useful in scenarios like link prediction, where the goal is to rank all possible tail entities for a given head entity and relation.

forward_k_vs_all(*x*)

Performs a forward pass for K-vs-All scoring.

TODO: Add mathematical format for sphinx.

Given a head entity and a relation (h,r), this method computes scores for all possible triples, i.e., [score(h,r,x)|x in Entities] => [0.0,0.1,…,0.8], shape=> (1, |Entities|), returning a score for each entity in the knowledge graph.

Parameters
x (*Tensor*) – Tensor containing indices for head entities and relations.

Returns
Scores for all triples formed with the given head entities and relations against all entities.

Return type
torch.FloatTensor

class dicee.models.octonion.**ConvO**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

ConvO extends the base knowledge graph embedding model by integrating convolutional operations with octonion algebra. This model applies convolutional neural networks to octonion-based embeddings, capturing complex interactions in knowledge graphs.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, number of output channels, kernel size, and dropout rates.

name

The name identifier for the ConvO model.

Type
str

conv2d

A 2D convolutional layer used for processing octonion-based embeddings.

Type
torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

norm_fc1

Normalization layer applied after the fully connected layer.

Type
Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

octonion_normalizer(*emb_rel_e0*, *emb_rel_e1*, ..., *emb_rel_e7*)

Normalizes octonion components to unit length.

residual_convolution(*O_1*, *O_2*)

Performs a residual convolution operation on two octonion embeddings.

forward_triples(*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_all(*x: torch.Tensor*)

Computes scores against a sampled subset of entities using convolutional operations.

Notes

ConvO aims to enhance the modeling capabilities of knowledge graph embeddings by adding more complex interaction patterns through convolutional layers, potentially improving performance on tasks like link prediction.

static octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, *emb_rel_e2: torch.Tensor*, *emb_rel_e3: torch.Tensor*, *emb_rel_e4: torch.Tensor*, *emb_rel_e5: torch.Tensor*, *emb_rel_e6: torch.Tensor*, *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Normalizes the components of an octonion to unit length.

Each component of the octonion is divided by the square root of the sum of the squares of all components.

Parameters

- **emb_rel_e0** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e1** (*torch.Tensor*) – The eight components of an octonion.
- **...** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e7** (*torch.Tensor*) – The eight components of an octonion.

Returns
The normalized components of the octonion.

Return type
Tuple[torch.Tensor, …]

residual_convolution(
O_1: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
O_2: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Performs a residual convolution operation on two sets of octonion embeddings.

The method combines two octonion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **O_1** (*Tuple[torch.Tensor, ...]*) – The first set of octonion embeddings.
- **O_2** (*Tuple[torch.Tensor, ...]*) – The second set of octonion embeddings.

Returns

The resulting octonion embeddings after the convolutional operation.

Return type

Tuple[torch.Tensor, …]

forward_triples (*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

The method processes head, relation, and tail embeddings using convolutional layers and computes the scores of the triples.

Parameters

x (*torch.Tensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns

Scores for the given batch of triples.

Return type

torch.Tensor

forward_k_vs_all (*x: torch.Tensor*) → torch.Tensor

Given a batch of head entities and relations (h,r), this method computes scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,…,0.8], shape=> (1, **|Entities|**) Given a batch of head entities and relations => shape (size of batch,| Entities|)

Parameters

x (*torch.Tensor*) – A tensor representing a batch of input triples in the form of (head entities, relations).

Returns

Scores for the input triples against all possible tail entities.

Return type

torch.Tensor

Notes

- The input *x* is a tensor of shape (batch_size, 2), where each row represents a pair of head entities and relations.
- **The method follows the following steps:**
  (1) Retrieve embeddings & Apply Dropout & Normalization.
  (2) Split the embeddings into real and imaginary parts.
  (3) Apply convolution operation on the real and imaginary parts.
  (4) Perform quaternion multiplication.

(5) Compute scores for all entities.

The method returns a tensor of shape (batch_size, num_entities) where each row contains scores for each entity in the knowledge graph.

class dicee.models.octonion.**AConvO**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Octonion(AConvO) extends the base knowledge graph embedding model by integrating additive convolutional operations with octonion algebra. This model applies convolutional neural networks to octonion-based embeddings, capturing complex interactions in knowledge graphs.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, number of output channels, kernel size, and dropout rates.

name

The name identifier for the AConvO model.

Type

str

conv2d

A 2D convolutional layer used for processing octonion-based embeddings.

Type

torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type

int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type

torch.nn.Linear

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type

torch.nn.BatchNorm2d

norm_fc1

Normalization layer applied after the fully connected layer.

Type

Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type

torch.nn.Dropout2d

octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, ..., *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, ...]

Normalizes octonion components to unit length.

residual_convolution (*self*, *O_1: Tuple[torch.Tensor, ...]*, *O_2: Tuple[torch.Tensor, ...]*) → Tuple[torch.Tensor, ...]

Performs a residual convolution operation on two octonion embeddings.

forward_triples (*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_all (*x: torch.Tensor*)

Computes scores against a sampled subset of entities using convolutional operations.

Notes

AConvO aims to enhance the modeling capabilities of knowledge graph embeddings by adding more complex interaction patterns through convolutional layers, potentially improving performance on tasks like link prediction.

static octonion_normalizer (*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, *emb_rel_e2: torch.Tensor*, *emb_rel_e3: torch.Tensor*, *emb_rel_e4: torch.Tensor*, *emb_rel_e5: torch.Tensor*, *emb_rel_e6: torch.Tensor*, *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Normalizes the components of an octonion to unit length.

Each component of the octonion is divided by the square root of the sum of the squares of all components.

Parameters

- **emb_rel_e0** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e1** (*torch.Tensor*) – The eight components of an octonion.
- **...** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e7** (*torch.Tensor*) – The eight components of an octonion.

Returns

The normalized components of the octonion.

Return type

Tuple[torch.Tensor, …]

residual_convolution (*O_1: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens*
O_2: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Performs a residual convolution operation on two sets of octonion embeddings.

The method combines two octonion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **O_1** (*Tuple[torch.Tensor, ...]*) – The first set of octonion embeddings.
- **O_2** (*Tuple[torch.Tensor, ...]*) – The second set of octonion embeddings.

Returns

The resulting octonion embeddings after the convolutional operation.

Return type

Tuple[torch.Tensor, …]

forward_triples(*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

The method processes head, relation, and tail embeddings using convolutional layers and computes the scores of the triples.

Parameters
x (*torch.Tensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.Tensor

forward_k_vs_all(*x: torch.Tensor*) → torch.Tensor

Compute scores for a head entity and a relation (h,r) against all entities in the knowledge graph.

Given a head entity and a relation (h, r), this method computes scores for (h, r, x) for all entities x in the knowledge graph.

Parameters
x (*torch.Tensor*) – A tensor containing indices for head entities and relations.

Returns
A tensor of scores representing the compatibility of (h, r, x) for all entities x in the knowledge graph.

Return type
torch.Tensor

Notes

This method supports batch processing, allowing the input tensor *x* to contain multiple head entities and relations.

The scores indicate how well each entity x in the knowledge graph fits the (h, r) pattern, with higher scores indicating better compatibility.

dicee.models.pykeen_models

Module Contents

Classes

PykeenKGE	A class for using knowledge graph embedding models implemented in Pykeen.

class dicee.models.pykeen_models.**PykeenKGE**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

A class for using knowledge graph embedding models implemented in Pykeen.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, random seed, and model-specific kwargs.

name

The name identifier for the PykeenKGE model.

Type
str

model

The Pykeen model instance.

Type
pykeen.models.base.Model

loss_history

A list to store the training loss history.

Type
list

args

The arguments used to initialize the model.

Type
dict

entity_embeddings

Entity embeddings learned by the model.

Type
torch.nn.Embedding

relation_embeddings

Relation embeddings learned by the model.

Type
torch.nn.Embedding

interaction

Interaction module used by the Pykeen model.

Type
pykeen.nn.modules.Interaction

forward_k_vs_all(*x: torch.LongTensor*) → torch.FloatTensor

Compute scores for all entities given a batch of head entities and relations.

forward_triples(*x: torch.LongTensor*) → torch.FloatTensor

Compute scores for a batch of triples.

forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: int*)

Compute scores against a sampled subset of entities.

Notes

This class provides an interface for using knowledge graph embedding models implemented in Pykeen. It initializes Pykeen models based on the provided arguments and allows for scoring triples and conducting knowledge graph embedding experiments.

forward_k_vs_all (*x: torch.LongTensor*)

TODO: Format in Numpy-style documentation

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads and relations + apply Dropout & Normalization if given. h, r = self.get_head_relation_representation(x) # (2) Reshape (1). if self.last_dim > 0:

h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim)

(3) Reshape all entities. if self.last_dim > 0:

t = self.entity_embeddings.weight.reshape(self.num_entities, self.embedding_dim, self.last_dim)

else:

t = self.entity_embeddings.weight

(4) Call the score_t from interactions to generate triple scores. return self.interaction.score_t(h=h, r=r, all_entities=t, slice_size=1)

forward_triples (*x: torch.LongTensor*) → torch.FloatTensor

TODO: Format in Numpy-style documentation

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads, relations and tails and apply Dropout & Normalization if given. h, r, t = self.get_triple_representation(x) # (2) Reshape (1). if self.last_dim > 0:

h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim) t = t.reshape(len(x), self.embedding_dim, self.last_dim)

(3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice_size=None, slice_dim=0)

abstract forward_k_vs_sample (*x: torch.LongTensor*, *target_entity_idx: int*)

Forward pass for K vs. Sample.

Raises

ValueError – This function is not implemented in the current model.

dicee.models.quaternion

Module Contents

Classes

QMult	QMult extends the base knowledge graph embedding model by integrating quaternion
ConvQ	Convolutional Quaternion Knowledge Graph Embeddings (ConvQ) is a model that extends
AConvQ	Additive Convolutional Quaternion Knowledge Graph Embeddings (AConvQ) model integrates

Functions

quaternion_mul_with_unit_norm(→ Tuple[float, float, ...)	Performs the multiplication of two quaternions with unit norm.

dicee.models.quaternion.**quaternion_mul_with_unit_norm**(*, *Q_1: Tuple[float, float, float, float]*, *Q_2: Tuple[float, float, float, float]*) → Tuple[float, float, float, float]

Performs the multiplication of two quaternions with unit norm.

Parameters

- **Q_1** (*Tuple[float, float, float, float]*) – The first quaternion represented as a tuple of four real numbers (a_h, b_h, c_h, d_h).
- **Q_2** (*Tuple[float, float, float, float]*) – The second quaternion represented as a tuple of four real numbers (a_r, b_r, c_r, d_r).

Returns

The result of the quaternion multiplication, represented as a tuple of four real numbers (r_val, i_val, j_val, k_val).

Return type

Tuple[float, float, float, float]

Notes

The function assumes that the input quaternions have unit norm. It first normalizes the second quaternion to eliminate the scaling effect, and then performs the Hamilton product of the two quaternions.

class dicee.models.quaternion.**QMult**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

QMult extends the base knowledge graph embedding model by integrating quaternion algebra. This model leverages the properties of quaternions to represent and process the embeddings of entities and relations in a knowledge graph, aiming to capture complex interactions and patterns.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions and learning rate.

name

The name identifier for the QMult model.

Type
str

quaternion_normalizer (*x: torch.FloatTensor*) → torch.FloatTensor

Normalizes the length of relation vectors.

score (*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes the score of a triple using quaternion multiplication.

k_vs_all_score (*bpe_head_ent_emb: torch.FloatTensor*, *bpe_rel_ent_emb: torch.FloatTensor*, *E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using quaternion embeddings.

forward_k_vs_all (*x: torch.FloatTensor*) → torch.FloatTensor

Performs a forward pass for K-vs-All scoring, returning scores for all entities.

forward_k_vs_sample (*x: torch.FloatTensor*, *target_entity_idx: int*) → torch.FloatTensor

Performs a forward pass for K-vs-Sample scoring, returning scores for the specified entities.

quaternion_multiplication_followed_by_inner_product (*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Performs quaternion multiplication followed by inner product, returning triple scores.

quaternion_multiplication_followed_by_inner_product (*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Performs quaternion multiplication followed by inner product.

Parameters

- **h** (*torch.FloatTensor*) – The head representations. Shape: (**batch_dims*, dim)
- **r** (*torch.FloatTensor*) – The relation representations. Shape: (**batch_dims*, dim)
- **t** (*torch.FloatTensor*) – The tail representations. Shape: (**batch_dims*, dim)

Returns
Triple scores.

Return type
torch.FloatTensor

static quaternion_normalizer (*x: torch.FloatTensor*) → torch.FloatTensor

TODO: Add mathematical format for sphinx. Normalize the length of relation vectors, if the forward constraint has not been applied yet.

The absolute value of a quaternion is calculated as follows: .. math:

```
|a + bi + cj + dk| = \sqrt{a^2 + b^2 + c^2 + d^2}
```

The L2 norm of a quaternion vector is computed as: .. math:

```
\|x\|^2 = \sum_{i=1}^d |x_i|^2
        = \sum_{i=1}^d (x_i.re^2 + x_i.im_1^2 + x_i.im_2^2 + x_i.im_3^2)
```

Parameters
x (*torch.FloatTensor*) – The vector containing quaternion values.

Returns
The normalized vector.

Return type
torch.FloatTensor

Notes

This function normalizes the length of relation vectors represented as quaternions. It ensures that the absolute value of each quaternion in the vector is equal to 1, preserving the unit length.

score(*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Compute scores for a batch of triples using octonion-based embeddings.

This method computes scores for a batch of triples using octonion-based embeddings of head entities, relation embeddings, and tail entities. It supports both explicit and non-explicit scoring methods.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – Tensor containing the octonion-based embeddings of head entities.
- **rel_ent_emb** (*torch.FloatTensor*) – Tensor containing the octonion-based embeddings of relations.
- **tail_ent_emb** (*torch.FloatTensor*) – Tensor containing the octonion-based embeddings of tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.FloatTensor

Notes

If no normalization is set, this method applies quaternion normalization to relation embeddings.

If the scoring method is explicit, it computes the scores using quaternion multiplication followed by an inner product of the real and imaginary parts of the resulting quaternions.

If the scoring method is non-explicit, it directly computes the inner product of the real and imaginary parts of the octonion-based embeddings.

k_vs_all_score(*bpe_head_ent_emb: torch.FloatTensor*, *bpe_rel_ent_emb: torch.FloatTensor*, *E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using quaternion embeddings for a batch of head entities and relations.

This method involves splitting the head entity and relation embeddings into quaternion components, optionally normalizing the relation embeddings, performing quaternion multiplication, and then calculating the score by performing an inner product with all tail entity embeddings.

Parameters

- **bpe_head_ent_emb** (*torch.FloatTensor*) – Batched embeddings of head entities, each represented as a quaternion.
- **bpe_rel_ent_emb** (*torch.FloatTensor*) – Batched embeddings of relations, each represented as a quaternion.
- **E** (*torch.FloatTensor*) – Embeddings of all possible tail entities.

Returns
Scores for all possible triples formed with the given head entities and relations against all entities. The shape of the output is (size of batch, number of entities).

Return type
torch.FloatTensor

Notes

The method is particularly useful in scenarios like link prediction, where the goal is to rank all possible tail entities for a given head entity and relation. Quaternion algebra is used to enhance the interaction modeling between entities and relations.

forward_k_vs_all (*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

This method retrieves embeddings for the head entities and relations from the input tensor *x*, applies necessary dropout and normalization, and then uses the *k_vs_all_score* method to compute the scores against all possible tail entities in the knowledge graph.

Parameters
x (*torch.FloatTensor*) – A tensor containing indices for head entities and relations. The tensor is expected to have a specific format suitable for the model's embedding retrieval process.

Returns
A tensor of scores, where each row corresponds to the scores of all tail entities for a single head entity and relation pair. The shape of the tensor is (size of the batch, number of entities).

Return type
torch.FloatTensor

Notes

This method is typically used in evaluating the model's performance in link prediction tasks, where it's important to rank the likelihood of every possible tail entity for a given head entity and relation.

forward_k_vs_sample (*x: torch.FloatTensor*, *target_entity_idx: int*) → torch.FloatTensor

Computes scores for a batch of triples against a sampled subset of entities in a K-vs-Sample setting.

Given a batch of head entities and relations (h,r), this method computes the scores for all possible triples formed with these head entities and relations against a subset of entities, i.e., [score(h,r,x)|x in Entities] => [0.0,0.1,…,0.8], shape=> (1, **|Entities|**). TODO: Add mathematical format for sphinx. The subset of entities is specified by the *target_entity_idx*, which is an integer index representing a specific entity. Given a batch of head entities and relations => shape (size of batch,| Entities|).

Parameters

- **x** (*torch.FloatTensor*) – A tensor containing indices for head entities and relations. The tensor is expected to have a specific format suitable for the model's embedding retrieval process.
- **target_entity_idx** (*int*) – Index of the target entity against which the scores are to be computed.

Returns
A tensor of scores where each element corresponds to the score of the target entity for a single head entity and relation pair. The shape of the tensor is (size of the batch, 1).

Return type
torch.FloatTensor

Notes

This method is particularly useful in scenarios like link prediction, where it's necessary to evaluate the likelihood of a specific relationship between a given head entity and a particular target entity.

class dicee.models.quaternion.**ConvQ**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings (ConvQ) is a model that extends the base knowledge graph embedding approach by using quaternion algebra and convolutional neural networks. This model aims to capture complex interactions in knowledge graphs by applying convolutions to quaternion-based entity and relation embeddings.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, number of output channels, kernel size, and dropout rates.

name

The name identifier for the ConvQ model.

Type
str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

conv2d

A 2D convolutional layer used for processing quaternion embeddings.

Type
torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

bn_conv1

First batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

bn_conv2

Second normalization layer applied after the fully connected layer.

Type
Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(*Q_1*, *Q_2*)

Performs a residual convolution operation on two sets of quaternion embeddings.

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on quaternion embeddings.

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

Notes

ConvQ leverages the properties of quaternions, a number system that extends complex numbers, to represent and process the embeddings of entities and relations. The convolutional layers aim to capture spatial relationships and complex patterns in the embeddings.

residual_convolution(
Q_1: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor],
Q_2: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor])
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Performs a residual convolution operation on two sets of quaternion embeddings.

The method combines two quaternion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **Q_1** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The first set of quaternion embeddings.
- **Q_2** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The second set of quaternion embeddings.

Returns
The resulting quaternion embeddings after the convolutional operation.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on quaternion embeddings.

The method processes head, relation, and tail embeddings using quaternion algebra and convolutional layers and computes the scores of the triples.

Parameters
indexed_triple (*torch.FloatTensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.FloatTensor

forward_k_vs_all (*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

This method retrieves embeddings for the head entities and relations from the input tensor *x*, applies necessary dropout and normalization, and then computes scores against all entities in the knowledge graph.

Parameters
x (*torch.FloatTensor*) – A tensor containing indices for head entities and relations.

Returns
Scores for all entities for the given batch of head entities and relations.

Return type
torch.FloatTensor

class dicee.models.quaternion.**AConvQ**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Quaternion Knowledge Graph Embeddings (AConvQ) model integrates quaternion algebra with convolutional neural networks for knowledge graph embeddings. This model is designed to capture complex interactions in knowledge graphs by applying additive convolutions to quaternion-based entity and relation embeddings.

name
The name identifier for the AConvQ model.

Type
str

entity_embeddings
Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings
Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

conv2d
A 2D convolutional layer used for processing quaternion embeddings.

Type
torch.nn.Conv2d

fc_num_input
The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

bn_conv1

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

bn_conv2

Normalization layer applied after the fully connected layer.

Type
Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(*Q_1*, *Q_2*)

Performs an additive residual convolution operation on two sets of quaternion embeddings.

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using additive convolutional operations on quaternion embeddings.

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

residual_convolution(
Q_1: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor],
Q_2: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor])
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Performs a residual convolution operation on two sets of quaternion embeddings.

The method combines two quaternion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **Q_1** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The first set of quaternion embeddings.
- **Q_2** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The second set of quaternion embeddings.

Returns
The resulting quaternion embeddings after the convolutional operation.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on quaternion embeddings.

The method processes head, relation, and tail embeddings using quaternion algebra and convolutional layers and computes the scores of the triples.

Parameters
indexed_triple (*torch.FloatTensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.FloatTensor

forward_k_vs_all (*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

This method retrieves embeddings for the head entities and relations from the input tensor *x*, applies necessary dropout and normalization, and then computes scores against all entities in the knowledge graph.

Parameters
x (*torch.FloatTensor*) – A tensor containing indices for head entities and relations.

Returns
Scores for all entities for the given batch of head entities and relations.

Return type
torch.FloatTensor

dicee.models.real

Module Contents

Classes

DistMult	DistMult model for learning and inference in knowledge bases. It represents both entities
TransE	TransE model for learning embeddings in multi-relational data. It is based on the idea of translating
Shallom	Shallom is a shallow neural model designed for relation prediction in knowledge graphs.
Pyke	Pyke is a physical embedding model for knowledge graphs, emphasizing the geometric relationships

class dicee.models.real.**DistMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

DistMult model for learning and inference in knowledge bases. It represents both entities and relations using embeddings and uses a simple bilinear form to compute scores for triples.

This implementation of the DistMult model is based on the paper: 'Embedding Entities and Relations for Learning and Inference in Knowledge Bases' (https://arxiv.org/abs/1412.6575).

name

The name identifier for the DistMult model.

Type
str

k_vs_all_score(*emb_h: torch.FloatTensor*, *emb_r: torch.FloatTensor*, *emb_E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using embeddings for a batch of head entities and relations.

forward_k_vs_all(*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for all entities given a batch of head entities and relations.

forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Computes scores for a sampled subset of entities given a batch of head entities and relations.

score(*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Computes the score of triples using DistMult's scoring function.

k_vs_all_score(*emb_h: torch.FloatTensor*, *emb_r: torch.FloatTensor*, *emb_E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using embeddings for a batch of head entities and relations.

This method multiplies the head entity and relation embeddings, applies a dropout and a normalization, and then computes the dot product with all tail entity embeddings.

Parameters

- **emb_h** (*torch.FloatTensor*) – Embeddings of head entities.
- **emb_r** (*torch.FloatTensor*) – Embeddings of relations.
- **emb_E** (*torch.FloatTensor*) – Embeddings of all entities.

Returns

Scores for all possible triples formed with the given head entities and relations against all entities.

Return type

torch.FloatTensor

forward_k_vs_all(*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for all entities given a batch of head entities and relations.

This method is used for K-vs-All scoring, where the model predicts the likelihood of each entity being the tail entity in a triple with each head entity and relation pair in the batch.

Parameters

x (*torch.LongTensor*) – Tensor containing indices for head entities and relations.

Returns

Scores for all entities for each head entity and relation pair in the batch.

Return type

torch.FloatTensor

forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Computes scores for a sampled subset of entities given a batch of head entities and relations.

This method is particularly useful when the full set of entities is too large to score with every batch and only a subset of entities is required.

Parameters

- **x** (*torch.LongTensor*) – Tensor containing indices for head entities and relations.
- **target_entity_idx** (*torch.LongTensor*) – Indices of the target entities against which the scores are to be computed.

Returns
Scores for each head entity and relation pair against the sampled subset of entities.

Return type
torch.FloatTensor

score(*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Computes the score of triples using DistMult's scoring function.

The scoring function multiplies head entity and relation embeddings, applies dropout and normalization, and computes the dot product with the tail entity embeddings.

Parameters

- **h** (*torch.FloatTensor*) – Embedding of the head entity.
- **r** (*torch.FloatTensor*) – Embedding of the relation.
- **t** (*torch.FloatTensor*) – Embedding of the tail entity.

Returns
The score of the triple.

Return type
torch.FloatTensor

class dicee.models.real.**TransE**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

TransE model for learning embeddings in multi-relational data. It is based on the idea of translating embeddings for head entities by the relation vector to approach the tail entity embeddings in the embedding space.

This implementation of TransE is based on the paper: 'Translating Embeddings for Modeling Multi-relational Data' (https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf).

name

The name identifier for the TransE model.

Type
str

_norm

The norm used for computing pairwise distances in the embedding space.

Type
int

margin

The margin value used in the scoring function.

Type
int

score(*head_ent_emb: torch.Tensor*, *rel_ent_emb: torch.Tensor*, *tail_ent_emb: torch.Tensor*) → torch.Tensor

Computes the score of triples using the TransE scoring function.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for all entities given a head entity and a relation.

score (*head_ent_emb: torch.Tensor*, *rel_ent_emb: torch.Tensor*, *tail_ent_emb: torch.Tensor*) → torch.Tensor

Computes the score of triples using the TransE scoring function.

The scoring function computes the L2 distance between the translated head entity and the tail entity embeddings and subtracts this distance from the margin.

Parameters

- **head_ent_emb** (*torch.Tensor*) – Embedding of the head entity.
- **rel_ent_emb** (*torch.Tensor*) – Embedding of the relation.
- **tail_ent_emb** (*torch.Tensor*) – Embedding of the tail entity.

Returns
The score of the triple.

Return type
torch.Tensor

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for all entities given a head entity and a relation.

This method is used for K-vs-All scoring, where the model predicts the likelihood of each entity being the tail entity in a triple with each head entity and relation.

Parameters
x (*torch.Tensor*) – Tensor containing indices for head entities and relations.

Returns
Scores for all entities for each head entity and relation pair.

Return type
torch.FloatTensor

class dicee.models.real.**Shallom**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

Shallom is a shallow neural model designed for relation prediction in knowledge graphs. The model combines entity embeddings and passes them through a neural network to predict the likelihood of different relations. It's based on the paper: 'A Shallow Neural Model for Relation Prediction' (https://arxiv.org/abs/2101.09090).

name

The name identifier for the Shallom model.

Type
str

shallom

A sequential neural network model used for predicting relations.

Type
torch.nn.Sequential

get_embeddings () → Tuple[np.ndarray, None]

Retrieves the entity embeddings.

forward_k_vs_all (*x*) → torch.FloatTensor

Computes relation scores for all pairs of entities in the batch.

forward_triples(*x*) → torch.FloatTensor

Computes relation scores for a batch of triples.

get_embeddings() → Tuple[numpy.ndarray, None]

Retrieves the entity embeddings from the model.

Returns
A tuple containing the entity embeddings as a NumPy array and None for the relation embeddings.

Return type
Tuple[np.ndarray, None]

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes relation scores for all pairs of entities in the batch.

Each pair of entities is passed through the Shallom neural network to predict the likelihood of various relations between them.

Parameters
x (*torch.Tensor*) – A tensor of entity pairs.

Returns
A tensor of relation scores for each pair of entities in the batch.

Return type
torch.FloatTensor

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes relation scores for a batch of triples.

This method first computes relation scores for all possible relations for each pair of entities and then selects the scores corresponding to the actual relations in the triples.

Parameters
x (*torch.Tensor*) – A tensor containing a batch of triples.

Returns
A flattened tensor of relation scores for the given batch of triples.

Return type
torch.FloatTensor

class dicee.models.real.**Pyke**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

Pyke is a physical embedding model for knowledge graphs, emphasizing the geometric relationships in the embedding space. The model aims to represent entities and relations in a way that captures the underlying structure of the knowledge graph.

name

The name identifier for the Pyke model.

Type
str

dist_func

A pairwise distance function to compute distances in the embedding space.

Type
torch.nn.PairwiseDistance

margin

The margin value used in the scoring function.

Type
float

forward_triples(*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples based on the physical embedding approach.

forward_triples(*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples based on the physical embedding approach.

The method calculates the Euclidean distance between the head and relation embeddings, and between the relation and tail embeddings. The average of these distances is subtracted from the margin to compute the score for each triple.

Parameters
x (*torch.LongTensor*) – A tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples. Lower scores indicate more likely triples according to the geometric arrangement of embeddings.

Return type
torch.FloatTensor

dicee.models.static_funcs

Module Contents

Functions

quaternion_mul(→ torch.Tensor, ...)	Tuple[torch.Tensor,	Perform quaternion multiplication.

dicee.models.static_funcs.**quaternion_mul**(*,
Q_1: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor],
Q_2: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor])
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Perform quaternion multiplication.

This function multiplies two quaternions, Q_1 and Q_2, and returns the result as a quaternion. Quaternion multiplication is a non-commutative operation used in various applications, including 3D rotation and orientation tasks.

Parameters

- **Q_1** (*Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]*) – The first quaternion, represented as a tuple of four components (a_h, b_h, c_h, d_h).
- **Q_2** (*Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]*) – The second quaternion, represented as a tuple of four components (a_r, b_r, c_r, d_r).

Returns

The resulting quaternion from the multiplication, represented as a tuple of four components (r_val, i_val, j_val, k_val).

Return type

Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Notes

The quaternion multiplication is defined as: r_val = a_h * a_r - b_h * b_r - c_h * c_r - d_h * d_r i_val = a_h * b_r + b_h * a_r + c_h * d_r - d_h * c_r j_val = a_h * c_r - b_h * d_r + c_h * a_r + d_h * b_r k_val = a_h * d_r + b_h * c_r - c_h * b_r + d_h * a_r

dicee.models.transformers

Module Contents

Classes

BytE	Base class for all neural network modules.
LayerNorm	LayerNorm but with an optional bias. PyTorch doesn't support simply bias=False
CausalSelfAttention	Base class for all neural network modules.
MLP	Base class for all neural network modules.
Block	Base class for all neural network modules.
GPTConfig	
GPT	Base class for all neural network modules.

class dicee.models.transformers.**BytE**(**args*, ***kwargs*)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training` (*`bool`*) – Boolean represents whether this module is in training or evaluation mode.

`loss_function`(*yhat_batch*, *y_batch*)

Parameters

- **`yhat_batch`** –
- **`y_batch`** –

`forward`(*x: torch.LongTensor*)

Parameters

`x` (*`B by T tensor`*) –

`generate`(*idx*, *max_new_tokens*, *temperature=1.0*, *top_k=None*)

Take a conditioning sequence of indices idx (LongTensor of shape (b,t)) and complete the sequence max_new_tokens times, feeding the predictions back into the model each time. Most likely you'll want to make sure to be in model.eval() mode of operation for this.

`training_step`(*batch*, *batch_idx=None*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **`batch`** – The output of your data iterable, normally a `DataLoader`.
- **`batch_idx`** – The index of this batch.
- **`dataloader_idx`** – The index of the dataloader that produced this batch. (only if multiple dataloaders used)

Returns

- `Tensor` - The loss tensor
- `dict` - A dictionary which can include any keys, but must include the key `'loss'` in the case of automatic optimization.
- `None` - In automatic optimization, this will skip to the next batch (but is not supported for multi-GPU, TPU, or DeepSpeed). For manual optimization, this has no special meaning, as returning the loss is not required.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

To use multiple optimizers, you can switch to 'manual optimization' and control their stepping:

```
def __init__(self):
    super().__init__()
    self.automatic_optimization = False


# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

    # do training_step with encoder
    ...
    opt1.step()
    # do training_step with decoder
    ...
    opt2.step()
```

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

class dicee.models.transformers.**LayerNorm**(*ndim*, *bias*)

Bases: `torch.nn.Module`

LayerNorm but with an optional bias. PyTorch doesn't support simply bias=False

forward(*input*)

class dicee.models.transformers.**CausalSelfAttention**(*config*)

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables
training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward(*x*)

class dicee.models.transformers.**MLP**(*config*)

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables
training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward(*x*)

class dicee.models.transformers.**Block**(*config*)

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward(*x*)

class dicee.models.transformers.**GPTConfig**

block_size: int = 1024

vocab_size: int = 50304

n_layer: int = 12

n_head: int = 12

n_embd: int = 768

dropout: float = 0.0

bias: bool = False

class dicee.models.transformers.**GPT**(*config*)

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables
training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

get_num_params(*non_embedding=True*)

Return the number of parameters in the model. For non-embedding count (default), the position embeddings get subtracted. The token embeddings would too, except due to the parameter sharing these params are actually used as weights in the final layer, so we include them.

forward(*idx*, *targets=None*)

crop_block_size(*block_size*)

classmethod from_pretrained(*model_type*, *override_args=None*)

configure_optimizers(*weight_decay*, *learning_rate*, *betas*, *device_type*)

estimate_mfu(*fwdbwd_per_iter*, *dt*)

estimate model flops utilization (MFU) in units of A100 bfloat16 peak FLOPS

Package Contents

Classes

BaseKGELightning	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
IdentityClass	A class that represents an identity function.
BaseKGE	Base class for all neural network modules.
DistMult	DistMult model for learning and inference in knowledge bases. It represents both entities
TransE	TransE model for learning embeddings in multi-relational data. It is based on the idea of translating
Shallom	Shallom is a shallow neural model designed for relation prediction in knowledge graphs.
Pyke	Pyke is a physical embedding model for knowledge graphs, emphasizing the geometric relationships
BaseKGE	Base class for all neural network modules.
ConEx	ConEx (Convolutional ComplEx) is a Knowledge Graph Embedding model that extends ComplEx embeddings with convolutional layers.
AConEx	AConEx (Additive Convolutional ComplEx) extends the ConEx model by incorporating
ComplEx	ComplEx (Complex Embeddings for Knowledge Graphs) is a model that extends
BaseKGE	Base class for all neural network modules.

continues on next page

Table 1 – continued from previous page

IdentityClass	A class that represents an identity function.
QMult	QMult extends the base knowledge graph embedding model by integrating quaternion
ConvQ	Convolutional Quaternion Knowledge Graph Embeddings (ConvQ) is a model that extends
AConvQ	Additive Convolutional Quaternion Knowledge Graph Embeddings (AConvQ) model integrates
BaseKGE	Base class for all neural network modules.
IdentityClass	A class that represents an identity function.
OMult	OMult extends the base knowledge graph embedding model by integrating octonion
ConvO	ConvO extends the base knowledge graph embedding model by integrating convolutional
AConvO	Additive Convolutional Octonion(AConvO) extends the base knowledge graph embedding model by integrating additive convolutional
Keci	The Keci class is a knowledge graph embedding model that incorporates Clifford algebra for embeddings.
KeciBase	Without learning dimension scaling
CMult	The CMult class represents a specific kind of mathematical object used in knowledge graph embeddings,
DeCaL	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
PykeenKGE	A class for using knowledge graph embedding models implemented in Pykeen.
BaseKGE	Base class for all neural network modules.
FMult	FMult is a model for learning neural networks on knowledge graphs. It extends
GFMult	GFMult (Graph Function Multiplication) extends the base knowledge graph embedding
FMult2	FMult2 is a model for learning neural networks on knowledge graphs, offering
LFMult1	Embedding with trigonometric functions. We represent all entities and relations in the complex number space as:
LFMult	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:

Functions

quaternion_mul(→ Tuple[torch.Tensor, torch.Tensor, ...)	Perform quaternion multiplication.
quaternion_mul_with_unit_norm(→ Tuple[float, float, ...)	Performs the multiplication of two quaternions with unit norm.
octonion_mul(→ Tuple[float, float, float, float, ...)	Performs the multiplication of two octonions.
octonion_mul_norm(→ Tuple[float, float, float, float, ...)	Performs the normalized multiplication of two octonions.

class dicee.models.**BaseKGELightning**(**args*, ***kwargs*)

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

mem_of_model() → Dict

Size of model in MB and number of params

training_step(*batch*, *batch_idx=None*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** – The output of your data iterable, normally a `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple dataloaders used)

Returns

- `Tensor` - The loss tensor
- `dict` - A dictionary which can include any keys, but must include the key `'loss'` in the case of automatic optimization.
- `None` - In automatic optimization, this will skip to the next batch (but is not supported for multi-GPU, TPU, or DeepSpeed). For manual optimization, this has no special meaning, as returning the loss is not required.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

To use multiple optimizers, you can switch to 'manual optimization' and control their stepping:

```
def __init__(self):
    super().__init__()
    self.automatic_optimization = False


# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

    # do training_step with encoder
    ...
    opt1.step()
    # do training_step with decoder
    ...
    opt2.step()
```

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

loss_function(*yhat_batch: torch.FloatTensor*, *y_batch: torch.FloatTensor*)

Parameters

- **yhat_batch** –
- **y_batch** –

on_train_epoch_end(**args*, ***kwargs*)

Called in the training loop at the very end of the epoch.

To access all batch outputs at the end of the epoch, you can cache step outputs as an attribute of the `LightningModule` and access them in this hook:

```
class MyLightningModule(L.LightningModule):
    def __init__(self):
        super().__init__()
        self.training_step_outputs = []

    def training_step(self):
        loss = ...
        self.training_step_outputs.append(loss)
        return loss

    def on_train_epoch_end(self):
        # do something with all training_step outputs, for example:
        epoch_mean = torch.stack(self.training_step_outputs).mean()
        self.log("training_epoch_mean", epoch_mean)
        # free up the memory
        self.training_step_outputs.clear()
```

test_epoch_end(*outputs: List[Any]*)

test_dataloader() → None

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `test()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Note: If you don't need a test dataset and a `test_step()`, you don't need to implement this method.

---

val_dataloader() → None

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:paramref:`~lightning.pytorch.trainer.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `fit()`
- `validate()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Note: If you don't need a validation dataset and a `validation_step()`, you don't need to implement this method.

---

predict_dataloader() → None

An iterable or collection of iterables specifying prediction samples.

For more information about multiple dataloaders, see this section.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `predict()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Returns

A `torch.utils.data.DataLoader` or a sequence of them specifying prediction samples.

train_dataloader() → None

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:paramref:`~lightning.pytorch.trainer.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

---

configure_optimizers(*parameters=None*)

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple. Optimization with multiple optimizers only works in the manual optimization mode.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` automatically in case of automatic optimization.
- If a learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizer.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, you will have to switch to 'manual optimization' mode and step them yourself.

- If you need to control how often the optimizer steps, override the `optimizer_step()` hook.

---

class dicee.models.**BaseKGE**(*args: dict*)

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_byte_pair_encoded_k_vs_all(*x: torch.LongTensor*)

Parameters

x (*B x 2 x T*) –

forward_byte_pair_encoded_triple(*x: Tuple[torch.LongTensor, torch.LongTensor]*)

Perform the forward pass for byte pair encoded triples.

Parameters

x (*Tuple[torch.LongTensor, torch.LongTensor]*) – The input tuple containing byte pair encoded entities and relations.

Returns

The output tensor containing the scores for the byte pair encoded triples.

Return type

torch.Tensor

init_params_with_sanity_checking()

forward(*x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor]*, *y_idx: torch.LongTensor = None*)

Perform the forward pass of the model.

Parameters

- **x** (*Union[torch.LongTensor, Tuple[torch.LongTensor, torch.LongTensor]]*) – The input tensor or a tuple containing the input tensor and target entity indexes.
- **y_idx** (*torch.LongTensor, optional*) – The target entity indexes (default is None).

Returns
The output of the forward pass.

Return type
Any

forward_triples(*x: torch.LongTensor*) → torch.Tensor

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

forward_k_vs_all(**args*, ***kwargs*)

Forward pass for K vs. All.

Raises
ValueError – This function is not implemented in the current model.

forward_k_vs_sample(**args*, ***kwargs*)

Forward pass for K vs. Sample.

Raises
ValueError – This function is not implemented in the current model.

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for the head and relation entities.

Parameters
indexed_triple (*torch.LongTensor*) – The indexes of the head and relation entities.

Returns
The representation for the head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_sentence_representation(*x: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Get the representation for a sentence.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The representation for the input sentence.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

get_bpe_head_and_relation_representation(*x: torch.LongTensor*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for BPE head and relation entities.

Parameters
x (*B x 2 x T*) –

Returns
The representation for BPE head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]

Get the entity and relation embeddings.

Returns
The entity and relation embeddings.

Return type
Tuple[np.ndarray, np.ndarray]

class dicee.models.**IdentityClass**(*args: Dict | None = None*)

Bases: torch.nn.Module

A class that represents an identity function.

Parameters
args (*dict, optional*) – A dictionary containing arguments (default is None).

__call__(*x*)

static forward(*x: torch.Tensor*) → torch.Tensor

The forward pass of the identity function.

Parameters
x (*torch.Tensor*) – The input tensor.

Returns
The output tensor, which is the same as the input.

Return type
torch.Tensor

class dicee.models.**BaseKGE**(*args: dict*)

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables
training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_byte_pair_encoded_k_vs_all(*x: torch.LongTensor*)

Parameters
x (*B x 2 x T*) –

forward_byte_pair_encoded_triple(*x: Tuple[torch.LongTensor, torch.LongTensor]*)

Perform the forward pass for byte pair encoded triples.

Parameters
x (*Tuple[torch.LongTensor, torch.LongTensor]*) – The input tuple containing byte pair encoded entities and relations.

Returns
The output tensor containing the scores for the byte pair encoded triples.

Return type
torch.Tensor

init_params_with_sanity_checking()

forward(*x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor]*, *y_idx: torch.LongTensor = None*)

Perform the forward pass of the model.

Parameters

- **x** (*Union[torch.LongTensor, Tuple[torch.LongTensor, torch.LongTensor]]*) – The input tensor or a tuple containing the input tensor and target entity indexes.
- **y_idx** (*torch.LongTensor, optional*) – The target entity indexes (default is None).

Returns
The output of the forward pass.

Return type
Any

forward_triples(*x: torch.LongTensor*) → torch.Tensor

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

forward_k_vs_all(**args*, ***kwargs*)

Forward pass for K vs. All.

Raises
ValueError – This function is not implemented in the current model.

forward_k_vs_sample(**args*, ***kwargs*)

Forward pass for K vs. Sample.

Raises
ValueError – This function is not implemented in the current model.

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for the head and relation entities.

Parameters
indexed_triple (*torch.LongTensor*) – The indexes of the head and relation entities.

Returns
The representation for the head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_sentence_representation(*x: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Get the representation for a sentence.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The representation for the input sentence.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

get_bpe_head_and_relation_representation(*x: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for BPE head and relation entities.

Parameters
x (*B x 2 x T*) –

Returns
The representation for BPE head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]

Get the entity and relation embeddings.

Returns
The entity and relation embeddings.

Return type
Tuple[np.ndarray, np.ndarray]

class dicee.models.**DistMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

DistMult model for learning and inference in knowledge bases. It represents both entities and relations using embeddings and uses a simple bilinear form to compute scores for triples.

This implementation of the DistMult model is based on the paper: 'Embedding Entities and Relations for Learning and Inference in Knowledge Bases' (https://arxiv.org/abs/1412.6575).

name

The name identifier for the DistMult model.

Type
str

k_vs_all_score(*emb_h: torch.FloatTensor*, *emb_r: torch.FloatTensor*, *emb_E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using embeddings for a batch of head entities and relations.

forward_k_vs_all(*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for all entities given a batch of head entities and relations.

forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Computes scores for a sampled subset of entities given a batch of head entities and relations.

score(*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Computes the score of triples using DistMult's scoring function.

k_vs_all_score(*emb_h: torch.FloatTensor*, *emb_r: torch.FloatTensor*, *emb_E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using embeddings for a batch of head entities and relations.

This method multiplies the head entity and relation embeddings, applies a dropout and a normalization, and then computes the dot product with all tail entity embeddings.

Parameters

- **emb_h** (*torch.FloatTensor*) – Embeddings of head entities.
- **emb_r** (*torch.FloatTensor*) – Embeddings of relations.
- **emb_E** (*torch.FloatTensor*) – Embeddings of all entities.

Returns
Scores for all possible triples formed with the given head entities and relations against all entities.

Return type
torch.FloatTensor

forward_k_vs_all (*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for all entities given a batch of head entities and relations.

This method is used for K-vs-All scoring, where the model predicts the likelihood of each entity being the tail entity in a triple with each head entity and relation pair in the batch.

Parameters
x (*torch.LongTensor*) – Tensor containing indices for head entities and relations.

Returns
Scores for all entities for each head entity and relation pair in the batch.

Return type
torch.FloatTensor

forward_k_vs_sample (*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Computes scores for a sampled subset of entities given a batch of head entities and relations.

This method is particularly useful when the full set of entities is too large to score with every batch and only a subset of entities is required.

Parameters

- **x** (*torch.LongTensor*) – Tensor containing indices for head entities and relations.
- **target_entity_idx** (*torch.LongTensor*) – Indices of the target entities against which the scores are to be computed.

Returns
Scores for each head entity and relation pair against the sampled subset of entities.

Return type
torch.FloatTensor

score (*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Computes the score of triples using DistMult's scoring function.

The scoring function multiplies head entity and relation embeddings, applies dropout and normalization, and computes the dot product with the tail entity embeddings.

Parameters

- **h** (*torch.FloatTensor*) – Embedding of the head entity.
- **r** (*torch.FloatTensor*) – Embedding of the relation.
- **t** (*torch.FloatTensor*) – Embedding of the tail entity.

Returns
The score of the triple.

Return type
torch.FloatTensor

class dicee.models.**TransE**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

TransE model for learning embeddings in multi-relational data. It is based on the idea of translating embeddings for head entities by the relation vector to approach the tail entity embeddings in the embedding space.

This implementation of TransE is based on the paper: 'Translating Embeddings for Modeling Multi-relational Data' (https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf).

name

The name identifier for the TransE model.

Type
str

_norm

The norm used for computing pairwise distances in the embedding space.

Type
int

margin

The margin value used in the scoring function.

Type
int

score(*head_ent_emb: torch.Tensor*, *rel_ent_emb: torch.Tensor*, *tail_ent_emb: torch.Tensor*)
→ torch.Tensor

Computes the score of triples using the TransE scoring function.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for all entities given a head entity and a relation.

score(*head_ent_emb: torch.Tensor*, *rel_ent_emb: torch.Tensor*, *tail_ent_emb: torch.Tensor*)
→ torch.Tensor

Computes the score of triples using the TransE scoring function.

The scoring function computes the L2 distance between the translated head entity and the tail entity embeddings and subtracts this distance from the margin.

Parameters

- **head_ent_emb** (*torch.Tensor*) – Embedding of the head entity.
- **rel_ent_emb** (*torch.Tensor*) – Embedding of the relation.
- **tail_ent_emb** (*torch.Tensor*) – Embedding of the tail entity.

Returns
The score of the triple.

Return type
torch.Tensor

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for all entities given a head entity and a relation.

This method is used for K-vs-All scoring, where the model predicts the likelihood of each entity being the tail entity in a triple with each head entity and relation.

Parameters
x (*torch.Tensor*) – Tensor containing indices for head entities and relations.

Returns
Scores for all entities for each head entity and relation pair.

Return type
torch.FloatTensor

class dicee.models.**Shallom**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

Shallom is a shallow neural model designed for relation prediction in knowledge graphs. The model combines entity embeddings and passes them through a neural network to predict the likelihood of different relations. It's based on the paper: 'A Shallow Neural Model for Relation Prediction' (https://arxiv.org/abs/2101.09090).

name

The name identifier for the Shallom model.

Type
str

shallom

A sequential neural network model used for predicting relations.

Type
torch.nn.Sequential

get_embeddings() → Tuple[np.ndarray, None]

Retrieves the entity embeddings.

forward_k_vs_all(*x*) → torch.FloatTensor

Computes relation scores for all pairs of entities in the batch.

forward_triples(*x*) → torch.FloatTensor

Computes relation scores for a batch of triples.

get_embeddings() → Tuple[numpy.ndarray, None]

Retrieves the entity embeddings from the model.

Returns
A tuple containing the entity embeddings as a NumPy array and None for the relation embeddings.

Return type
Tuple[np.ndarray, None]

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes relation scores for all pairs of entities in the batch.

Each pair of entities is passed through the Shallom neural network to predict the likelihood of various relations between them.

Parameters
x (*torch.Tensor*) – A tensor of entity pairs.

Returns
A tensor of relation scores for each pair of entities in the batch.

Return type
torch.FloatTensor

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes relation scores for a batch of triples.

This method first computes relation scores for all possible relations for each pair of entities and then selects the scores corresponding to the actual relations in the triples.

Parameters
x (*torch.Tensor*) – A tensor containing a batch of triples.

Returns
A flattened tensor of relation scores for the given batch of triples.

Return type
torch.FloatTensor

class dicee.models.**Pyke**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

Pyke is a physical embedding model for knowledge graphs, emphasizing the geometric relationships in the embedding space. The model aims to represent entities and relations in a way that captures the underlying structure of the knowledge graph.

name

The name identifier for the Pyke model.

Type
str

dist_func

A pairwise distance function to compute distances in the embedding space.

Type
torch.nn.PairwiseDistance

margin

The margin value used in the scoring function.

Type
float

forward_triples(*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples based on the physical embedding approach.

forward_triples(*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples based on the physical embedding approach.

The method calculates the Euclidean distance between the head and relation embeddings, and between the relation and tail embeddings. The average of these distances is subtracted from the margin to compute the score for each triple.

Parameters
x (*torch.LongTensor*) – A tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples. Lower scores indicate more likely triples according to the geometric arrangement of embeddings.

Return type
torch.FloatTensor

class dicee.models.**BaseKGE**(*args: dict*)

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_byte_pair_encoded_k_vs_all(*x: torch.LongTensor*)

Parameters

x (*B x 2 x T*) –

forward_byte_pair_encoded_triple(*x: Tuple[torch.LongTensor, torch.LongTensor]*)

Perform the forward pass for byte pair encoded triples.

Parameters

x (*Tuple[torch.LongTensor, torch.LongTensor]*) – The input tuple containing byte pair encoded entities and relations.

Returns

The output tensor containing the scores for the byte pair encoded triples.

Return type

torch.Tensor

init_params_with_sanity_checking()

forward(*x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor]*, *y_idx: torch.LongTensor = None*)

Perform the forward pass of the model.

Parameters

- **x** (*Union[torch.LongTensor, Tuple[torch.LongTensor, torch.LongTensor]]*) – The input tensor or a tuple containing the input tensor and target entity indexes.
- **y_idx** (*torch.LongTensor, optional*) – The target entity indexes (default is None).

Returns
The output of the forward pass.

Return type
Any

forward_triples(*x: torch.LongTensor*) → torch.Tensor

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

forward_k_vs_all(**args*, ***kwargs*)

Forward pass for K vs. All.

Raises
ValueError – This function is not implemented in the current model.

forward_k_vs_sample(**args*, ***kwargs*)

Forward pass for K vs. Sample.

Raises
ValueError – This function is not implemented in the current model.

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for the head and relation entities.

Parameters
indexed_triple (*torch.LongTensor*) – The indexes of the head and relation entities.

Returns
The representation for the head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_sentence_representation(*x: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Get the representation for a sentence.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The representation for the input sentence.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

get_bpe_head_and_relation_representation(*x: torch.LongTensor*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for BPE head and relation entities.

Parameters
x (*B x 2 x T*) –

Returns
The representation for BPE head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]

Get the entity and relation embeddings.

Returns
The entity and relation embeddings.

Return type
Tuple[np.ndarray, np.ndarray]

class dicee.models.**ConEx**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

ConEx (Convolutional ComplEx) is a Knowledge Graph Embedding model that extends ComplEx embeddings with convolutional layers. It integrates convolutional neural networks into the embedding process to capture complex patterns in the data.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, kernel size, number of output channels, and dropout rates.

name

The name identifier for the ConEx model.

Type
str

conv2d

A 2D convolutional layer used for processing complex-valued embeddings.

Type
torch.nn.Conv2d

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

norm_fc1

Normalization layer applied after the fully connected layer.

Type
Normalizer

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(*C_1: Tuple[torch.Tensor, torch.Tensor]*, *C_2: Tuple[torch.Tensor, torch.Tensor]*) → Tuple[torch.Tensor, torch.Tensor]

Performs a residual convolution operation on two complex-valued embeddings.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional operations on embeddings.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*) → torch.Tensor

Computes scores against a sampled subset of entities using convolutional operations.

Notes

ConEx combines complex-valued embeddings with convolutional neural networks to capture intricate patterns and interactions in the knowledge graph, potentially leading to improved performance on tasks like link prediction.

residual_convolution(*C_1: Tuple[torch.Tensor, torch.Tensor]*, *C_2: Tuple[torch.Tensor, torch.Tensor]*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Computes the residual score of two complex-valued embeddings by applying convolutional operations. This method is a key component of the ConEx model, combining complex embeddings with convolutional neural networks.

Parameters

- **C_1** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple consisting of two PyTorch tensors representing the real and imaginary components of the first complex-valued embedding.
- **C_2** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple consisting of two PyTorch tensors representing the real and imaginary components of the second complex-valued embedding.

Returns
A tuple of two tensors, representing the real and imaginary parts of the convolutionally transformed embeddings.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

Notes

The method involves concatenating the real and imaginary components of the embeddings, applying a 2D convolution, followed by batch normalization, ReLU activation, dropout, and a fully connected layer. This process is intended to capture complex interactions between the embeddings in a convolutional manner.

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional operations on complex-valued embeddings. This method is used for evaluating the performance of the model by computing scores for each head entity and relation pair against all possible tail entities.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of head entities and relations. Expected tensor shape: (n, 2), where 'n' is the batch size and '2' represents head entity and relation pairs.

Returns

A tensor containing the scores for each head entity and relation pair against all possible tail entities. Tensor shape: (n, |E|), where '|E|' is the number of entities in the knowledge graph.

Return type

torch.FloatTensor

Notes

The method retrieves embeddings for head entities and relations, splits them into real and imaginary parts, and applies a convolution operation. It then computes the Hermitian product of the transformed embeddings with all tail entity embeddings to generate scores. This approach allows for capturing complex relational patterns in the knowledge graph.

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on complex-valued embeddings. This method is crucial for evaluating the performance of the model on individual triples in the knowledge graph.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of triples. Each triple consists of indices for a head entity, a relation, and a tail entity. Expected tensor shape: (n, 3), where 'n' is the number of triples.

Returns

A tensor containing the scores for each triple in the batch. Tensor shape: (n,), where 'n' is the number of triples.

Return type

torch.FloatTensor

Notes

The method retrieves embeddings for head entities, relations, and tail entities, and splits them into real and imaginary parts. It then applies a convolution operation on these embeddings and computes the Hermitian inner product, which involves a combination of real and imaginary parts of the embeddings. This process is designed to capture complex relational patterns and interactions within the knowledge graph, leveraging the power of convolutional neural networks.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*) → torch.Tensor

Computes scores against a sampled subset of entities using convolutional operations on complex-valued embeddings. This method is particularly useful for large knowledge graphs where computing scores against all entities is computationally expensive.

Parameters

- **x** (*torch.Tensor*) – A tensor representing a batch of head entities and relations. Expected tensor shape: (batch_size, 2), where 'batch_size' is the number of head entity and relation pairs.
- **target_entity_idx** (*torch.Tensor*) – A tensor of target entity indices for sampling. Tensor shape: (batch_size, num_selected_entities).

Returns

A tensor containing the scores for each head entity and relation pair against the sampled subset of tail entities. Tensor shape: (batch_size, num_selected_entities).

Return type

torch.Tensor

Notes

The method first retrieves and processes the embeddings for head entities and relations. It then applies a convolution operation and computes the Hermitian inner product with the embeddings of the sampled tail entities. This process enables capturing complex relational patterns in a computationally efficient manner.

class dicee.models.**AConEx**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

AConEx (Additive Convolutional ComplEx) extends the ConEx model by incorporating additive connections in the convolutional operations. This model integrates convolutional neural networks with complex-valued embeddings, emphasizing additive feature interactions for knowledge graph embeddings.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, kernel size, number of output channels, and dropout rates.

name

The name identifier for the AConEx model.

Type

str

conv2d

A 2D convolutional layer used for processing complex-valued embeddings.

Type

torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type

int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

norm_fc1

Normalization layer applied after the fully connected layer.

Type
Normalizer

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(C_1: Tuple[torch.Tensor, torch.Tensor],

C_2: Tuple[torch.Tensor, torch.Tensor]) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Performs a residual convolution operation on two complex-valued embeddings.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional operations on embeddings.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*)

Computes scores against a sampled subset of entities using convolutional operations.

Notes

AConEx aims to enhance the modeling capabilities of knowledge graph embeddings by adding more complex interaction patterns through convolutional layers, potentially improving performance on tasks like link prediction.

residual_convolution(*C_1: Tuple[torch.Tensor, torch.Tensor]*,
C_2: Tuple[torch.Tensor, torch.Tensor])
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Computes the residual convolution of two complex-valued embeddings. This method is a core part of the AConEx model, applying convolutional neural network techniques to complex-valued embeddings to capture intricate relationships in the data.

Parameters

- **C_1** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple of two PyTorch tensors representing the real and imaginary components of the first complex-valued embedding.
- **C_2** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple of two PyTorch tensors representing the real and imaginary components of the second complex-valued embedding.

Returns
A tuple of four tensors, each representing a component of the convolutionally transformed

embeddings. These components correspond to the modified real and imaginary parts of the input embeddings.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Notes

The method concatenates the real and imaginary components of the embeddings and applies a 2D convolution, followed by batch normalization, ReLU activation, dropout, and a fully connected layer. This convolutional process is designed to enhance the model's ability to capture complex patterns in knowledge graph embeddings.

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional and additive operations on complex-valued embeddings. This method evaluates the performance of the model by computing scores for each head entity and relation pair against all possible tail entities.

Parameters
x (*torch.Tensor*) – A tensor representing a batch of head entities and relations. Expected tensor shape: (batch_size, 2), where 'batch_size' is the number of head entity and relation pairs.

Returns
A tensor containing the scores for each head entity and relation pair against all possible tail entities. Tensor shape: (batch_size, |E|), where '|E|' is the number of entities in the knowledge graph.

Return type
torch.FloatTensor

Notes

The method first retrieves embeddings for head entities and relations, splits them into real and imaginary parts, and applies a convolutional operation. It then computes the Hermitian inner product with all tail entity embeddings, using an additive approach that combines the convolutional results with the original embeddings. This technique aims to capture complex relational patterns in the knowledge graph.

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations and additive connections on complex-valued embeddings. This method is key for evaluating the model's performance on individual triples within the knowledge graph.

Parameters
x (*torch.Tensor*) – A tensor representing a batch of triples. Each triple consists of indices for a head entity, a relation, and a tail entity. Expected tensor shape: (n, 3), where 'n' is the number of triples.

Returns
A tensor containing the scores for each triple in the batch. Tensor shape: (n,), where 'n' is the number of triples.

Return type
torch.FloatTensor

Notes

The method retrieves embeddings for head entities, relations, and tail entities, and splits them into real and imaginary parts. It then applies a convolution operation on these embeddings and computes the Hermitian inner product, enhanced with an additive connection. This approach allows the model to capture complex relational patterns within the knowledge graph, potentially improving prediction accuracy and interpretability.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of samples (entity pairs) given a batch of queries. This method is used to predict the scores for different tail entities for a set of query triples.

Parameters

- **x** (*torch.Tensor*) – A tensor representing a batch of query triples. Each triple consists of indices for a head entity, a relation, and a dummy tail entity (used for scoring). Expected tensor shape: (n, 3), where 'n' is the number of query triples.
- **target_entity_idx** (*torch.Tensor*) – A tensor containing the indices of the target tail entities for which scores are to be predicted. Expected tensor shape: (n, m), where 'n' is the number of queries and 'm' is the number of target entities.

Returns

A tensor containing the scores for each query-triple and target-entity pair. Tensor shape: (n, m), where 'n' is the number of queries and 'm' is the number of target entities.

Return type

torch.FloatTensor

Notes

This method retrieves embeddings for the head entities and relations in the query triples, splits them into real and imaginary parts, and applies convolutional operations with additive connections to capture complex patterns. It also retrieves embeddings for the target tail entities and computes Hermitian inner products to obtain scores, allowing the model to rank the tail entities based on their relevance to the queries.

class dicee.models.**ComplEx**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

ComplEx (Complex Embeddings for Knowledge Graphs) is a model that extends the base knowledge graph embedding approach by using complex-valued embeddings. It emphasizes the interaction of real and imaginary components of embeddings to capture the asymmetric relationships often found in knowledge graphs.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, learning rate, and regularization methods.

name

The name identifier for the ComplEx model.

Type

str

score(head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor,

tail_ent_emb: torch.FloatTensor) -> torch.FloatTensor

Computes the score of a triple using the ComplEx scoring function.

k_vs_all_score(emb_h: torch.FloatTensor, emb_r: torch.FloatTensor,

emb_E: torch.FloatTensor) -> torch.FloatTensor

Computes scores in a K-vs-All setting using complex-valued embeddings.

forward_k_vs_all (*x: torch.LongTensor*) → torch.FloatTensor

Performs a forward pass for K-vs-All scoring, returning scores for all entities.

Notes

ComplEx is particularly suited for modeling asymmetric relations and has been shown to perform well on various knowledge graph benchmarks. The use of complex numbers allows the model to encode additional information compared to real-valued models.

static score (*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Compute the scoring function for a given triple using complex-valued embeddings.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – The complex embedding of the head entity.
- **rel_ent_emb** (*torch.FloatTensor*) – The complex embedding of the relation.
- **tail_ent_emb** (*torch.FloatTensor*) – The complex embedding of the tail entity.

Returns
The score of the triple calculated using the Hermitian dot product of complex embeddings.

Return type
torch.FloatTensor

Notes

The scoring function exploits the complex vector space to model the interactions between entities and relations. It involves element-wise multiplication and summation of real and imaginary parts.

static k_vs_all_score (*emb_h: torch.FloatTensor*, *emb_r: torch.FloatTensor*, *emb_E: torch.FloatTensor*) → torch.FloatTensor

Compute scores for a head entity and relation against all entities in a K-vs-All scenario.

Parameters

- **emb_h** (*torch.FloatTensor*) – The complex embedding of the head entity.
- **emb_r** (*torch.FloatTensor*) – The complex embedding of the relation.
- **emb_E** (*torch.FloatTensor*) – The complex embeddings of all possible tail entities.

Returns
Scores for all possible triples formed with the given head entity and relation.

Return type
torch.FloatTensor

Notes

This method is useful for tasks like link prediction where the model predicts the likelihood of a relation between a given entity pair.

forward_k_vs_all(*x: torch.LongTensor*) → torch.FloatTensor

Perform a forward pass for K-vs-all scoring using complex-valued embeddings.

Parameters
x (*torch.LongTensor*) – Tensor containing indices for head entities and relations.

Returns
Scores for all triples formed with the given head entities and relations against all entities.

Return type
torch.FloatTensor

Notes

This method is typically used in training and evaluation of the model in a link prediction setting, where the goal is to rank all possible tail entities for a given head entity and relation.

dicee.models.**quaternion_mul**(*, *Q_1: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]*, *Q_2: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]*)
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Perform quaternion multiplication.

This function multiplies two quaternions, Q_1 and Q_2, and returns the result as a quaternion. Quaternion multiplication is a non-commutative operation used in various applications, including 3D rotation and orientation tasks.

Parameters

- **Q_1** (*Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]*) – The first quaternion, represented as a tuple of four components (a_h, b_h, c_h, d_h).
- **Q_2** (*Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]*) – The second quaternion, represented as a tuple of four components (a_r, b_r, c_r, d_r).

Returns
The resulting quaternion from the multiplication, represented as a tuple of four components (r_val, i_val, j_val, k_val).

Return type
Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Notes

The quaternion multiplication is defined as: r_val = a_h * a_r - b_h * b_r - c_h * c_r - d_h * d_r i_val = a_h * b_r + b_h * a_r + c_h * d_r - d_h * c_r j_val = a_h * c_r - b_h * d_r + c_h * a_r + d_h * b_r k_val = a_h * d_r + b_h * c_r - c_h * b_r + d_h * a_r

class dicee.models.**BaseKGE**(*args: dict*)

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables
training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_byte_pair_encoded_k_vs_all (*x: torch.LongTensor*)

Parameters
x (*B x 2 x T*) –

forward_byte_pair_encoded_triple (*x: Tuple[torch.LongTensor, torch.LongTensor]*)

Perform the forward pass for byte pair encoded triples.

Parameters
x (*Tuple[torch.LongTensor, torch.LongTensor]*) – The input tuple containing byte pair encoded entities and relations.

Returns
The output tensor containing the scores for the byte pair encoded triples.

Return type
torch.Tensor

init_params_with_sanity_checking()

forward (*x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor]*, *y_idx: torch.LongTensor = None*)

Perform the forward pass of the model.

Parameters

- **x** (*Union[torch.LongTensor, Tuple[torch.LongTensor, torch.LongTensor]]*) – The input tensor or a tuple containing the input tensor and target entity indexes.
- **y_idx** (*torch.LongTensor, optional*) – The target entity indexes (default is None).

Returns
The output of the forward pass.

Return type
Any

forward_triples(*x: torch.LongTensor*) → torch.Tensor

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

forward_k_vs_all(**args*, ***kwargs*)

Forward pass for K vs. All.

Raises
ValueError – This function is not implemented in the current model.

forward_k_vs_sample(**args*, ***kwargs*)

Forward pass for K vs. Sample.

Raises
ValueError – This function is not implemented in the current model.

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for the head and relation entities.

Parameters
indexed_triple (*torch.LongTensor*) – The indexes of the head and relation entities.

Returns
The representation for the head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_sentence_representation(*x: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Get the representation for a sentence.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The representation for the input sentence.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

get_bpe_head_and_relation_representation(*x: torch.LongTensor*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for BPE head and relation entities.

Parameters
x (*B x 2 x T*) –

Returns
The representation for BPE head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]

Get the entity and relation embeddings.

Returns
The entity and relation embeddings.

Return type
Tuple[np.ndarray, np.ndarray]

class dicee.models.**IdentityClass**(*args: Dict | None = None*)

Bases: torch.nn.Module

A class that represents an identity function.

Parameters
args (*dict, optional*) – A dictionary containing arguments (default is None).

__call__(*x*)

static forward(*x: torch.Tensor*) → torch.Tensor

The forward pass of the identity function.

Parameters
x (*torch.Tensor*) – The input tensor.

Returns
The output tensor, which is the same as the input.

Return type
torch.Tensor

dicee.models.**quaternion_mul_with_unit_norm**(*, *Q_1: Tuple[float, float, float, float]*, *Q_2: Tuple[float, float, float, float]*) → Tuple[float, float, float, float]

Performs the multiplication of two quaternions with unit norm.

Parameters

- **Q_1** (*Tuple[float, float, float, float]*) – The first quaternion represented as a tuple of four real numbers (a_h, b_h, c_h, d_h).
- **Q_2** (*Tuple[float, float, float, float]*) – The second quaternion represented as a tuple of four real numbers (a_r, b_r, c_r, d_r).

Returns
The result of the quaternion multiplication, represented as a tuple of four real numbers (r_val, i_val, j_val, k_val).

Return type
Tuple[float, float, float, float]

Notes

The function assumes that the input quaternions have unit norm. It first normalizes the second quaternion to eliminate the scaling effect, and then performs the Hamilton product of the two quaternions.

class dicee.models.**QMult**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

QMult extends the base knowledge graph embedding model by integrating quaternion algebra. This model leverages the properties of quaternions to represent and process the embeddings of entities and relations in a knowledge graph, aiming to capture complex interactions and patterns.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions and learning rate.

name

The name identifier for the QMult model.

Type

str

quaternion_normalizer(*x: torch.FloatTensor*) → torch.FloatTensor

Normalizes the length of relation vectors.

score(*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes the score of a triple using quaternion multiplication.

k_vs_all_score(*bpe_head_ent_emb: torch.FloatTensor*, *bpe_rel_ent_emb: torch.FloatTensor*, *E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using quaternion embeddings.

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Performs a forward pass for K-vs-All scoring, returning scores for all entities.

forward_k_vs_sample(*x: torch.FloatTensor*, *target_entity_idx: int*) → torch.FloatTensor

Performs a forward pass for K-vs-Sample scoring, returning scores for the specified entities.

quaternion_multiplication_followed_by_inner_product(*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Performs quaternion multiplication followed by inner product, returning triple scores.

quaternion_multiplication_followed_by_inner_product(*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Performs quaternion multiplication followed by inner product.

Parameters

- **h** (*torch.FloatTensor*) – The head representations. Shape: (**batch_dims*, dim)
- **r** (*torch.FloatTensor*) – The relation representations. Shape: (**batch_dims*, dim)
- **t** (*torch.FloatTensor*) – The tail representations. Shape: (**batch_dims*, dim)

Returns

Triple scores.

Return type

torch.FloatTensor

static quaternion_normalizer(*x: torch.FloatTensor*) → torch.FloatTensor

TODO: Add mathematical format for sphinx. Normalize the length of relation vectors, if the forward constraint has not been applied yet.

The absolute value of a quaternion is calculated as follows: .. math:

```
|a + bi + cj + dk| = \sqrt{a^2 + b^2 + c^2 + d^2}
```

The L2 norm of a quaternion vector is computed as: .. math:

```
\|x\|^2 = \sum_{i=1}^d |x_i|^2
        = \sum_{i=1}^d (x_i.re^2 + x_i.im_1^2 + x_i.im_2^2 + x_i.im_3^2)
```

Parameters

x (*torch.FloatTensor*) – The vector containing quaternion values.

Returns

The normalized vector.

Return type

torch.FloatTensor

Notes

This function normalizes the length of relation vectors represented as quaternions. It ensures that the absolute value of each quaternion in the vector is equal to 1, preserving the unit length.

score(*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Compute scores for a batch of triples using octonion-based embeddings.

This method computes scores for a batch of triples using octonion-based embeddings of head entities, relation embeddings, and tail entities. It supports both explicit and non-explicit scoring methods.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – Tensor containing the octonion-based embeddings of head entities.
- **rel_ent_emb** (*torch.FloatTensor*) – Tensor containing the octonion-based embeddings of relations.
- **tail_ent_emb** (*torch.FloatTensor*) – Tensor containing the octonion-based embeddings of tail entities.

Returns

Scores for the given batch of triples.

Return type

torch.FloatTensor

Notes

If no normalization is set, this method applies quaternion normalization to relation embeddings.

If the scoring method is explicit, it computes the scores using quaternion multiplication followed by an inner product of the real and imaginary parts of the resulting quaternions.

If the scoring method is non-explicit, it directly computes the inner product of the real and imaginary parts of the octonion-based embeddings.

k_vs_all_score(*bpe_head_ent_emb: torch.FloatTensor*, *bpe_rel_ent_emb: torch.FloatTensor*, *E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using quaternion embeddings for a batch of head entities and relations.

This method involves splitting the head entity and relation embeddings into quaternion components, optionally normalizing the relation embeddings, performing quaternion multiplication, and then calculating the score by performing an inner product with all tail entity embeddings.

Parameters

- **bpe_head_ent_emb** (*torch.FloatTensor*) – Batched embeddings of head entities, each represented as a quaternion.
- **bpe_rel_ent_emb** (*torch.FloatTensor*) – Batched embeddings of relations, each represented as a quaternion.
- **E** (*torch.FloatTensor*) – Embeddings of all possible tail entities.

Returns

Scores for all possible triples formed with the given head entities and relations against all entities. The shape of the output is (size of batch, number of entities).

Return type

torch.FloatTensor

Notes

The method is particularly useful in scenarios like link prediction, where the goal is to rank all possible tail entities for a given head entity and relation. Quaternion algebra is used to enhance the interaction modeling between entities and relations.

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

This method retrieves embeddings for the head entities and relations from the input tensor *x*, applies necessary dropout and normalization, and then uses the *k_vs_all_score* method to compute the scores against all possible tail entities in the knowledge graph.

Parameters

x (*torch.FloatTensor*) – A tensor containing indices for head entities and relations. The tensor is expected to have a specific format suitable for the model's embedding retrieval process.

Returns

A tensor of scores, where each row corresponds to the scores of all tail entities for a single head entity and relation pair. The shape of the tensor is (size of the batch, number of entities).

Return type

torch.FloatTensor

Notes

This method is typically used in evaluating the model's performance in link prediction tasks, where it's important to rank the likelihood of every possible tail entity for a given head entity and relation.

forward_k_vs_sample(*x: torch.FloatTensor*, *target_entity_idx: int*) → torch.FloatTensor

Computes scores for a batch of triples against a sampled subset of entities in a K-vs-Sample setting.

Given a batch of head entities and relations (h,r), this method computes the scores for all possible triples formed with these head entities and relations against a subset of entities, i.e., [score(h,r,x)|x in Entities] => [0.0,0.1,…,0.8], shape=> (1, |Entities|). TODO: Add mathematical format for sphinx. The subset of entities is specified by the *target_entity_idx*, which is an integer index representing a specific entity. Given a batch of head entities and relations => shape (size of batch,| Entities|).

Parameters

- **x** (*torch.FloatTensor*) – A tensor containing indices for head entities and relations. The tensor is expected to have a specific format suitable for the model's embedding retrieval process.
- **target_entity_idx** (*int*) – Index of the target entity against which the scores are to be computed.

Returns

A tensor of scores where each element corresponds to the score of the target entity for a single head entity and relation pair. The shape of the tensor is (size of the batch, 1).

Return type

torch.FloatTensor

Notes

This method is particularly useful in scenarios like link prediction, where it's necessary to evaluate the likelihood of a specific relationship between a given head entity and a particular target entity.

class dicee.models.**ConvQ**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings (ConvQ) is a model that extends the base knowledge graph embedding approach by using quaternion algebra and convolutional neural networks. This model aims to capture complex interactions in knowledge graphs by applying convolutions to quaternion-based entity and relation embeddings.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, number of output channels, kernel size, and dropout rates.

name

The name identifier for the ConvQ model.

Type

str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type

torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

conv2d

A 2D convolutional layer used for processing quaternion embeddings.

Type
torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

bn_conv1

First batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

bn_conv2

Second normalization layer applied after the fully connected layer.

Type
Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(*Q_1*, *Q_2*)

Performs a residual convolution operation on two sets of quaternion embeddings.

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on quaternion embeddings.

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

Notes

ConvQ leverages the properties of quaternions, a number system that extends complex numbers, to represent and process the embeddings of entities and relations. The convolutional layers aim to capture spatial relationships and complex patterns in the embeddings.

residual_convolution(
Q_1: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor],
Q_2: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor])
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Performs a residual convolution operation on two sets of quaternion embeddings.

The method combines two quaternion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **Q_1** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The first set of quaternion embeddings.
- **Q_2** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The second set of quaternion embeddings.

Returns
The resulting quaternion embeddings after the convolutional operation.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on quaternion embeddings.

The method processes head, relation, and tail embeddings using quaternion algebra and convolutional layers and computes the scores of the triples.

Parameters
indexed_triple (*torch.FloatTensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.FloatTensor

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

This method retrieves embeddings for the head entities and relations from the input tensor *x*, applies necessary dropout and normalization, and then computes scores against all entities in the knowledge graph.

Parameters
x (*torch.FloatTensor*) – A tensor containing indices for head entities and relations.

Returns
Scores for all entities for the given batch of head entities and relations.

Return type
torch.FloatTensor

class dicee.models.**AConvQ**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Quaternion Knowledge Graph Embeddings (AConvQ) model integrates quaternion algebra with convolutional neural networks for knowledge graph embeddings. This model is designed to capture complex interactions in knowledge graphs by applying additive convolutions to quaternion-based entity and relation embeddings.

name

The name identifier for the AConvQ model.

Type
str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

conv2d

A 2D convolutional layer used for processing quaternion embeddings.

Type
torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

bn_conv1

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

bn_conv2

Normalization layer applied after the fully connected layer.

Type
Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(*Q_1*, *Q_2*)

Performs an additive residual convolution operation on two sets of quaternion embeddings.

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using additive convolutional operations on quaternion embeddings.

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

residual_convolution(
Q_1: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor],
Q_2: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor])
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Performs a residual convolution operation on two sets of quaternion embeddings.

The method combines two quaternion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **Q_1** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The first set of quaternion embeddings.
- **Q_2** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The second set of quaternion embeddings.

Returns
The resulting quaternion embeddings after the convolutional operation.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on quaternion embeddings.

The method processes head, relation, and tail embeddings using quaternion algebra and convolutional layers and computes the scores of the triples.

Parameters
indexed_triple (*torch.FloatTensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.FloatTensor

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

This method retrieves embeddings for the head entities and relations from the input tensor *x*, applies necessary dropout and normalization, and then computes scores against all entities in the knowledge graph.

Parameters
x (*torch.FloatTensor*) – A tensor containing indices for head entities and relations.

Returns
Scores for all entities for the given batch of head entities and relations.

Return type
torch.FloatTensor

class dicee.models.**BaseKGE**(*args: dict*)

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables
training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_byte_pair_encoded_k_vs_all(*x: torch.LongTensor*)

Parameters
x (*B x 2 x T*) –

forward_byte_pair_encoded_triple(*x: Tuple[torch.LongTensor, torch.LongTensor]*)

Perform the forward pass for byte pair encoded triples.

Parameters
x (*Tuple[torch.LongTensor, torch.LongTensor]*) – The input tuple containing byte pair encoded entities and relations.

Returns
The output tensor containing the scores for the byte pair encoded triples.

Return type
torch.Tensor

init_params_with_sanity_checking()

forward(*x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor]*, *y_idx: torch.LongTensor = None*)

Perform the forward pass of the model.

Parameters

- **x** (*Union[torch.LongTensor, Tuple[torch.LongTensor, torch.LongTensor]]*) – The input tensor or a tuple containing the input tensor and target entity indexes.
- **y_idx** (*torch.LongTensor, optional*) – The target entity indexes (default is None).

Returns
The output of the forward pass.

Return type
Any

forward_triples(*x: torch.LongTensor*) → torch.Tensor

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

forward_k_vs_all(**args*, ***kwargs*)

Forward pass for K vs. All.

Raises
ValueError – This function is not implemented in the current model.

forward_k_vs_sample(**args*, ***kwargs*)

Forward pass for K vs. Sample.

Raises
ValueError – This function is not implemented in the current model.

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for the head and relation entities.

Parameters
indexed_triple (*torch.LongTensor*) – The indexes of the head and relation entities.

Returns
The representation for the head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_sentence_representation(*x: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Get the representation for a sentence.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The representation for the input sentence.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

get_bpe_head_and_relation_representation(*x: torch.LongTensor*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for BPE head and relation entities.

Parameters
x (*B x 2 x T*) –

Returns
The representation for BPE head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]

Get the entity and relation embeddings.

Returns
The entity and relation embeddings.

Return type
Tuple[np.ndarray, np.ndarray]

class dicee.models.**IdentityClass**(*args: Dict | None = None*)

Bases: torch.nn.Module

A class that represents an identity function.

Parameters
args (*dict, optional*) – A dictionary containing arguments (default is None).

__call__(*x*)

static forward(*x: torch.Tensor*) → torch.Tensor

The forward pass of the identity function.

Parameters
x (*torch.Tensor*) – The input tensor.

Returns
The output tensor, which is the same as the input.

Return type
torch.Tensor

dicee.models.**octonion_mul**(*, *O_1: Tuple[float, float, float, float, float, float, float, float]*, *O_2: Tuple[float, float, float, float, float, float, float, float]*) → Tuple[float, float, float, float, float, float, float, float]

Performs the multiplication of two octonions.

Octonions are an extension of quaternions and are represented here as 8-tuples of floats. This function computes the product of two octonions using their components.

Parameters

- **O_1** (*Tuple[float, float, float, float, float, float, float, float]*) – The first octonion, represented as an 8-tuple of float components.

- **O_2** (*Tuple[float, float, float, float, float, float, float, float]*) – The second octonion, represented as an 8-tuple of float components.

Returns
The product of the two octonions, represented as an 8-tuple of float components.

Return type
Tuple[float, float, float, float, float, float, float, float]

dicee.models.**octonion_mul_norm**(*, *O_1: Tuple[float, float, float, float, float, float, float, float]*, *O_2: Tuple[float, float, float, float, float, float, float, float]*) → Tuple[float, float, float, float, float, float, float, float]

Performs the normalized multiplication of two octonions.

This function first normalizes the second octonion to unit length to eliminate the scaling effect and then computes the product of two octonions using their components.

Parameters

- **O_1** (*Tuple[float, float, float, float, float, float, float, float]*) – The first octonion, represented as an 8-tuple of float components.
- **O_2** (*Tuple[float, float, float, float, float, float, float, float]*) – The second octonion, represented as an 8-tuple of float components.

Returns
The product of the two octonions, represented as an 8-tuple of float components.

Return type
Tuple[float, float, float, float, float, float, float, float]

Notes

Normalization may cause NaNs due to floating-point precision issues, especially if the second octonion's magnitude is very small.

class dicee.models.**OMult**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

OMult extends the base knowledge graph embedding model by integrating octonion algebra. This model leverages the properties of octonions to represent and process the embeddings of entities and relations in a knowledge graph, aiming to capture complex interactions and patterns.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions and learning rate.

name
The name identifier for the OMult model.

Type
str

octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, ..., *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, ...]

Normalizes octonion components to unit length.

score(*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes the score of a triple using octonion multiplication.

k_vs_all_score(*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*) → torch.FloatTensor

Computes scores in a K-vs-All setting using octonion embeddings.

forward_k_vs_all(*x*) → torch.FloatTensor

Performs a forward pass for K-vs-All scoring, returning scores for all entities.

static octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, *emb_rel_e2: torch.Tensor*, *emb_rel_e3: torch.Tensor*, *emb_rel_e4: torch.Tensor*, *emb_rel_e5: torch.Tensor*, *emb_rel_e6: torch.Tensor*, *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Normalizes the components of an octonion.

Each component of the octonion is divided by the square root of the sum of the squares of all components, normalizing it to unit length.

Parameters

- **emb_rel_e0** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e1** (*torch.Tensor*) – The eight components of an octonion.
- **...** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e7** (*torch.Tensor*) – The eight components of an octonion.

Returns
The normalized components of the octonion.

Return type
Tuple[torch.Tensor, …]

score(*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes the score of a triple using octonion multiplication.

The method involves splitting the embeddings into real and imaginary parts, normalizing the relation embeddings, performing octonion multiplication, and then calculating the score based on the inner product.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – Embedding of the head entity.
- **rel_ent_emb** (*torch.FloatTensor*) – Embedding of the relation.
- **tail_ent_emb** (*torch.FloatTensor*) – Embedding of the tail entity.

Returns
The score of the triple.

Return type
torch.FloatTensor

k_vs_all_score(*bpe_head_ent_emb: torch.FloatTensor*, *bpe_rel_ent_emb: torch.FloatTensor*, *E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using octonion embeddings for a batch of head entities and relations.

This method splits the head entity and relation embeddings into their octonion components, normalizes the relation embeddings if necessary, and then applies octonion multiplication. It computes the score by performing an inner product with all tail entity embeddings.

Parameters

- **bpe_head_ent_emb** (*torch.FloatTensor*) – Batched embeddings of head entities, each represented as an octonion.

- **bpe_rel_ent_emb** (*torch.FloatTensor*) – Batched embeddings of relations, each represented as an octonion.
- **E** (*torch.FloatTensor*) – Embeddings of all possible tail entities.

Returns
Scores for all possible triples formed with the given head entities and relations against all entities. The shape of the output is (size of batch, number of entities).

Return type
torch.FloatTensor

Notes

The method is particularly useful in scenarios like link prediction, where the goal is to rank all possible tail entities for a given head entity and relation.

forward_k_vs_all(*x*)

Performs a forward pass for K-vs-All scoring.

TODO: Add mathematical format for sphinx.

Given a head entity and a relation (h,r), this method computes scores for all possible triples, i.e., [score(h,r,x)|x in Entities] => [0.0,0.1,…,0.8], shape=> (1, **|Entities|**), returning a score for each entity in the knowledge graph.

Parameters
x (*Tensor*) – Tensor containing indices for head entities and relations.

Returns
Scores for all triples formed with the given head entities and relations against all entities.

Return type
torch.FloatTensor

class dicee.models.**ConvO**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

ConvO extends the base knowledge graph embedding model by integrating convolutional operations with octonion algebra. This model applies convolutional neural networks to octonion-based embeddings, capturing complex interactions in knowledge graphs.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, number of output channels, kernel size, and dropout rates.

name

The name identifier for the ConvO model.

Type
str

conv2d

A 2D convolutional layer used for processing octonion-based embeddings.

Type
torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

norm_fc1

Normalization layer applied after the fully connected layer.

Type
Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

octonion_normalizer(*emb_rel_e0*, *emb_rel_e1*, ..., *emb_rel_e7*)

Normalizes octonion components to unit length.

residual_convolution(*O_1*, *O_2*)

Performs a residual convolution operation on two octonion embeddings.

forward_triples(*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_all(*x: torch.Tensor*)

Computes scores against a sampled subset of entities using convolutional operations.

Notes

ConvO aims to enhance the modeling capabilities of knowledge graph embeddings by adding more complex interaction patterns through convolutional layers, potentially improving performance on tasks like link prediction.

static octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, *emb_rel_e2: torch.Tensor*, *emb_rel_e3: torch.Tensor*, *emb_rel_e4: torch.Tensor*, *emb_rel_e5: torch.Tensor*, *emb_rel_e6: torch.Tensor*, *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Normalizes the components of an octonion to unit length.

Each component of the octonion is divided by the square root of the sum of the squares of all components.

Parameters

- **emb_rel_e0** (*torch.Tensor*) – The eight components of an octonion.

- **emb_rel_e1** (*torch.Tensor*) – The eight components of an octonion.
- **...** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e7** (*torch.Tensor*) – The eight components of an octonion.

Returns
The normalized components of the octonion.

Return type
Tuple[torch.Tensor, …]

residual_convolution (
O_1: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
O_2: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Performs a residual convolution operation on two sets of octonion embeddings.

The method combines two octonion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters
- **O_1** (*Tuple[torch.Tensor, ...]*) – The first set of octonion embeddings.
- **O_2** (*Tuple[torch.Tensor, ...]*) – The second set of octonion embeddings.

Returns
The resulting octonion embeddings after the convolutional operation.

Return type
Tuple[torch.Tensor, …]

forward_triples (*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

The method processes head, relation, and tail embeddings using convolutional layers and computes the scores of the triples.

Parameters
x (*torch.Tensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.Tensor

forward_k_vs_all (*x: torch.Tensor*) → torch.Tensor

Given a batch of head entities and relations (h,r), this method computes scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,…,0.8], shape=> (1, |Entities|) Given a batch of head entities and relations => shape (size of batch,| Entities|)

Parameters
x (*torch.Tensor*) – A tensor representing a batch of input triples in the form of (head entities, relations).

Returns
Scores for the input triples against all possible tail entities.

Return type
torch.Tensor

Notes

- The input x is a tensor of shape (batch_size, 2), where each row represents a pair of head entities and relations.
- **The method follows the following steps:**
  (1) Retrieve embeddings & Apply Dropout & Normalization.
  (2) Split the embeddings into real and imaginary parts.
  (3) Apply convolution operation on the real and imaginary parts.
  (4) Perform quaternion multiplication.
  (5) Compute scores for all entities.

The method returns a tensor of shape (batch_size, num_entities) where each row contains scores for each entity in the knowledge graph.

class dicee.models.**AConvO**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Octonion(AConvO) extends the base knowledge graph embedding model by integrating additive convolutional operations with octonion algebra. This model applies convolutional neural networks to octonion-based embeddings, capturing complex interactions in knowledge graphs.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, number of output channels, kernel size, and dropout rates.

name

The name identifier for the AConvO model.

Type
str

conv2d

A 2D convolutional layer used for processing octonion-based embeddings.

Type
torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

norm_fc1

Normalization layer applied after the fully connected layer.

Type
Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, ..., *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, ...]

Normalizes octonion components to unit length.

residual_convolution(*self*, *O_1: Tuple[torch.Tensor, ...]*, *O_2: Tuple[torch.Tensor, ...]*) → Tuple[torch.Tensor, ...]

Performs a residual convolution operation on two octonion embeddings.

forward_triples(*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_all(*x: torch.Tensor*)

Computes scores against a sampled subset of entities using convolutional operations.

Notes

AConvO aims to enhance the modeling capabilities of knowledge graph embeddings by adding more complex interaction patterns through convolutional layers, potentially improving performance on tasks like link prediction.

static octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, *emb_rel_e2: torch.Tensor*, *emb_rel_e3: torch.Tensor*, *emb_rel_e4: torch.Tensor*, *emb_rel_e5: torch.Tensor*, *emb_rel_e6: torch.Tensor*, *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Normalizes the components of an octonion to unit length.

Each component of the octonion is divided by the square root of the sum of the squares of all components.

Parameters

- **emb_rel_e0** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e1** (*torch.Tensor*) – The eight components of an octonion.
- **...** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e7** (*torch.Tensor*) – The eight components of an octonion.

Returns
The normalized components of the octonion.

Return type
Tuple[torch.Tensor, …]

residual_convolution(
O_1: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
O_2: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Performs a residual convolution operation on two sets of octonion embeddings.

The method combines two octonion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **O_1** (*Tuple[torch.Tensor, ...]*) – The first set of octonion embeddings.
- **O_2** (*Tuple[torch.Tensor, ...]*) – The second set of octonion embeddings.

Returns
The resulting octonion embeddings after the convolutional operation.

Return type
Tuple[torch.Tensor, …]

forward_triples(*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

The method processes head, relation, and tail embeddings using convolutional layers and computes the scores of the triples.

Parameters
x (*torch.Tensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.Tensor

forward_k_vs_all(*x: torch.Tensor*) → torch.Tensor

Compute scores for a head entity and a relation (h,r) against all entities in the knowledge graph.

Given a head entity and a relation (h, r), this method computes scores for (h, r, x) for all entities x in the knowledge graph.

Parameters
x (*torch.Tensor*) – A tensor containing indices for head entities and relations.

Returns
A tensor of scores representing the compatibility of (h, r, x) for all entities x in the knowledge graph.

Return type
torch.Tensor

Notes

This method supports batch processing, allowing the input tensor *x* to contain multiple head entities and relations.

The scores indicate how well each entity x in the knowledge graph fits the (h, r) pattern, with higher scores indicating better compatibility.

class dicee.models.**Keci**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

The Keci class is a knowledge graph embedding model that incorporates Clifford algebra for embeddings. It supports different dimensions of Clifford algebra by setting the parameters p and q. The class utilizes Clifford multiplication for embedding interactions and computes scores for knowledge graph triples.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model.

name

The name identifier for the Keci class.

Type

str

p

The parameter 'p' in Clifford algebra, representing the number of positive square terms.

Type

int

q

The parameter 'q' in Clifford algebra, representing the number of negative square terms.

Type

int

r

A derived attribute for dimension scaling based on 'p' and 'q'.

Type

int

p_coefficients

Embedding for scaling coefficients of 'p' terms, if 'p' > 0.

Type

torch.nn.Embedding (optional)

q_coefficients

Embedding for scaling coefficients of 'q' terms, if 'q' > 0.

Type

torch.nn.Embedding (optional)

compute_sigma_pp(*hp: torch.Tensor*, *rp: torch.Tensor*) → torch.Tensor

Computes the sigma_pp component in Clifford multiplication.

compute_sigma_qq(*hq: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_qq component in Clifford multiplication.

compute_sigma_pq(*hp: torch.Tensor*, *hq: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_pq component in Clifford multiplication.

apply_coefficients(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → tuple

Applies scaling coefficients to the base vectors in Clifford algebra.

clifford_multiplication(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → tuple

Performs Clifford multiplication of head and relation embeddings.

construct_cl_multivector (*x: torch.FloatTensor*, *r: int*, *p: int*, *q: int*) → tuple

Constructs a multivector in Clifford algebra Cl_{p,q}(mathbb{R}^d).

forward_k_vs_with_explicit (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities using explicit Clifford multiplication.

k_vs_all_score (*bpe_head_ent_emb: torch.Tensor*, *bpe_rel_ent_emb: torch.Tensor*, *E: torch.Tensor*) → torch.FloatTensor

Computes scores for all triples using Clifford multiplication in a K-vs-All setup.

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Wrapper function for K-vs-All scoring.

forward_k_vs_sample (*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Computes scores for a sampled subset of entities.

score (*h: torch.Tensor*, *r: torch.Tensor*, *t: torch.Tensor*) → torch.FloatTensor

Computes the score for a given triple using Clifford multiplication.

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples.

Notes

The class is designed to work with embeddings in the context of knowledge graph completion tasks, leveraging the properties of Clifford algebra for embedding interactions.

compute_sigma_pp (*hp: torch.Tensor*, *rp: torch.Tensor*) → torch.Tensor

Computes the sigma_pp component in Clifford multiplication, representing the interactions between the positive square terms in the Clifford algebra.

sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k, TODO: Add mathematical format for sphinx.

sigma_{pp} captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

for k in range(i + 1, p):
results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

Parameters

- **hp** (*torch.Tensor*) – The 'p' part of the head entity embedding in Clifford algebra.
- **rp** (*torch.Tensor*) – The 'p' part of the relation embedding in Clifford algebra.

Returns

sigma_pp – The sigma_pp component of the Clifford multiplication.

Return type
torch.Tensor

compute_sigma_qq(*hq: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_qq component in Clifford multiplication, representing the interactions between the negative square terms in the Clifford algebra.

TODO: Add mathematical format for sphinx.

sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k sigma_{q} captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

for k in range(j + 1, q):
results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

Parameters

- **hq** (*torch.Tensor*) – The 'q' part of the head entity embedding in Clifford algebra.
- **rq** (*torch.Tensor*) – The 'q' part of the relation embedding in Clifford algebra.

Returns
sigma_qq – The sigma_qq component of the Clifford multiplication.

Return type
torch.Tensor

compute_sigma_pq(*, *hp: torch.Tensor*, *hq: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_pq component in Clifford multiplication, representing the interactions between the positive and negative square terms in the Clifford algebra.

TODO: Add mathematical format for sphinx.

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] # sigma_pq = torch.zeros(b, r, p, q) # for i in range(p): # for j in range(q): # sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i] # print(sigma_pq.shape)

Parameters

- **hp** (*torch.Tensor*) – The 'p' part of the head entity embedding in Clifford algebra.
- **hq** (*torch.Tensor*) – The 'q' part of the head entity embedding in Clifford algebra.
- **rp** (*torch.Tensor*) – The 'p' part of the relation embedding in Clifford algebra.
- **rq** (*torch.Tensor*) – The 'q' part of the relation embedding in Clifford algebra.

Returns
sigma_pq – The sigma_pq component of the Clifford multiplication.

Return type
torch.Tensor

apply_coefficients(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTe

Applies scaling coefficients to the base vectors in the Clifford algebra. This method is used for adjusting the contributions of different components in the algebra.

Parameters

- **h0** (*torch.Tensor*) – The scalar part of the head entity embedding.
- **hp** (*torch.Tensor*) – The 'p' part of the head entity embedding.
- **hq** (*torch.Tensor*) – The 'q' part of the head entity embedding.
- **r0** (*torch.Tensor*) – The scalar part of the relation embedding.
- **rp** (*torch.Tensor*) – The 'p' part of the relation embedding.
- **rq** (*torch.Tensor*) – The 'q' part of the relation embedding.

Returns
Tuple containing the scaled components of the head and relation embeddings.

Return type
tuple

clifford_multiplication(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTe

Performs Clifford multiplication of head and relation embeddings. This method computes the various components of the Clifford product, combining the scalar, 'p', and 'q' parts of the embeddings.

TODO: Add mathematical format for sphinx.

h = h_0 + sum_{i=1}^p h_i e_i + sum_{j=p+1}^{p+q} h_j e_j r = r_0 + sum_{i=1}^p r_i e_i + sum_{j=p+1}^{p+q} r_j e_j

ei ^2 = +1 for i =< i =< p ej ^2 = -1 for p < j =< p+q ei ej = -eje1 for i

eq j

h r = sigma_0 + sigma_p + sigma_q + sigma_{pp} + sigma_{q}+ sigma_{pq} where

(1) sigma_0 = h_0 r_0 + sum_{i=1}^p (h_0 r_i) e_i - sum_{j=p+1}^{p+q} (h_j r_j) e_j

(2) sigma_p = sum_{i=1}^p (h_0 r_i + h_i r_0) e_i

(3) sigma_q = sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j

(4) sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k

(5) sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k

(6) sigma_{pq} = sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

h0
[torch.Tensor] The scalar part of the head entity embedding.

hp
[torch.Tensor] The 'p' part of the head entity embedding.

hq
[torch.Tensor] The 'q' part of the head entity embedding.

r0
[torch.Tensor] The scalar part of the relation embedding.

rp
[torch.Tensor] The 'p' part of the relation embedding.

rq
[torch.Tensor] The 'q' part of the relation embedding.

tuple
Tuple containing the components of the Clifford product.

construct_cl_multivector (*x: torch.FloatTensor*, *r: int*, *p: int*, *q: int*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors Cl_{p,q}(mathbb{R}^d)

Parameter

x
[torch.FloatTensor] The embedding vector with shape (n, d).

r
[int] The dimension of the scalar part.

p
[int] The number of positive square terms.

q
[int] The number of negative square terms.

returns

- **a0** (*torch.FloatTensor*) – Tensor with (n,r) shape
- **ap** (*torch.FloatTensor*) – Tensor with (n,r,p) shape
- **aq** (*torch.FloatTensor*) – Tensor with (n,r,q) shape

forward_k_vs_with_explicit (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities using explicit Clifford multiplication. This method is used for K-vs-All training and evaluation.

Parameters
x (*torch.Tensor*) – Tensor representing a batch of head entities and relations.

Returns
A tensor containing scores for each triple against all entities.

Return type
torch.FloatTensor

k_vs_all_score (*bpe_head_ent_emb: torch.Tensor*, *bpe_rel_ent_emb: torch.Tensor*, *E: torch.Tensor*)
→ torch.FloatTensor

Computes scores for all triples using Clifford multiplication in a K-vs-All setup. This method involves constructing multivectors for head entities and relations in Clifford algebra, applying coefficients, and computing interaction scores based on different components of the Clifford algebra.

Parameters

- **bpe_head_ent_emb** (*torch.Tensor*) – Batch of head entity embeddings in BPE (Byte Pair Encoding) format. Tensor shape: (batch_size, embedding_dim).
- **bpe_rel_ent_emb** (*torch.Tensor*) – Batch of relation embeddings in BPE format. Tensor shape: (batch_size, embedding_dim).
- **E** (*torch.Tensor*) – Tensor containing all entity embeddings. Tensor shape: (num_entities, embedding_dim).

Returns
Tensor containing the scores for each triple in the K-vs-All setting. Tensor shape: (batch_size, num_entities).

Return type
torch.FloatTensor

Notes

The method computes scores based on the basis of 1 (scalar part), the bases of 'p' (positive square terms), and the bases of 'q' (negative square terms). Additional computations involve sigma_pp, sigma_qq, and sigma_pq components in Clifford multiplication, corresponding to different interaction terms.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

TODO: Add mathematical format for sphinx. Performs the forward pass for K-vs-All training and evaluation in knowledge graph embeddings. This method involves retrieving real-valued embedding vectors for head entities and relations mathbb{R}^d, constructing Clifford algebra multivectors for these embeddings according to Cl_{p,q}(mathbb{R}^d), performing Clifford multiplication, and computing the inner product with all entity embeddings.

Parameters
x (*torch.Tensor*) – A tensor representing a batch of head entities and relations for the K-vs-All evaluation. Expected tensor shape: (n, 2), where 'n' is the batch size and '2' represents head entity and relation pairs.

Returns
A tensor containing the scores for each head entity and relation pair against all possible tail entities in the knowledge graph. Tensor shape: (n, |E|), where '|E|' is the number of entities in the knowledge graph.

Return type
torch.FloatTensor

Notes

This method is similar to the 'forward_k_vs_with_explicit' function in functionality. It is typically used in scenarios where every possible combination of a head entity and a relation is scored against all tail entities, commonly used in knowledge graph completion tasks.

forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*)
→ torch.FloatTensor

TODO: Add mathematical format for sphinx.

Performs the forward pass for K-vs-Sample training in knowledge graph embeddings. This method involves retrieving real-valued embedding vectors for head entities and relations mathbb{R}^d, constructing Clifford algebra multivectors for these embeddings according to Cl_{p,q}(mathbb{R}^d), performing Clifford multiplication, and computing the inner product with a sampled subset of entity embeddings.

Parameters

- **x** (*torch.LongTensor*) – A tensor representing a batch of head entities and relations for the K-vs-Sample evaluation. Expected tensor shape: (n, 2), where 'n' is the batch size and '2' represents head entity and relation pairs.
- **target_entity_idx** (*torch.LongTensor*) – A tensor of target entity indices for sampling in the K-vs-Sample evaluation. Tensor shape: (n, sample_size), where 'sample_size' is the number of entities sampled.

Returns

A tensor containing the scores for each head entity and relation pair against the sampled subset of tail entities. Tensor shape: (n, sample_size).

Return type

torch.FloatTensor

Notes

This method is used in scenarios where every possible combination of a head entity and a relation is scored against a sampled subset of tail entities, commonly used in knowledge graph completion tasks with a large number of entities.

score (*h: torch.Tensor*, *r: torch.Tensor*, *t: torch.Tensor*) → torch.FloatTensor

Computes the score for a given triple using Clifford multiplication in the context of knowledge graph embeddings. This method involves constructing Clifford algebra multivectors for head entities, relations, and tail entities, applying coefficients, and computing interaction scores based on different components of the Clifford algebra.

Parameters

- **h** (*torch.Tensor*) – Tensor representing the embeddings of head entities. Expected shape: (n, d), where 'n' is the number of triples and 'd' is the embedding dimension.
- **r** (*torch.Tensor*) – Tensor representing the embeddings of relations. Expected shape: (n, d).
- **t** (*torch.Tensor*) – Tensor representing the embeddings of tail entities. Expected shape: (n, d).

Returns

Tensor containing the scores for each triple. Tensor shape: (n,).

Return type

torch.FloatTensor

Notes

The method computes scores based on the scalar part, the bases of 'p' (positive square terms), and the bases of 'q' (negative square terms) in Clifford algebra. It includes additional computations involving sigma_pp, sigma_qq, and sigma_pq components, which correspond to different interaction terms in the Clifford product.

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using Clifford multiplication. This method is involved in the forward pass of the model during training or evaluation. It retrieves embeddings for head entities, relations, and tail entities, constructs Clifford algebra multivectors, applies coefficients, and computes interaction scores based on different components of Clifford algebra.

Parameters
x (*torch.Tensor*) – A tensor representing a batch of triples. Each triple consists of indices for a head entity, a relation, and a tail entity. Expected tensor shape: (n, 3), where 'n' is the number of triples.

Returns
A tensor containing the scores for each triple in the batch. Tensor shape: (n,), where 'n' is the number of triples.

Return type
torch.FloatTensor

Notes

The method computes scores based on the scalar part, the bases of 'p' (positive square terms), and the bases of 'q' (negative square terms) in Clifford algebra. It includes additional computations involving sigma_pp, sigma_qq, and sigma_pq components, corresponding to different interaction terms in the Clifford product.

class dicee.models.**KeciBase**(*args*)

Bases: *Keci*

Without learning dimension scaling

class dicee.models.**CMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

The CMult class represents a specific kind of mathematical object used in knowledge graph embeddings, involving Clifford algebra multiplication. It defines several algebraic structures based on the signature (p, q), such as Real Numbers, Complex Numbers, Quaternions, and others. The class provides functionality for performing Clifford multiplication, a generalization of the geometric product for vectors in a Clifford algebra.

TODO: Add mathematical format for sphinx.

Cl_(0,0) => Real Numbers

Cl_(0,1) =>
A multivector mathbf{a} = a_0 + a_1 e_1 A multivector mathbf{b} = b_0 + b_1 e_1

multiplication is isomorphic to the product of two complex numbers

mathbf{a} imes mathbf{b} = a_0 b_0 + a_0b_1 e1 + a_1 b_1 e_1 e_1
= (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) e_1

Cl_(2,0) =>
A multivector mathbf{a} = a_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 e_2 A multivector mathbf{b} = b_0 + b_1 e_1 + b_2 e_2 + b_{12} e_1 e_2

mathbf{a} imes mathbf{b} = a_0b_0 + a_0b_1 e_1 + a_0b_2e_2 + a_0 b_12 e_1 e_2

- a_1 b_0 e_1 + a_1b_1 e_1_e1 ..

Cl_(0,2) => Quaternions

name

The name identifier for the CMult class.

Type
str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

p

Non-negative integer representing the number of positive square terms in the Clifford algebra.

Type
int

q

Non-negative integer representing the number of negative square terms in the Clifford algebra.

Type
int

clifford_mul (*x: torch.FloatTensor*, *y: torch.FloatTensor*, *p: int*, *q: int*) → tuple

Performs Clifford multiplication based on the given signature (p, q).

score (*head_ent_emb*, *rel_ent_emb*, *tail_ent_emb*) → torch.FloatTensor

Computes a scoring function for a head entity, relation, and tail entity embeddings.

forward_triples (*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples.

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities in the knowledge graph.

clifford_mul (*x: torch.FloatTensor*, *y: torch.FloatTensor*, *p: int*, *q: int*) → tuple

Performs Clifford multiplication in the Clifford algebra Cl_{p,q}. This method generalizes the geometric product of vectors in a Clifford algebra, handling different algebraic structures like real numbers, complex numbers, quaternions, etc., based on the signature (p, q).

Clifford multiplication Cl_{p,q} (mathbb{R})

ei ^2 = +1 for i =< i =< p ej ^2 = -1 for p < j =< p+q ei ej = -eje1 for i

eq j

x
[torch.FloatTensor] The first multivector operand with shape (n, d).

y
[torch.FloatTensor] The second multivector operand with shape (n, d).

p
[int] A non-negative integer representing the number of positive square terms in the Clifford algebra.

q
[int] A non-negative integer representing the number of negative square terms in the Clifford algebra.

tuple
The result of Clifford multiplication, a tuple of tensors representing the components of the resulting multivector.

score (*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes a scoring function for a given triple of head entity, relation, and tail entity embeddings. The method involves Clifford multiplication of the head entity and relation embeddings, followed by a calculation of the score with the tail entity embedding.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – Embedding of the head entity.
- **rel_ent_emb** (*torch.FloatTensor*) – Embedding of the relation.
- **tail_ent_emb** (*torch.FloatTensor*) – Embedding of the tail entity.

Returns
A tensor representing the score of the given triple.

Return type
torch.FloatTensor

forward_triples (*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples. This method is typically used in training or evaluation of knowledge graph embedding models. It applies Clifford multiplication to the embeddings of head entities and relations and then calculates the score with respect to the tail entity embeddings.

Parameters
x (*torch.LongTensor*) – A tensor with shape (n, 3) representing a batch of triples, where each triple consists of indices for a head entity, a relation, and a tail entity.

Returns
A tensor with shape (n,) containing the scores for each triple in the batch.

Return type
torch.FloatTensor

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities in the knowledge graph, often used in KvsAll evaluation. This method retrieves embeddings for heads and relations, performs Clifford multiplication, and then computes the inner product with all entity embeddings to get scores for every possible triple involving the given heads and relations.

Parameters
x (*torch.Tensor*) – A tensor with shape (n, 3) representing a batch of triples, where each triple consists of indices for a head entity and a relation. The tail entity is to be compared against all possible entities.

Returns
A tensor with shape (n,) containing scores for each triple against all possible tail entities.

Return type
torch.FloatTensor

class dicee.models.**DeCaL**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables
`training` (*`bool`*) – Boolean represents whether this module is in training or evaluation mode.

`forward_triples`(*x: torch.Tensor*) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,3) shape

rtype
torch.FloatTensor with (n) shape

`cl_pqr`(*a*)

Input: tensor(batch_size, emb_dim) —-> output: tensor with 1+p+q+r components with size (batch_size, emb_dim/(1+p+q+r)) each.

1) takes a tensor of size (batch_size, emb_dim), split it into 1 + p + q +r components, hence 1+p+q+r must be a divisor of the emb_dim. 2) Return a list of the 1+p+q+r components vectors, each are tensors of size (batch_size, emb_dim/(1+p+q+r))

`compute_sigmas_single`(*list_h_emb*, *list_r_emb*, *list_t_emb*)

here we compute all the sums with no others vectors interaction taken with the scalar product with t, that is, 1) s0 = h_0r_0t_0 2) s1 = sum_{i=1}^{p}h_ir_it_0 3) s2 = sum_{j=p+1}^{p+q}h_jr_jt_0 4) s3 = sum_{i=1}^{q}(h_0r_it_i + h_ir_0t_i) 5) s4 = sum_{i=p+1}^{p+q}(h_0r_it_i + h_ir_0t_i) 5) s5 = sum_{i=p+q+1}^{p+q+r}(h_0r_it_i + h_ir_0t_i)

and return:

*) sigma_0t = sigma_0 cdot t_0 = s0 + s1 -s2 *) s3, s4 and s5

`compute_sigmas_multivect`(*list_h_emb*, *list_r_emb*)

Here we compute and return all the sums with vectors interaction for the same and different bases.

For same bases vectors interaction we have

1) sigma_pp = sum_{i=1}^{p-1}sum_{i'=i+1}^{p}(h_ir_{i'}-h_{i'}r_i) (models the interactions between e_i and e_i' for 1 <= i, i' <= p)

2) sigma_qq = sum_{j=p+1}^{p+q-1}sum_{j'=j+1}^{p+q}(h_jr_{j'}-h_{j'} (models the interactions between e_j and e_j' for p+1 <= j, j' <= p+q)

3) sigma_rr = sum_{k=p+q+1}^{p+q+r-1}sum_{k'=k+1}^{p}(h_kr_{k'}-h_{k'}r_k) (models the interactions between e_k and e_k' for p+q+1 <= k, k' <= p+q+r)

For different base vector interactions, we have

4) sigma_pq = sum_{i=1}^{p}sum_{j=p+1}^{p+q}(h_ir_j - h_jr_i) (interactionsn between e_i and e_j for 1<=i <=p and p+1<= j <= p+q)

5) sigma_pr = sum_{i=1}^{p}sum_{k=p+q+1}^{p+q+r}(h_ir_k - h_kr_i) (interactionsn between e_i and e_k for 1<=i <=p and p+q+1<= k <= p+q+r)

6) sigma_qr = sum_{j=p+1}^{p+q}sum_{j=p+q+1}^{p+q+r}(h_jr_k - h_kr_j) (interactionsn between e_j and e_k for p+1 <= j <=p+q and p+q+1<= j <= p+q+r)

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Kvsall training

(1) Retrieve real-valued embedding vectors for heads and relations mathbb{R}^d .

(2) Construct head entity and relation embeddings according to Cl_{p,q}(mathbb{R}^d) .

(3) Perform Cl multiplication

(4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this funcitons are identical Parameter ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, |E|) shape

apply_coefficients (*h0*, *hp*, *hq*, *hk*, *r0*, *rp*, *rq*, *rk*)

Multiplying a base vector with its scalar coefficient

construct_cl_multivector (*x: torch.FloatTensor*, *re: int*, *p: int*, *q: int*, *r: int*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors Cl_{p,q,r}(mathbb{R}^d)

Parameter

x: torch.FloatTensor with (n,d) shape

returns

- **a0** (*torch.FloatTensor*)
- **ap** (*torch.FloatTensor*)
- **aq** (*torch.FloatTensor*)
- **ar** (*torch.FloatTensor*)

compute_sigma_pp (*hp*, *rp*)

sigma_{p,p}^* = sum_{i=1}^{p-1}sum_{i'=i+1}^{p}(x_iy_{i'}-x_{i'}y_i)

sigma_{pp} captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

for k in range(i + 1, p):
results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

compute_sigma_qq(*hq*, *rq*)

Compute sigma_{q,q}^* = sum_{j=p+1}^{p+q-1}sum_{j'=j+1}^{p+q}(x_jy_{j'}-x_{j'}y_j) Eq. 16 sigma_{q} captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

for k in range(j + 1, q):
results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

compute_sigma_rr(*hk*, *rk*)

sigma_{r,r}^* = sum_{k=p+q+1}^{p+q+r-1}sum_{k'=k+1}^{p}(x_ky_{k'}-x_{k'}y_k)

compute_sigma_pq(*, *hp*, *hq*, *rp*, *rq*)

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):
sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

print(sigma_pq.shape)

compute_sigma_pr(*, *hp*, *hk*, *rp*, *rk*)

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):
sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

print(sigma_pq.shape)

compute_sigma_qr(*, *hq*, *hk*, *rq*, *rk*)

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):
sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

print(sigma_pq.shape)

class dicee.models.**BaseKGE**(*args: dict*)

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_byte_pair_encoded_k_vs_all(*x: torch.LongTensor*)

Parameters

x (*B x 2 x T*) –

forward_byte_pair_encoded_triple(*x: Tuple[torch.LongTensor, torch.LongTensor]*)

Perform the forward pass for byte pair encoded triples.

Parameters

x (*Tuple[torch.LongTensor, torch.LongTensor]*) – The input tuple containing byte pair encoded entities and relations.

Returns

The output tensor containing the scores for the byte pair encoded triples.

Return type

torch.Tensor

init_params_with_sanity_checking()

forward(*x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor]*, *y_idx: torch.LongTensor = None*)

Perform the forward pass of the model.

Parameters

- **x** (*Union[torch.LongTensor, Tuple[torch.LongTensor, torch.LongTensor]]*) – The input tensor or a tuple containing the input tensor and target entity indexes.
- **y_idx** (*torch.LongTensor, optional*) – The target entity indexes (default is None).

Returns
The output of the forward pass.

Return type
Any

forward_triples(*x: torch.LongTensor*) → torch.Tensor

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

forward_k_vs_all(**args*, ***kwargs*)

Forward pass for K vs. All.

Raises
ValueError – This function is not implemented in the current model.

forward_k_vs_sample(**args*, ***kwargs*)

Forward pass for K vs. Sample.

Raises
ValueError – This function is not implemented in the current model.

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for the head and relation entities.

Parameters
indexed_triple (*torch.LongTensor*) – The indexes of the head and relation entities.

Returns
The representation for the head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_sentence_representation(*x: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Get the representation for a sentence.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The representation for the input sentence.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

get_bpe_head_and_relation_representation(*x: torch.LongTensor*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for BPE head and relation entities.

Parameters
x (*B x 2 x T*) –

Returns
The representation for BPE head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]

Get the entity and relation embeddings.

Returns
The entity and relation embeddings.

Return type
Tuple[np.ndarray, np.ndarray]

class dicee.models.**PykeenKGE**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

A class for using knowledge graph embedding models implemented in Pykeen.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, random seed, and model-specific kwargs.

name

The name identifier for the PykeenKGE model.

Type
str

model

The Pykeen model instance.

Type
pykeen.models.base.Model

loss_history

A list to store the training loss history.

Type
list

args

The arguments used to initialize the model.

Type
dict

entity_embeddings

Entity embeddings learned by the model.

Type
torch.nn.Embedding

relation_embeddings

Relation embeddings learned by the model.

Type
torch.nn.Embedding

interaction

Interaction module used by the Pykeen model.

Type
pykeen.nn.modules.Interaction

forward_k_vs_all(*x: torch.LongTensor*) → torch.FloatTensor

Compute scores for all entities given a batch of head entities and relations.

forward_triples(*x: torch.LongTensor*) → torch.FloatTensor

Compute scores for a batch of triples.

forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: int*)

Compute scores against a sampled subset of entities.

Notes

This class provides an interface for using knowledge graph embedding models implemented in Pykeen. It initializes Pykeen models based on the provided arguments and allows for scoring triples and conducting knowledge graph embedding experiments.

forward_k_vs_all(*x: torch.LongTensor*)

TODO: Format in Numpy-style documentation

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads and relations + apply Dropout & Normalization if given. h, r = self.get_head_relation_representation(x) # (2) Reshape (1). if self.last_dim > 0:

h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim)

(3) Reshape all entities. if self.last_dim > 0:

t = self.entity_embeddings.weight.reshape(self.num_entities, self.embedding_dim, self.last_dim)

else:
t = self.entity_embeddings.weight

(4) Call the score_t from interactions to generate triple scores. return self.interaction.score_t(h=h, r=r, all_entities=t, slice_size=1)

forward_triples(*x: torch.LongTensor*) → torch.FloatTensor

TODO: Format in Numpy-style documentation

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads, relations and tails and apply Dropout & Normalization if given. h, r, t = self.get_triple_representation(x) # (2) Reshape (1). if self.last_dim > 0:

h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim) t = t.reshape(len(x), self.embedding_dim, self.last_dim)

(3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice_size=None, slice_dim=0)

abstract forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: int*)

Forward pass for K vs. Sample.

Raises

ValueError – This function is not implemented in the current model.

class dicee.models.**BaseKGE**(*args: dict*)

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_byte_pair_encoded_k_vs_all(*x: torch.LongTensor*)

Parameters

x (*B x 2 x T*) –

forward_byte_pair_encoded_triple(*x: Tuple[torch.LongTensor, torch.LongTensor]*)

Perform the forward pass for byte pair encoded triples.

Parameters

x (*Tuple[torch.LongTensor, torch.LongTensor]*) – The input tuple containing byte pair encoded entities and relations.

Returns
The output tensor containing the scores for the byte pair encoded triples.

Return type
torch.Tensor

init_params_with_sanity_checking()

forward(*x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor]*, *y_idx: torch.LongTensor = None*)

Perform the forward pass of the model.

Parameters

- **x** (*Union[torch.LongTensor, Tuple[torch.LongTensor, torch.LongTensor]]*) – The input tensor or a tuple containing the input tensor and target entity indexes.
- **y_idx** (*torch.LongTensor, optional*) – The target entity indexes (default is None).

Returns
The output of the forward pass.

Return type
Any

forward_triples(*x: torch.LongTensor*) → torch.Tensor

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

forward_k_vs_all(**args*, ***kwargs*)

Forward pass for K vs. All.

Raises
ValueError – This function is not implemented in the current model.

forward_k_vs_sample(**args*, ***kwargs*)

Forward pass for K vs. Sample.

Raises
ValueError – This function is not implemented in the current model.

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple: torch.LongTensor*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for the head and relation entities.

Parameters
indexed_triple (*torch.LongTensor*) – The indexes of the head and relation entities.

Returns
The representation for the head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_sentence_representation(*x: torch.LongTensor*) → Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Get the representation for a sentence.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The representation for the input sentence.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

get_bpe_head_and_relation_representation(*x: torch.LongTensor*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for BPE head and relation entities.

Parameters
x (*B x 2 x T*) –

Returns
The representation for BPE head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]

Get the entity and relation embeddings.

Returns
The entity and relation embeddings.

Return type
Tuple[np.ndarray, np.ndarray]

class dicee.models.**FMult**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

FMult is a model for learning neural networks on knowledge graphs. It extends the base knowledge graph embedding model by integrating neural network computations with entity and relation embeddings. The model is designed to work with complex embeddings and utilizes a neural network-based approach for embedding interactions.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions and other model-specific parameters.

name

The name identifier for the FMult model.

Type
str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

k

Dimension size for reshaping weights in neural network layers.

Type
int

num_sample

The number of samples to consider in the model computations.

Type
int

gamma

Randomly initialized weights for the neural network layers.

Type
torch.Tensor

roots

Precomputed roots for Legendre polynomials.

Type
torch.Tensor

weights

Precomputed weights for Legendre polynomials.

Type
torch.Tensor

compute_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.FloatTensor

Computes the output of a two-layer neural network for given weights and input.

chain_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.Tensor

Chains two linear neural network layers for a given input.

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Performs a forward pass for a batch of triples and computes the embedding interactions.

compute_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.FloatTensor

Compute the output of a two-layer neural network.

Parameters

- **weights** (*torch.FloatTensor*) – The weights of the neural network, split into two sets for two layers.
- **x** (*torch.Tensor*) – The input tensor for the neural network.

Returns
The output tensor after passing through the two-layer neural network.

Return type
torch.FloatTensor

chain_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.Tensor

Chain two linear layers of a neural network for given weights and input.

Parameters

- **weights** (*torch.FloatTensor*) – The weights of the neural network, split into two sets for two layers.
- **x** (*torch.Tensor*) – The input tensor for the neural network.

Returns
The output tensor after chaining the two linear layers.

Return type
torch.Tensor

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Forward pass for a batch of triples to compute embedding interactions.

Parameters
idx_triple (*torch.Tensor*) – Tensor containing indices of triples.

Returns
The computed scores for the batch of triples.

Return type
torch.Tensor

class dicee.models.**GFMult**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

GFMult (Graph Function Multiplication) extends the base knowledge graph embedding model by integrating neural network computations with entity and relation embeddings. This model is designed to leverage the strengths of neural networks in capturing complex interactions within knowledge graphs.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, learning rate, and other model-specific parameters.

name

The name identifier for the GFMult model.

Type
str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

k

The dimension size for reshaping weights in neural network layers.

Type
int

num_sample

The number of samples to use in the model computations.

Type
int

roots

Precomputed roots for Legendre polynomials, repeated for each dimension.

Type
torch.Tensor

weights

Precomputed weights for Legendre polynomials.

Type
torch.Tensor

compute_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.FloatTensor

Computes the output of a two-layer neural network for given weights and input.

chain_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.Tensor

Chains two linear neural network layers for a given input.

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Performs a forward pass for a batch of triples and computes the embedding interactions.

compute_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.FloatTensor

Compute the output of a two-layer neural network.

Parameters

- **weights** (*torch.FloatTensor*) – The weights of the neural network, split into two sets for two layers.
- **x** (*torch.Tensor*) – The input tensor for the neural network.

Returns
The output tensor after passing through the two-layer neural network.

Return type
torch.FloatTensor

chain_func(*weights: torch.FloatTensor*, *x: torch.Tensor*) → torch.Tensor

Chain two linear layers of a neural network for given weights and input.

Parameters

- **weights** (*torch.FloatTensor*) – The weights of the neural network, split into two sets for two layers.
- **x** (*torch.Tensor*) – The input tensor for the neural network.

Returns
The output tensor after chaining the two linear layers.

Return type
torch.Tensor

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Forward pass for a batch of triples to compute embedding interactions.

Parameters
idx_triple (*torch.Tensor*) – Tensor containing indices of triples.

Returns
The computed scores for the batch of triples.

Return type
torch.Tensor

class dicee.models.**FMult2**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

FMult2 is a model for learning neural networks on knowledge graphs, offering enhanced capabilities for capturing complex interactions in the graph. It extends the base knowledge graph embedding model by integrating multi-layer neural network computations with entity and relation embeddings.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, learning rate, number of layers, and other model-specific parameters.

name

The name identifier for the FMult2 model.

Type
str

n_layers

Number of layers in the neural network.

Type
int

k

Dimension size for reshaping weights in neural network layers.

Type
int

n

The number of discrete points for computations.

Type
int

a

Lower bound of the range for discrete points.

Type
float

b

Upper bound of the range for discrete points.

Type
float

score_func

The scoring function used in the model.

Type
str

discrete_points

Tensor of discrete points used in the computations.

Type
torch.Tensor

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

build_func(*Vec: torch.Tensor*) → Tuple[List[torch.Tensor], torch.Tensor]

Constructs a multi-layer neural network from a vector representation.

build_chain_funcs(*list_Vec: List[torch.Tensor]*) → Tuple[List[torch.Tensor], torch.Tensor]

Builds chained functions from a list of vector representations.

compute_func(*W: List[torch.Tensor]*, *b: torch.Tensor*, *x: torch.Tensor*) → torch.FloatTensor

Computes the output of a multi-layer neural network.

function(*list_W: List[List[torch.Tensor]]*, *list_b: List[torch.Tensor]*)
→ Callable[[torch.Tensor], torch.Tensor]

Defines a function for neural network computation based on weights and biases.

trapezoid(*list_W: List[List[torch.Tensor]]*, *list_b: List[torch.Tensor]*) → torch.Tensor

Applies the trapezoidal rule for integration on the function output.

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Performs a forward pass for a batch of triples and computes the embedding interactions.

build_func(*Vec: torch.Tensor*) → Tuple[List[torch.Tensor], torch.Tensor]

Constructs a multi-layer neural network from a vector representation.

Parameters
Vec (*torch.Tensor*) – The vector representation from which the neural network is constructed.

Returns
A tuple containing the list of weight matrices for each layer and the bias vector.

Return type
Tuple[List[torch.Tensor], torch.Tensor]

build_chain_funcs(*list_Vec: List[torch.Tensor]*) → Tuple[List[torch.Tensor], torch.Tensor]

Builds chained functions from a list of vector representations. This method constructs a sequence of neural network layers and their corresponding biases based on the provided vector representations.

Each vector representation in the list is first transformed into a set of weights and biases for a neural network layer using the *build_func* method. The method then computes a chained multiplication of these weights, adjusted by biases, to form a composite neural network function.

Parameters
list_Vec (*List[torch.Tensor]*) – A list of vector representations, each corresponding to a set of parameters for constructing a neural network layer.

Returns
A tuple where the first element is a list of weight tensors for each layer of the composite neural network, and the second element is the bias tensor for the last layer in the list.

Return type
Tuple[List[torch.Tensor], torch.Tensor]

Notes

This method is specifically designed to work with the neural network architecture defined in the FMult2 model. It assumes that each vector in *list_Vec* can be decomposed into weights and biases suitable for a layer in a neural network.

compute_func (*W: List[torch.Tensor]*, *b: torch.Tensor*, *x: torch.Tensor*) → torch.FloatTensor

Computes the output of a multi-layer neural network defined by the given weights and bias.

This method sequentially applies a series of matrix multiplications and non-linear transformations to an input tensor *x*, using the provided weights *W*. The method alternates between applying a non-linear function (tanh) and a linear transformation to the intermediate outputs. The final output is adjusted with a bias term *b*.

Parameters

- **W** (*List[torch.Tensor]*) – A list of weight tensors for each layer in the neural network. Each tensor in the list represents the weights of a layer.
- **b** (*torch.Tensor*) – The bias tensor to be added to the output of the final layer.
- **x** (*torch.Tensor*) – The input tensor to be processed by the neural network.

Returns
The output tensor after processing by the multi-layer neural network.

Return type
torch.FloatTensor

Notes

The method assumes an odd-indexed layer applies a non-linearity (tanh), while even-indexed layers apply linear transformations. This design choice is based on empirical observations for better performance in the context of the FMult2 model.

function (*list_W: List[List[torch.Tensor]]*, *list_b: List[torch.Tensor]*)
→ Callable[[torch.Tensor], torch.Tensor]

Defines a function that computes the output of a composite neural network. This higher-order function returns a callable that applies a sequence of transformations defined by the provided weights and biases.

The returned function (f) takes an input tensor x and applies a series of neural network computations on it. If only one set of weights and biases is provided, it directly computes the output using *compute_func*. Otherwise, it sequentially multiplies the outputs of multiple calls to *compute_func*, each using a different set of weights and biases from *list_W* and *list_b*.

Parameters

- **list_W** (*List[List[torch.Tensor]]*) – A list where each element is a list of weight tensors for a neural network.
- **list_b** (*List[torch.Tensor]*) – A list of bias tensors corresponding to each set of weights in *list_W*.

Returns
A function that takes an input tensor and returns the output of the composite neural network.

Return type
Callable[[torch.Tensor], torch.Tensor]

Notes

This method is part of the FMult2 model's approach to construct complex scoring functions for knowledge graph embeddings. The flexibility in combining multiple neural network layers enables capturing intricate patterns in the data.

trapezoid(*list_W: List[List[torch.Tensor]]*, *list_b: List[torch.Tensor]*) → torch.Tensor

Computes the integral of the output of a composite neural network function over a range of discrete points using the trapezoidal rule.

This method first constructs a composite neural network function using the *function* method with the provided weights *list_W* and biases *list_b*. It then evaluates this function at a series of discrete points (*self.discrete_points*) and applies the trapezoidal rule to approximate the integral of the function over these points. The sum of the integral approximations across all dimensions is returned.

Parameters

- **list_W** (*List[List[torch.Tensor]]*) – A list where each element is a list of weight tensors for a neural network.
- **list_b** (*List[torch.Tensor]*) – A list of bias tensors corresponding to each set of weights in *list_W*.

Returns
The sum of the integral of the composite function's output over the range of discrete points, computed using the trapezoidal rule.

Return type
torch.Tensor

Notes

The trapezoidal rule is a numerical method to approximate definite integrals. In the context of the FMult2 model, this method is used to integrate the output of the neural network over a range of inputs, which is crucial for certain types of calculations in knowledge graph embeddings.

forward_triples(*idx_triple: torch.Tensor*) → torch.Tensor

Forward pass for a batch of triples to compute embedding interactions.

Parameters
idx_triple (*torch.Tensor*) – Tensor containing indices of triples.

Returns
The computed scores for the batch of triples.

Return type
torch.Tensor

class dicee.models.**LFMult1**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Embedding with trigonometric functions. We represent all entities and relations in the complex number space as: f(x) = sum_{k=0}^{k=d-1}wk e^{kix}. and use the three differents scoring function as in the paper to evaluate the score

forward_triples(*idx_triple*)

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

tri_score(*h*, *r*, *t*)

vtp_score(*h*, *r*, *t*)

class dicee.models.**LFMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Embedding with polynomial functions. We represent all entities and relations in the polynomial space as: f(x) = sum_{i=0}^{d-1} a_k x^{i%d} and use the three differents scoring function as in the paper to evaluate the score. We also consider combining with Neural Networks.

forward_triples(*idx_triple*)

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

construct_multi_coeff(*x*)

poly_NN(*x*, *coefh*, *coefr*, *coeft*)

Constructing a 2 layers NN to represent the embeddings. h = sigma(wh^T x + bh), r = sigma(wr^T x + br), t = sigma(wt^T x + bt)

linear(*x*, *w*, *b*)

scalar_batch_NN(*a*, *b*, *c*)

element wise multiplication between a,b and c: Inputs : a, b, c ====> torch.tensor of size batch_size x m x d Output : a tensor of size batch_size x d

tri_score(*coeff_h*, *coeff_r*, *coeff_t*)

this part implement the trilinear scoring techniques:

score(h,r,t) = int_{0}{1} h(x)r(x)t(x) dx = sum_{i,j,k = 0}^{d-1} dfrac{a_i*b_j*c_k}{1+(i+j+k)%d}

1. generate the range for i,j and k from [0 d-1]
2. perform dfrac{a_i*b_j*c_k}{1+(i+j+k)%d} in parallel for every batch

3. take the sum over each batch

vtp_score(*h*, *r*, *t*)

this part implement the vector triple product scoring techniques:

score(h,r,t) = int_{0}{1} h(x)r(x)t(x) dx = sum_{i,j,k = 0}^{d-1} dfrac{a_i*c_j*b_k - b_i*c_j*a_k}{(1+(i+j)%d)(1+k)}

1. generate the range for i,j and k from [0 d-1]
2. Compute the first and second terms of the sum
3. Multiply with then denominator and take the sum
4. take the sum over each batch

comp_func(*h*, *r*, *t*)

this part implement the function composition scoring techniques: i.e. score = <hor, t>

polynomial(*coeff*, *x*, *degree*)

This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,…d] and return a vector tensor (coeff[0][0] + coeff[0][1]x +…+ coeff[0][d]x^d,

coeff[1][0] + coeff[1][1]x +…+ coeff[1][d]x^d)

pop(*coeff*, *x*, *degree*)

This function allow us to evaluate the composition of two polynomes without for loops :) it takes a matrix tensor of coefficients (coeff), a matrix tensor of points x and range of integer [0,1,…d]

and return a tensor (coeff[0][0] + coeff[0][1]x +…+ coeff[0][d]x^d,

coeff[1][0] + coeff[1][1]x +…+ coeff[1][d]x^d)

dicee.read_preprocess_save_load_kg

Submodules

dicee.read_preprocess_save_load_kg.preprocess

Module Contents

Classes

PreprocessKG	Preprocess the data in memory for a knowledge graph.

class dicee.read_preprocess_save_load_kg.preprocess.**PreprocessKG**(*kg*)

Preprocess the data in memory for a knowledge graph.

This class handles the preprocessing of the knowledge graph data which includes reading the data, adding noise or reciprocal triples, constructing vocabularies, and indexing datasets based on the backend being used.

kg

An instance representing the knowledge graph.

Type
object

start() → None

Preprocess train, valid, and test datasets stored in the knowledge graph instance.

This method applies the appropriate preprocessing technique based on the backend specified in the knowledge graph instance.

Parameters
None –

Return type
None

Raises
KeyError – If the specified backend is not supported.

preprocess_with_byte_pair_encoding() → None

Preprocess the datasets using byte-pair encoding (BPE).

This method applies byte-pair encoding to the raw training, validation, and test sets of the knowledge graph. It transforms string representations of entities and relations into sequences of subword tokens. The method also handles padding of these sequences and constructs the necessary mappings for entities and relations.

Parameters
None –

Return type
None

Notes

- Byte-pair encoding is used to handle the out-of-vocabulary problem in natural language

processing by splitting words into more frequently occurring subword units. - This method modifies the knowledge graph instance in place by setting various attributes related to the byte-pair encoding such as padded sequences, mappings, and the maximum length of subword tokens. - The method assumes that the raw datasets are available as Pandas DataFrames within the knowledge graph instance. - If the 'add_reciprical' flag is set in the knowledge graph instance, reciprocal triples are added to the datasets. - After encoding and padding, the method also constructs mappings from the subword token sequences to their corresponding integer indices.

preprocess_with_byte_pair_encoding_with_padding() → None

preprocess_with_pandas() → None

Preprocess train, valid, and test datasets stored in the knowledge graph instance using pandas.

This method involves adding reciprocal or noisy triples, constructing vocabularies for entities and relations, and indexing the datasets. The preprocessing is performed using the pandas library, which facilitates the handling and transformation of the data.

Parameters
None –

Return type
None

Notes

- The method begins by optionally adding reciprocal or noisy triples to the raw training, validation, and test sets.
- Sequential vocabulary construction is performed to create a bijection mapping of entities and relations to integer indices.
- The datasets (train, valid, test) are then indexed based on these mappings.
- The method modifies the knowledge graph instance in place by setting various attributes such as the indexed datasets,

the number of entities, and the number of relations. - The method assumes that the raw datasets are available as pandas DataFrames within the knowledge graph instance. - This preprocessing is crucial for converting the raw string-based datasets into a numerical format suitable for training machine learning models.

preprocess_with_polars() → None

Preprocess train, valid, and test datasets stored in the knowledge graph instance using Polars.

This method involves preprocessing the datasets with the Polars library, which is designed for efficient data manipulation and indexing. The process includes adding reciprocal triples, indexing entities and relations, and transforming the datasets from string-based to integer-based formats.

Parameters
None –

Return type
None

Notes

- The method begins by adding reciprocal triples to the raw datasets if the 'add_reciprical' flag is set

in the knowledge graph instance. - It then constructs a bijection mapping from entities and relations to integer indices, using the unique entities and relations found in the concatenated datasets. - The datasets (train, valid, test) are indexed based on these mappings and converted to NumPy arrays. - The method updates the knowledge graph instance by setting attributes such as the number of entities, the number of relations, and the indexed datasets. - Polars is used for its performance advantages in handling large datasets and its efficient data manipulation capabilities. - This preprocessing step is crucial for converting the raw string-based datasets into a numerical format suitable for training machine learning models.

sequential_vocabulary_construction() → None

Construct sequential vocabularies for entities and relations in the knowledge graph.

This method processes the raw training, validation, and test sets to create sequential mappings (bijection) of entities and relations to integer indices. These mappings are essential for converting the string-based representations of entities and relations to numerical formats that can be processed by machine learning models.

Parameters
None –

Return type
None

Notes

- The method first concatenates the raw datasets and then creates unique lists of all entities and relations.
- It then assigns a unique integer index to each entity and relation, creating two dictionaries:

'entity_to_idx' and 'relation_to_idx'. - These dictionaries are used to index entities and relations in the knowledge graph. - The method updates the knowledge graph instance by setting attributes such as 'entity_to_idx', 'relation_to_idx', 'num_entities', and 'num_relations'. - This method is a crucial preprocessing step for transforming knowledge graph data into a format suitable for training and evaluating machine learning models. - The method assumes that the raw datasets are available as Pandas DataFrames within the knowledge graph instance.

remove_triples_from_train_with_condition()

Remove specific triples from the training set based on a predefined condition.

This method filters out triples from the raw training dataset of the knowledge graph based on a condition, such as the frequency of entities or relations. This is often used to refine the training data, for instance, by removing infrequent entities or relations that may not be significant for the model's training.

Parameters
None –

Return type
None

Notes

- The method specifically targets the removal of triples that contain entities or relations

occurring below a certain frequency threshold. - The frequency threshold is determined by the 'min_freq_for_vocab' attribute of the knowledge graph instance. - The method updates the knowledge graph instance by modifying the 'raw_train_set' attribute, which holds the raw training dataset. - This preprocessing step is crucial for ensuring the quality of the training data and can impact the performance and generalization ability of the resulting machine learning models. - The method assumes that the raw training dataset is available as a Pandas DataFrame within the knowledge graph instance.

dicee.read_preprocess_save_load_kg.read_from_disk

Module Contents

Classes

ReadFromDisk	Read the data from disk into memory.

class dicee.read_preprocess_save_load_kg.read_from_disk.**ReadFromDisk**(*kg*)

Read the data from disk into memory.

This class is responsible for loading a knowledge graph from various sources such as disk files, triple stores, or SPARQL endpoints, and then making it available in memory for further processing.

kg

An instance representing the knowledge graph.

Type
object

start() → None

Read a knowledge graph from disk into memory.

add_noisy_triples_into_training() → None

Add noisy triples into the training set of the knowledge graph.

start() → None

Read a knowledge graph from disk into memory.

This method reads the knowledge graph data from the specified source (disk, triple store, or SPARQL endpoint) and loads it into memory. The data is made available in the train_set, test_set, and valid_set attributes of the knowledge graph instance.

Parameters
None –

Return type
None

Raises
RuntimeError – If the data source is invalid or not specified correctly.

add_noisy_triples_into_training() → None

Add noisy triples into the training set of the knowledge graph.

This method injects a specified proportion of noisy triples into the training set. Noisy triples are randomly generated by shuffling the entities and relations in the knowledge graph. The purpose of adding noisy triples is often to test the robustness of the model or to augment the training data.

Parameters
None –

Return type
None

Notes

The number of noisy triples added is determined by the 'add_noise_rate' attribute of the knowledge graph. The method ensures that the total number of triples (original plus noisy) in the training set matches the expected count after adding the noisy triples.

dicee.read_preprocess_save_load_kg.save_load_disk

Module Contents

Classes

LoadSaveToDisk	Handle the saving and loading of a knowledge graph to and from disk.

class dicee.read_preprocess_save_load_kg.save_load_disk.**LoadSaveToDisk**(*kg*)

Handle the saving and loading of a knowledge graph to and from disk.

This class provides functionality to serialize and deserialize the components of a knowledge graph, such as entity and relation indices, datasets, and byte-pair encoding mappings, to and from disk storage.

kg

An instance of the knowledge graph to be saved or loaded.

Type
object

save() → None

Save the knowledge graph components to disk.

load() → None

Load the knowledge graph components from disk.

save()

Save the knowledge graph components to disk.

This method serializes various components of the knowledge graph such as entity and relation indices, datasets, and byte-pair encoding mappings, and saves them to the specified file paths in the knowledge graph instance. The method handles different data types and structures based on the configuration of the knowledge graph.

Parameters
None –

Return type
None

Raises
AssertionError – If the path for serialization is not set or other required conditions are not met.

Notes

- The method checks if the 'path_for_serialization' attribute is set in the knowledge graph instance.
- Depending on the configuration (e.g., whether byte-pair encoding is used), different components are saved.
- The method uses custom functions like 'save_pickle' and 'save_numpy_ndarray' for serialization.

load()

Load the knowledge graph components from disk.

This method deserializes various components of the knowledge graph such as entity and relation indices, datasets, and byte-pair encoding mappings from the specified file paths in the knowledge graph instance. The method reconstructs the knowledge graph instance with the loaded data.

Parameters
None –

Return type
None

Raises

AssertionError – If the path for deserialization is not set or other required conditions are not met.

Notes

- The method checks if the 'path_for_deserialization' attribute is set in the knowledge graph instance.
- The method updates the knowledge graph instance with the loaded components.
- The method uses custom functions like 'load_pickle' and 'load_numpy_ndarray' for deserialization.
- If evaluation models are used, additional components like vocabularies and constraints are also loaded.

dicee.read_preprocess_save_load_kg.util

Module Contents

Functions

apply_reciprical_or_noise(→ Union[pandas.DataFrame, None])	Add reciprocal triples to the knowledge graph dataset.
timeit(→ Callable)	A decorator to measure the execution time of a function.
read_with_polars(→ polars.DataFrame)	Load and preprocess a dataset using Polars.
read_with_pandas(data_path[, read_only_few, ...])	
read_from_disk(→ Union[pandas.DataFrame, ...])	Load and preprocess a dataset from disk using specified backend.
read_from_triple_store(→ pandas.DataFrame)	Read triples from a SPARQL endpoint (triple store) and load them into a Pandas DataFrame.
get_er_vocab(→ collections.defaultdict)	Create a vocabulary mapping from (entity, relation) pairs to lists of tail entities.
get_re_vocab(→ collections.defaultdict)	Create a vocabulary mapping from (relation, tail entity) pairs to lists of head entities.
get_ee_vocab(→ collections.defaultdict)	Create a vocabulary mapping from (head entity, tail entity) pairs to lists of relations.
create_constraints(→ Tuple[dict, dict])	Create domain and range constraints for each relation in a set of triples.
load_with_pandas(→ None)	Deserialize data and load it into the knowledge graph instance using Pandas.
save_numpy_ndarray(*, data, file_path)	Save a numpy ndarray to disk.
load_numpy_ndarray(→ numpy.ndarray)	Load a numpy ndarray from a file.
save_pickle(*, data, file_path)	Serialize an object and save it to a file using pickle.
load_pickle(→ Any)	Load data from a pickle file.
create_reciprocal_triples(→ pandas.DataFrame)	Add inverse triples to a DataFrame of knowledge graph triples.
index_triples_with_pandas(→ pandas.DataFrame)	Index knowledge graph triples in a pandas DataFrame using provided entity and relation mappings.
dataset_sanity_checking(→ None)	Perform sanity checks on a knowledge graph dataset.

dicee.read_preprocess_save_load_kg.util.**apply_reciprical_or_noise**(*add_reciprical: bool*, *eval_model: str*, *df: pandas.DataFrame = None*, *info: str = None*) → pandas.DataFrame | None

Add reciprocal triples to the knowledge graph dataset.

This function augments a dataset by adding reciprocal triples. For each triple (s, p, o) in the dataset, it adds a reciprocal triple (o, p_inverse, s). This augmentation is often used in knowledge graph embedding models to improve the learning of relation patterns.

Parameters

- **add_reciprical** (*bool*) – A flag indicating whether to add reciprocal triples.
- **eval_model** (*str*) – The name of the evaluation model being used, which determines whether the reciprocal triples are required.
- **df** (*pd.DataFrame, optional*) – A pandas DataFrame containing the original triples of the knowledge graph. Each row should represent a triple (subject, predicate, object).
- **info** (*str, optional*) – An informational string describing the dataset being processed (e.g., 'train', 'test').

Returns

The augmented dataset with reciprocal triples added if the conditions are met. Returns the original DataFrame if conditions are not met, or None if the input DataFrame is None.

Return type

Union[pd.DataFrame, None]

Notes

- The function checks if both 'add_reciprical' and 'eval_model' are set to truthy values before proceeding with the addition of reciprocal triples.
- If 'df' is None, the function returns None, indicating that no dataset was provided for processing.
- The reciprocal triples are created using a custom function 'create_reciprocal_triples'.

dicee.read_preprocess_save_load_kg.util.**timeit**(*func*) → Callable

A decorator to measure the execution time of a function.

This decorator, when applied to a function, logs the time taken by the function to execute. It uses *time.perf_counter()* for precise time measurement. The decorator also reports the memory usage of the process at the time of the function's execution completion.

Parameters

func (*Callable*) – The function to be decorated.

Returns

The decorated function with added execution time and memory usage logging.

Return type

Callable

Notes

- The decorator uses *functools.wraps* to preserve the metadata of the original function.
- Time is measured using *time.perf_counter()*, which provides a higher resolution time measurement.
- Memory usage is obtained using *psutil.Process(os.getpid()).memory_info().rss*, which gives the resident set size.
- This decorator is useful for performance profiling and debugging.

dicee.read_preprocess_save_load_kg.util.**read_with_polars**(*data_path: str*, *read_only_few: int = None*, *sample_triples_ratio: float = None*) → polars.DataFrame

Load and preprocess a dataset using Polars.

This function reads a dataset from a specified file path using the Polars library. It can handle CSV, TXT, and Parquet file formats. The function also provides options to read only a subset of the data and to sample a fraction of the data. Additionally, it applies a heuristic to filter out triples with literal values in RDF knowledge graphs.

Parameters

- **data_path** (*str*) – The file path to the dataset. Supported formats include CSV, TXT, and Parquet.
- **read_only_few** (*int, optional*) – If specified, only this number of rows will be read from the dataset. Defaults to reading the entire dataset.
- **sample_triples_ratio** (*float, optional*) – If specified, a fraction of the dataset will be randomly sampled. For example, a value of 0.1 samples 10% of the data.

Returns
The loaded and optionally sampled dataset as a Polars DataFrame.

Return type
polars.DataFrame

Notes

- The function determines the file format based on the file extension.
- If 'sample_triples_ratio' is provided, the dataset is subsampled accordingly.
- A heuristic is applied to remove triples where the subject or object does not start with '<', which is common in RDF knowledge graphs to indicate entities.
- This function uses Polars for efficient data loading and manipulation, especially useful for large datasets.

dicee.read_preprocess_save_load_kg.util.**read_with_pandas**(*data_path*, *read_only_few: int = None*, *sample_triples_ratio: float = None*)

dicee.read_preprocess_save_load_kg.util.**read_from_disk**(*data_path: str*, *read_only_few: int = None*, *sample_triples_ratio: float = None*, *backend: str = None*) → pandas.DataFrame | polars.DataFrame | None

Load and preprocess a dataset from disk using specified backend.

This function reads a dataset from a specified file path, supporting different backends such as pandas, polars, and rdflib. It can handle various file formats including TTL, OWL, RDF/XML, and others. The function provides options to read only a subset of the data and to sample a fraction of the data.

Parameters

- **data_path** (*str*) – The file path to the dataset.
- **read_only_few** (*int, optional*) – If specified, only this number of rows will be read from the dataset. Defaults to reading the entire dataset.
- **sample_triples_ratio** (*float, optional*) – If specified, a fraction of the dataset will be randomly sampled.
- **backend** (*str*) – The backend to use for reading the dataset. Supported values are 'pandas', 'polars', and 'rdflib'.

Returns
The loaded dataset as a DataFrame, depending on the specified backend. Returns None if the file is not found.

Return type
Union[pd.DataFrame, polars.DataFrame, None]

Raises
- **RuntimeError** – If the data format is not compatible with the specified backend, or if the backend is not recognized.
- **AssertionError** – If the backend is not provided.

Notes

- The function automatically detects the data format based on the file extension.
- For RDF/XML, TTL, OWL, and similar formats, the 'rdflib' backend is required.
- This function is a general interface for loading datasets, allowing flexibility in choosing the backend based on data format and processing needs.

dicee.read_preprocess_save_load_kg.util.**read_from_triple_store**(*endpoint: str = None*) → pandas.DataFrame

Read triples from a SPARQL endpoint (triple store) and load them into a Pandas DataFrame.

This function executes a SPARQL query against a specified SPARQL endpoint to retrieve all triples in the store. The result is then formatted into a Pandas DataFrame for further processing or analysis.

Parameters
endpoint (*str*) – The URL of the SPARQL endpoint from which to retrieve triples.

Returns
A DataFrame containing the triples retrieved from the triple store, with columns 'subject', 'relation', and 'object'.

Return type
pd.DataFrame

Raises
AssertionError – If the 'endpoint' parameter is None or not a string, or if the response from the endpoint is not successful.

Notes

- The function sends a SPARQL query to the provided endpoint to retrieve all triples in the format {?subject ?predicate ?object}.
- The response is expected in JSON format, conforming to the SPARQL query results JSON format.
- This function is specifically designed for reading data from a SPARQL endpoint and requires an endpoint that responds to POST requests with SPARQL queries.

dicee.read_preprocess_save_load_kg.util.**get_er_vocab**(
data: Iterable[Tuple[Any, Any, Any]], *file_path: str = None*) → collections.defaultdict

Create a vocabulary mapping from (entity, relation) pairs to lists of tail entities.

This function processes a dataset of triples and constructs a mapping where each key is a tuple of (head entity, relation) and the corresponding value is a list of all tail entities associated with that (head entity, relation) pair. Optionally, this vocabulary can be saved to a file.

Parameters

- **data** (*Iterable[Tuple[Any, Any, Any]]*) – An iterable of triples, where each triple is a tuple (head entity, relation, tail entity).
- **file_path** (*str, optional*) – The file path where the vocabulary should be saved as a pickle file. If not provided, the vocabulary is not saved to disk.

Returns
A default dictionary where keys are (entity, relation) tuples and values are lists of tail entities.

Return type
defaultdict

Notes

- The function uses a *defaultdict* to handle keys that may not exist in the dictionary.
- It is useful for creating a quick lookup of all possible tail entities for given (entity, relation) pairs, which can be used in various knowledge graph tasks like link prediction.
- If 'file_path' is provided, the vocabulary is saved using the *save_pickle* function.

dicee.read_preprocess_save_load_kg.util.**get_re_vocab**(
data: Iterable[Tuple[Any, Any, Any]], *file_path: str = None*) → collections.defaultdict

Create a vocabulary mapping from (relation, tail entity) pairs to lists of head entities.

This function processes a dataset of triples and constructs a mapping where each key is a tuple of (relation, tail entity) and the corresponding value is a list of all head entities associated with that (relation, tail entity) pair. Optionally, this vocabulary can be saved to a file.

Parameters

- **data** (*Iterable[Tuple[Any, Any, Any]]*) – An iterable of triples, where each triple is a tuple (head entity, relation, tail entity).
- **file_path** (*str, optional*) – The file path where the vocabulary should be saved as a pickle file. If not provided, the vocabulary is not saved to disk.

Returns
A default dictionary where keys are (relation, tail entity) tuples and values are lists of head entities.

Return type
defaultdict

Notes

- The function uses a *defaultdict* to handle keys that may not exist in the dictionary.
- It is useful for creating a quick lookup of all possible head entities for given (relation, tail entity) pairs, which can be used in various knowledge graph tasks like link prediction.
- If 'file_path' is provided, the vocabulary is saved using the *save_pickle* function.

dicee.read_preprocess_save_load_kg.util.**get_ee_vocab**(
data: Iterable[Tuple[Any, Any, Any]], *file_path: str = None*) → collections.defaultdict

Create a vocabulary mapping from (head entity, tail entity) pairs to lists of relations.

This function processes a dataset of triples and constructs a mapping where each key is a tuple of (head entity, tail entity) and the corresponding value is a list of all relations that connect these two entities. Optionally, this vocabulary can be saved to a file.

Parameters

- **data** (*Iterable[Tuple[Any, Any, Any]]*) – An iterable of triples, where each triple is a tuple (head entity, relation, tail entity).
- **file_path** (*str, optional*) – The file path where the vocabulary should be saved as a pickle file. If not provided, the vocabulary is not saved to disk.

Returns
A default dictionary where keys are (head entity, tail entity) tuples and values are lists of relations.

Return type
defaultdict

Notes

- The function uses a *defaultdict* to handle keys that may not exist in the dictionary.
- This vocabulary is useful for tasks that require knowledge of all relations between specific pairs of entities, such as in certain types of link prediction or relation extraction tasks.
- If 'file_path' is provided, the vocabulary is saved using the *save_pickle* function.

dicee.read_preprocess_save_load_kg.util.**create_constraints**(
triples: numpy.ndarray, *file_path: str = None*) → Tuple[dict, dict]

Create domain and range constraints for each relation in a set of triples.

This function processes a dataset of triples and constructs domain and range constraints for each relation. The domain of a relation is defined as the set of all head entities that appear with that relation, and the range is defined as the set of all tail entities. The constraints are formed by finding entities that are not in the domain or range of each relation.

Parameters

- **triples** (*np.ndarray*) – A numpy array of triples, where each row is a triple (head entity, relation, tail entity).
- **file_path** (*str, optional*) – The file path where the constraints should be saved as a pickle file. If not provided, the constraints are not saved to disk.

Returns
A tuple containing two dictionaries. The first dictionary maps each relation to a list of entities not in its domain, and the second maps each relation to a list of entities not in its range.

Return type
Tuple[dict, dict]

Notes

- The function assumes that the input triples are in the form of a numpy array with three columns.
- The domain and range constraints are useful in tasks that require understanding the valid head and tail entities for each relation, such as in link prediction.
- If 'file_path' is provided, the constraints are saved using the *save_pickle* function.

dicee.read_preprocess_save_load_kg.util.**load_with_pandas**(*self*) → None

Deserialize data and load it into the knowledge graph instance using Pandas.

This method loads serialized data from disk, converting it into the appropriate data structures for use in the knowledge graph instance. It deserializes entity and relation mappings, training, validation, and test datasets, and constructs vocabularies and constraints necessary for the evaluation of the model.

Parameters
None –

Return type
None

Notes

- This method reads serialized data stored in Parquet format with gzip compression.
- It deserializes mappings for entities and relations into dictionaries for efficient access.
- Training, validation, and test sets are loaded into numpy arrays.
- If evaluation is enabled, vocabularies for entity-relation, relation-entity, and entity-entity pairs are created along with domain and range constraints for relations.
- This method handles the absence of validation or test sets gracefully, setting the corresponding attributes to None if the files are not found.
- Deserialization paths and progress are logged, including time taken for each step.

dicee.read_preprocess_save_load_kg.util.**save_numpy_ndarray**(*, *data: numpy.ndarray*, *file_path: str*)

Save a numpy ndarray to disk.

This function saves a given numpy ndarray to a specified file path using NumPy's binary format. The function is specifically designed to handle arrays with a shape (n, 3), typically representing triples in knowledge graphs.

Parameters

- **data** (*np.ndarray*) – A numpy ndarray to be saved, expected to have the shape (n, 3) where 'n' is the number of rows and 'd' is the number of columns (specifically 3).
- **file_path** (*str*) – The file path where the ndarray will be saved.

Raises

AssertionError – If the number of rows ‘n’ in ‘data’ is not positive or the number of columns ‘d’ is not equal to 3.

Notes

- The ndarray is saved in NumPy’s binary format (.npy file).
- This function is particularly useful for saving datasets of triples in knowledge graph applications.
- The file is opened in binary write mode and the data is saved using NumPy’s *save* function.

dicee.read_preprocess_save_load_kg.util.**load_numpy_ndarray**(*, *file_path: str*) → numpy.ndarray

Load a numpy ndarray from a file.

This function reads a numpy ndarray from a specified file path. The file is expected to be in NumPy’s binary format (.npy file). It’s commonly used to load datasets, especially in knowledge graph contexts.

Parameters

file_path (*str*) – The path of the file from which the ndarray will be loaded.

Returns

The numpy ndarray loaded from the specified file.

Return type

np.ndarray

Notes

- The function opens the file in binary read mode and loads the data using NumPy’s *load* function.
- This function is particularly useful for loading datasets of triples in knowledge graph applications or other numerical data saved in NumPy’s binary format.
- It’s important to ensure that the file at ‘file_path’ exists and is a valid NumPy binary file to avoid runtime errors.

dicee.read_preprocess_save_load_kg.util.**save_pickle**(*, *data: object*, *file_path: str*)

Serialize an object and save it to a file using pickle.

This function serializes a given Python object using the pickle protocol and saves it to the specified file path. It’s a general-purpose function that can be used to persist a wide range of Python objects.

Parameters

- **data** (*object*) – The Python object to be serialized and saved. This can be any object that is serializable by the pickle module.
- **file_path** (*str*) – The path of the file where the serialized object will be saved. The file will be created if it does not exist.

dicee.read_preprocess_save_load_kg.util.**load_pickle**(*file_path: str*) → Any

Load data from a pickle file.

Parameters

file_path (*str*) – The file path to the pickle file to be loaded.

Returns
The data loaded from the pickle file.

Return type
Any

dicee.read_preprocess_save_load_kg.util.**create_reciprocal_triples**(*x: pandas.DataFrame*) → pandas.DataFrame

Add inverse triples to a DataFrame of knowledge graph triples.

Parameters
x (*pd.DataFrame*) – The DataFrame containing knowledge graph triples with columns "subject," "relation," and "object."

Returns
A new DataFrame that includes the original triples and their inverse counterparts.

Return type
pd.DataFrame

Notes

This function takes a DataFrame of knowledge graph triples and adds their inverse triples to it. For each triple (s, r, o) in the input DataFrame, an inverse triple (o, r_inverse, s) is added to the output. The "relation" column of the inverse triples is created by appending "_inverse" to the original relation.

dicee.read_preprocess_save_load_kg.util.**index_triples_with_pandas**(*train_set: pandas.DataFrame*, *entity_to_idx: dict*, *relation_to_idx: dict*) → pandas.DataFrame

Index knowledge graph triples in a pandas DataFrame using provided entity and relation mappings.

Parameters

- **train_set** (*pd.DataFrame*) – A pandas DataFrame containing knowledge graph triples with columns "subject," "relation," and "object."
- **entity_to_idx** (*dict*) – A mapping from entity names (str) to integer indices.
- **relation_to_idx** (*dict*) – A mapping from relation names (str) to integer indices.

Returns
A new pandas DataFrame where the entities and relations in the original triples are replaced with their corresponding integer indices.

Return type
pd.DataFrame

Notes

This function takes a pandas DataFrame of knowledge graph triples, along with mappings from entity and relation names to integer indices. It replaces the entity and relation names in the DataFrame with their corresponding integer indices, effectively indexing the triples. The resulting DataFrame has the same structure as the input, with integer indices replacing entity and relation names.

dicee.read_preprocess_save_load_kg.util.**dataset_sanity_checking**(*train_set: numpy.ndarray*, *num_entities: int*, *num_relations: int*) → None

Perform sanity checks on a knowledge graph dataset.

Parameters

- **train_set** (*np.ndarray*) – The training dataset represented as a NumPy array. Each row represents a triple with columns “subject,” “relation,” and “object.”
- **num_entities** (*int*) – The total number of entities in the knowledge graph.
- **num_relations** (*int*) – The total number of relations in the knowledge graph.

Return type
None

Raises
AssertionError – If any of the sanity checks fail, assertions are raised to indicate potential issues in the dataset.

Notes

This function performs a series of sanity checks on a knowledge graph dataset to ensure its integrity and consistency. It checks the data type of the dataset, the number of columns, the size of the dataset, and the validity of entity and relation indices. If any of the checks fail, assertions are raised to signal potential problems in the dataset.

The checks performed include: - Verifying that the input dataset is a NumPy array. - Checking that the dataset has the correct number of columns (3 for subject, relation, and object). - Ensuring that the dataset size is greater than 0. - Validating that the maximum entity indices in the dataset do not exceed the specified number of entities. - Validating that the maximum relation index in the dataset does not exceed the specified number of relations.

Package Contents

Classes

PreprocessKG	Preprocess the data in memory for a knowledge graph.
LoadSaveToDisk	Handle the saving and loading of a knowledge graph to and from disk.
ReadFromDisk	Read the data from disk into memory.

class dicee.read_preprocess_save_load_kg.**PreprocessKG**(*kg*)

Preprocess the data in memory for a knowledge graph.

This class handles the preprocessing of the knowledge graph data which includes reading the data, adding noise or reciprocal triples, constructing vocabularies, and indexing datasets based on the backend being used.

kg

An instance representing the knowledge graph.

Type
object

start() → None

Preprocess train, valid, and test datasets stored in the knowledge graph instance.

This method applies the appropriate preprocessing technique based on the backend specified in the knowledge graph instance.

Parameters
None –

Return type
None

Raises
KeyError – If the specified backend is not supported.

preprocess_with_byte_pair_encoding() → None

Preprocess the datasets using byte-pair encoding (BPE).

This method applies byte-pair encoding to the raw training, validation, and test sets of the knowledge graph. It transforms string representations of entities and relations into sequences of subword tokens. The method also handles padding of these sequences and constructs the necessary mappings for entities and relations.

Parameters
None –

Return type
None

Notes

- Byte-pair encoding is used to handle the out-of-vocabulary problem in natural language

processing by splitting words into more frequently occurring subword units. - This method modifies the knowledge graph instance in place by setting various attributes related to the byte-pair encoding such as padded sequences, mappings, and the maximum length of subword tokens. - The method assumes that the raw datasets are available as Pandas DataFrames within the knowledge graph instance. - If the 'add_reciprical' flag is set in the knowledge graph instance, reciprocal triples are added to the datasets. - After encoding and padding, the method also constructs mappings from the subword token sequences to their corresponding integer indices.

preprocess_with_byte_pair_encoding_with_padding() → None

preprocess_with_pandas() → None

Preprocess train, valid, and test datasets stored in the knowledge graph instance using pandas.

This method involves adding reciprocal or noisy triples, constructing vocabularies for entities and relations, and indexing the datasets. The preprocessing is performed using the pandas library, which facilitates the handling and transformation of the data.

Parameters
None –

Return type
None

Notes

- The method begins by optionally adding reciprocal or noisy triples to the raw training, validation, and test sets.
- Sequential vocabulary construction is performed to create a bijection mapping of entities and relations to integer indices.
- The datasets (train, valid, test) are then indexed based on these mappings.
- The method modifies the knowledge graph instance in place by setting various attributes such as the indexed datasets,

the number of entities, and the number of relations. - The method assumes that the raw datasets are available as pandas DataFrames within the knowledge graph instance. - This preprocessing is crucial for converting the raw string-based datasets into a numerical format suitable for training machine learning models.

preprocess_with_polars() → None

Preprocess train, valid, and test datasets stored in the knowledge graph instance using Polars.

This method involves preprocessing the datasets with the Polars library, which is designed for efficient data manipulation and indexing. The process includes adding reciprocal triples, indexing entities and relations, and transforming the datasets from string-based to integer-based formats.

Parameters
None –

Return type
None

Notes

- The method begins by adding reciprocal triples to the raw datasets if the 'add_reciprical' flag is set

in the knowledge graph instance. - It then constructs a bijection mapping from entities and relations to integer indices, using the unique entities and relations found in the concatenated datasets. - The datasets (train, valid, test) are indexed based on these mappings and converted to NumPy arrays. - The method updates the knowledge graph instance by setting attributes such as the number of entities, the number of relations, and the indexed datasets. - Polars is used for its performance advantages in handling large datasets and its efficient data manipulation capabilities. - This preprocessing step is crucial for converting the raw string-based datasets into a numerical format suitable for training machine learning models.

sequential_vocabulary_construction() → None

Construct sequential vocabularies for entities and relations in the knowledge graph.

This method processes the raw training, validation, and test sets to create sequential mappings (bijection) of entities and relations to integer indices. These mappings are essential for converting the string-based representations of entities and relations to numerical formats that can be processed by machine learning models.

Parameters
None –

Return type
None

Notes

- The method first concatenates the raw datasets and then creates unique lists of all entities and relations.
- It then assigns a unique integer index to each entity and relation, creating two dictionaries:

'entity_to_idx' and 'relation_to_idx'. - These dictionaries are used to index entities and relations in the knowledge graph. - The method updates the knowledge graph instance by setting attributes such as 'entity_to_idx', 'relation_to_idx', 'num_entities', and 'num_relations'. - This method is a crucial preprocessing step for transforming knowledge graph data into a format suitable for training and evaluating machine learning models. - The method assumes that the raw datasets are available as Pandas DataFrames within the knowledge graph instance.

remove_triples_from_train_with_condition()

Remove specific triples from the training set based on a predefined condition.

This method filters out triples from the raw training dataset of the knowledge graph based on a condition, such as the frequency of entities or relations. This is often used to refine the training data, for instance, by removing infrequent entities or relations that may not be significant for the model's training.

Parameters
None –

Return type
None

Notes

- The method specifically targets the removal of triples that contain entities or relations

occurring below a certain frequency threshold. - The frequency threshold is determined by the 'min_freq_for_vocab' attribute of the knowledge graph instance. - The method updates the knowledge graph instance by modifying the 'raw_train_set' attribute, which holds the raw training dataset. - This preprocessing step is crucial for ensuring the quality of the training data and can impact the performance and generalization ability of the resulting machine learning models. - The method assumes that the raw training dataset is available as a Pandas DataFrame within the knowledge graph instance.

class dicee.read_preprocess_save_load_kg.**LoadSaveToDisk**(*kg*)

Handle the saving and loading of a knowledge graph to and from disk.

This class provides functionality to serialize and deserialize the components of a knowledge graph, such as entity and relation indices, datasets, and byte-pair encoding mappings, to and from disk storage.

kg

An instance of the knowledge graph to be saved or loaded.

Type
object

save() → None

Save the knowledge graph components to disk.

load() → None

Load the knowledge graph components from disk.

save()

Save the knowledge graph components to disk.

This method serializes various components of the knowledge graph such as entity and relation indices, datasets, and byte-pair encoding mappings, and saves them to the specified file paths in the knowledge graph instance. The method handles different data types and structures based on the configuration of the knowledge graph.

Parameters
None –

Return type
None

Raises
AssertionError – If the path for serialization is not set or other required conditions are not met.

Notes

- The method checks if the 'path_for_serialization' attribute is set in the knowledge graph instance.
- Depending on the configuration (e.g., whether byte-pair encoding is used), different components are saved.
- The method uses custom functions like 'save_pickle' and 'save_numpy_ndarray' for serialization.

load()

Load the knowledge graph components from disk.

This method deserializes various components of the knowledge graph such as entity and relation indices, datasets, and byte-pair encoding mappings from the specified file paths in the knowledge graph instance. The method reconstructs the knowledge graph instance with the loaded data.

Parameters
None –

Return type
None

Raises
AssertionError – If the path for deserialization is not set or other required conditions are not met.

Notes

- The method checks if the 'path_for_deserialization' attribute is set in the knowledge graph instance.
- The method updates the knowledge graph instance with the loaded components.
- The method uses custom functions like 'load_pickle' and 'load_numpy_ndarray' for deserialization.
- If evaluation models are used, additional components like vocabularies and constraints are also loaded.

class dicee.read_preprocess_save_load_kg.**ReadFromDisk**(*kg*)

Read the data from disk into memory.

This class is responsible for loading a knowledge graph from various sources such as disk files, triple stores, or SPARQL endpoints, and then making it available in memory for further processing.

kg

An instance representing the knowledge graph.

Type
object

start() → None

Read a knowledge graph from disk into memory.

add_noisy_triples_into_training() → None

Add noisy triples into the training set of the knowledge graph.

start() → None

Read a knowledge graph from disk into memory.

This method reads the knowledge graph data from the specified source (disk, triple store, or SPARQL endpoint) and loads it into memory. The data is made available in the train_set, test_set, and valid_set attributes of the knowledge graph instance.

Parameters
None –

Return type
None

Raises
RuntimeError – If the data source is invalid or not specified correctly.

add_noisy_triples_into_training() → None

Add noisy triples into the training set of the knowledge graph.

This method injects a specified proportion of noisy triples into the training set. Noisy triples are randomly generated by shuffling the entities and relations in the knowledge graph. The purpose of adding noisy triples is often to test the robustness of the model or to augment the training data.

Parameters
None –

Return type
None

Notes

The number of noisy triples added is determined by the 'add_noise_rate' attribute of the knowledge graph. The method ensures that the total number of triples (original plus noisy) in the training set matches the expected count after adding the noisy triples.

dicee.scripts

Submodules

dicee.scripts.index

Module Contents

Functions

get_default_arguments()
main()

dicee.scripts.index.**get_default_arguments**()

dicee.scripts.index.**main**()

dicee.scripts.run

Module Contents

Functions

get_default_arguments(→ arg-parse.Namespace)	Get default command-line arguments for the knowledge graph embedding execution.
main()	

dicee.scripts.run.**get_default_arguments**(*description: str | None = None*) → argparse.Namespace

Get default command-line arguments for the knowledge graph embedding execution.

This function returns a set of default command-line arguments that can be used to configure the knowledge graph embedding execution. It includes parameters related to dataset paths, model selection, training settings, optimization, and more.

Parameters

description (*str, optional*) – A description of the command-line arguments (default is None).

Returns

A namespace containing the default command-line arguments.

Return type

argparse.Namespace

dicee.scripts.run.**main**()

dicee.scripts.serve

Module Contents

Classes

NeuralSearcher	A class for performing neural-based vector search using a pre-trained model and a vector database.

Functions

get_default_arguments(→ arg-parse.Namespace)	Get default command-line arguments for a specific task.
root()	
search_embeddings(q)	
retrieve_embeddings(q)	
main()	

Attributes

app
neural_searcher

dicee.scripts.serve.**app**

dicee.scripts.serve.**neural_searcher**

dicee.scripts.serve.**get_default_arguments**() → argparse.Namespace

Get default command-line arguments for a specific task.

This function returns a set of default command-line arguments that are used for a specific task. The arguments include options for specifying the path to a pre-trained model, the name of a vector database collection, the location of the collection, host information, and port number.

Returns
A namespace containing the default command-line arguments.

Return type
argparse.Namespace

async dicee.scripts.serve.**root**()

async dicee.scripts.serve.**search_embeddings**(*q: str*)

async dicee.scripts.serve.**retrieve_embeddings**(*q: str*)

class dicee.scripts.serve.**NeuralSearcher**(*args*)

A class for performing neural-based vector search using a pre-trained model and a vector database.

This class is designed for searching for entities in a vector database using a neural network-based model. It initializes the model and the Qdrant client for performing vector searches.

Parameters
args (*argparse.Namespace*) – A namespace containing the configuration and settings for the searcher.

collection_name

The name of the vector database collection to perform searches in.

Type
str

model

An instance of the knowledge graph embedding model for encoding entities into vectors.

Type
KGE

qdrant_client

An instance of the Qdrant client for interacting with the vector database.

Type
QdrantClient

search(*entity: str*) → List[Dict[str, str | float]]

Search for the closest vectors to the input entity in the vector database.

search(*entity: str*) → List[Dict[str, str | float]]

Search for the closest vectors to the input entity in the vector database.

Parameters
entity (*str*) – The entity for which to find the closest matches in the database.

Returns
A list of dictionaries containing search results, where each dictionary has "hit" (str) and "score" (float) keys.

Return type
List[Dict[str, Union[str, float]]]

dicee.scripts.serve.**main**()

dicee.trainer

Submodules

dicee.trainer.dice_trainer

Module Contents

Classes

DICE_Trainer	Implements a training framework for knowledge graph embedding models using [PyTorch Lightning](https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html),

Functions

initialize_trainer(→ Any)	Initialize the trainer for knowledge graph embedding.
get_callbacks(→ List[Any])	Get a list of callback objects based on the specified training configuration.

dicee.trainer.dice_trainer.**initialize_trainer**(*args: Dict[str, Any]*, *callbacks: List[Any]*) → Any

Initialize the trainer for knowledge graph embedding.

This function initializes and returns a trainer object based on the specified training configuration.

Parameters

- **args** (*dict*) – A dictionary containing the training configuration parameters.
- **callbacks** (*list*) – A list of callback objects to be used during training.

Returns
An initialized trainer object based on the specified configuration.

Return type
Any

dicee.trainer.dice_trainer.**get_callbacks**(*args: Dict[str, Any]*) → List[Any]

Get a list of callback objects based on the specified training configuration.

This function constructs and returns a list of callback objects to be used during training.

Parameters
args (*dict*) – A dictionary containing the training configuration parameters.

Returns
A list of callback objects.

Return type
list

class dicee.trainer.dice_trainer.**DICE_Trainer**(*args*, *is_continual_training: bool*, *storage_path: str*, *evaluator: object | None = None*)

Implements a training framework for knowledge graph embedding models using [PyTorch Lightning](https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html), supporting [multi-GPU](https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html) and CPU training. This trainer can handle continual training scenarios and supports different forms of labeling and evaluation methods.

Parameters

- **args** (Namespace) – Command line arguments or configurations specifying training parameters and model settings.
- **is_continual_training** (*bool*) – Flag indicating whether the training session is part of a continual learning process.
- **storage_path** (*str*) – Path to the directory where training checkpoints and models are stored.
- **evaluator** (*object, optional*) – An evaluation object responsible for model evaluation. This can be any object that implements an *eval* method accepting model predictions and returning evaluation metrics.

report

A dictionary to store training reports and metrics.

Type
dict

trainer

The PyTorch Lightning Trainer instance used for model training.

Type
lightening.Trainer or None

form_of_labelling

The form of labeling used during training, which can be "EntityPrediction", "RelationPrediction", or "Pyke".

Type
str or None

continual_start()

Initializes and starts the training process, including model loading and fitting.

initialize_trainer(*callbacks: List*) → lightening.Trainer

Initializes a PyTorch Lightning Trainer instance with the specified callbacks.

initialize_or_load_model()

Initializes or loads a model for training based on the training configuration.

initialize_dataloader(*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

Initializes a DataLoader for the given dataset.

initialize_dataset(*dataset: KG*, *form_of_labelling*) → torch.utils.data.Dataset

Prepares and initializes a dataset for training.

start(*knowledge_graph: KG*) → Tuple[*BaseKGE*, str]

Starts the training process for a given knowledge graph.

k_fold_cross_validation(*dataset*) → Tuple[*BaseKGE*, str]

Performs K-fold cross-validation on the dataset and returns the trained model and form of labelling.

continual_start()

Initializes and starts the training process, including model loading and fitting. This method is specifically designed for continual training scenarios.

Returns

- **model** (*BaseKGE*) – The trained knowledge graph embedding model instance. *BaseKGE* is a placeholder for the actual model class, which should be a subclass of the base model class used in your framework.
- **form_of_labelling** (*str*) – The form of labeling used during the training. This can indicate the type of prediction task the model is trained for, such as "EntityPrediction", "RelationPrediction", or other custom labeling forms defined in your implementation.

initialize_trainer(*callbacks: List*) → lightning.Trainer

Initializes a PyTorch Lightning Trainer instance.

Parameters
callbacks (*List*) – A list of PyTorch Lightning callbacks to be used during training.

Returns
The initialized PyTorch Lightning Trainer instance.

Return type
pl.Trainer

initialize_or_load_model() → Tuple[*dicee.models.base_model.BaseKGE*, str]

Initializes or loads a knowledge graph embedding model based on the training configuration. This method decides whether to start training from scratch or to continue training from a previously saved model state, depending on the *is_continual_training* attribute.

Returns

- **model** (*BaseKGE*) – The model instance that is either initialized from scratch or loaded from a saved state. *BaseKGE* is a generic placeholder for the actual model class, which is a subclass of the base knowledge graph embedding model class used in your implementation.
- **form_of_labelling** (*str*) – A string indicating the type of prediction task the model is configured for. Possible values include "EntityPrediction" and "RelationPrediction", which signify whether the model is trained to predict missing entities or relations in a knowledge graph. The actual values depend on the specific tasks supported by your implementation.

Notes

The method uses the *is_continual_training* attribute to determine if the model should be loaded from a saved state. If *is_continual_training* is True, the method attempts to load the model and its configuration from the specified *storage_path*. If *is_continual_training* is False or the model cannot be loaded, a new model instance is initialized.

This method also sets the *form_of_labelling* attribute based on the model's configuration, which is used to inform downstream training and evaluation processes about the type of prediction task.

initialize_dataloader(*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

Initializes and returns a PyTorch DataLoader object for the given dataset.

This DataLoader is configured based on the training arguments provided, including batch size, shuffle status, and the number of workers.

Parameters
dataset (*torch.utils.data.Dataset*) – The dataset to be loaded into the DataLoader. This dataset should already be processed and ready for training or evaluation.

Returns
A DataLoader instance ready for training or evaluation, configured with the appropriate batch size, shuffle setting, and number of workers.

Return type
torch.utils.data.DataLoader

initialize_dataset(*dataset: dicee.knowledge_graph.KG*, *form_of_labelling: str*)
→ torch.utils.data.Dataset

Initializes and returns a dataset suitable for training or evaluation, based on the knowledge graph data and the specified form of labelling.

Parameters

- **dataset** (KG) – The knowledge graph data used to construct the dataset. This should include training, validation, and test sets along with any other necessary information like entity and relation mappings.
- **form_of_labelling** (*str*) – The form of labelling to be used for the dataset, indicating the prediction task (e.g., "EntityPrediction", "RelationPrediction").

Returns
A processed dataset ready for use with a PyTorch DataLoader, tailored to the specified form of labelling and containing all necessary data for training or evaluation.

Return type
torch.utils.data.Dataset

start (*knowledge_graph: dicee.knowledge_graph.KG*) → Tuple[*dicee.models.base_model.BaseKGE*, str]

Starts the training process for the selected model using the provided knowledge graph data. The method selects and trains the model based on the configuration specified in the arguments.

Parameters
knowledge_graph (KG) – The knowledge graph data containing entities, relations, and triples, which will be used for training the model.

Returns
A tuple containing the trained model instance and the form of labelling used during training. The form of labelling indicates the type of prediction task.

Return type
Tuple[*BaseKGE*, str]

k_fold_cross_validation (*dataset: dicee.knowledge_graph.KG*)
→ Tuple[*dicee.models.base_model.BaseKGE*, str]

Conducts K-fold cross-validation on the provided dataset to assess the performance of the model specified in the training arguments. The process involves partitioning the dataset into K distinct subsets, iteratively using one subset for testing and the remainder for training. The model's performance is evaluated on each test split to compute the Mean Reciprocal Rank (MRR) scores.

Steps: 1. The dataset is divided into K train and test splits. 2. For each split: 2.1. A trainer and model are initialized based on the provided configuration. 2.2. The model is trained using the training portion of the split. 2.3. The MRR score of the trained model is computed using the test portion of the split. 3. The process aggregates the MRR scores across all splits to report the mean and standard deviation of the MRR, providing a comprehensive evaluation of the model's performance.

Parameters
dataset (KG) – The dataset to be used for K-fold cross-validation. This dataset should include the triples (head entity, relation, tail entity) for the entire knowledge graph.

Returns
A tuple containing: - The trained model instance from the last fold of the cross-validation. - The form of labelling used during training, indicating the prediction task (e.g., "EntityPrediction", "RelationPrediction").

Return type
Tuple[*BaseKGE*, str]

Notes

The function assumes the presence of a predefined number of folds (K) specified in the training arguments. It utilizes PyTorch Lightning for model training and evaluation, leveraging GPU acceleration if available. The final output includes the model trained on the last fold and a summary of the cross-validation performance metrics.

dicee.trainer.torch_trainer

Module Contents

Classes

TorchTrainer	A trainer class for PyTorch models that supports training on a single GPU or multiple CPUs.

class dicee.trainer.torch_trainer.**TorchTrainer**(*args*, *callbacks*)

Bases: *dicee.abstracts.AbstractTrainer*

A trainer class for PyTorch models that supports training on a single GPU or multiple CPUs.

Parameters

- **args** (*dict*) – Configuration arguments for training, including model hyperparameters and training options.
- **callbacks** (*List[Callable]*) – List of callback functions to be called at various points of the training process.

loss_function

The loss function used for training.

Type
Callable

optimizer

The optimizer used for training.

Type
torch.optim.Optimizer

model

The PyTorch model being trained.

Type
torch.nn.Module

train_dataloaders

torch.utils.data.DataLoader providing access to the training data.

Type
torch.utils.data.DataLoader

training_step

The training step function defining the forward pass and loss computation.

Type
Callable

device

The device (CPU or GPU) on which training is performed.

Type
torch.device

_run_batch(*i: int*, *x_batch: torch.Tensor*, *y_batch: torch.Tensor*) → float:

Executes a training step for a single batch and returns the loss value.

_run_epoch(*epoch: int*) → float:

Executes training for one epoch and returns the average loss.

fit(**args*, *train_dataloaders: torch.utils.data.DataLoader*, ***kwargs*) → None:

Starts the training process for the given model and data.

forward_backward_update(*x_batch: torch.Tensor*, *y_batch: torch.Tensor*) → float:

Performs the forward pass, computes the loss, and updates model weights.

extract_input_outputs_set_device(*batch: list*) → Tuple[torch.Tensor, torch.Tensor]:

Prepares and moves batch data to the appropriate device.

fit(**args*, *train_dataloaders: torch.utils.data.DataLoader*, ***kwargs*) → None

Starts the training process for the given model and training data.

Parameters

- **model** (*torch.nn.Module*) – The model to be trained.
- **train_dataloaders** (*torch.utils.data.DataLoader*) – A DataLoader instance providing access to the training data.

forward_backward_update(*x_batch: torch.Tensor*, *y_batch: torch.Tensor*) → float

Performs the forward pass, computes the loss, performs the backward pass to compute gradients, and updates the model weights.

Parameters

- **x_batch** (*torch.Tensor*) – The batch of input features.
- **y_batch** (*torch.Tensor*) – The batch of target outputs.

Returns

The loss value computed for the batch.

Return type

float

extract_input_outputs_set_device(*batch: list*) → Tuple[torch.Tensor, torch.Tensor]

Prepares a batch by extracting inputs and outputs and moving them to the correct device.

Parameters

batch (*list*) – A list containing inputs and outputs for the batch.

Returns

A tuple containing the batch of input features and target outputs, both moved to the appropriate device.

Return type

Tuple[torch.Tensor, torch.Tensor]

dicee.trainer.torch_trainer_ddp

Module Contents

Classes

TorchDDPTrainer	A Trainer class that leverages PyTorch's DistributedDataParallel (DDP) for distributed training across
NodeTrainer	Manages the training process of a PyTorch model on a single node in a distributed training setup using
DDPTrainer	Distributed Data Parallel (DDP) Trainer for PyTorch models. Orchestrates the model training across multiple GPUs

Functions

print_peak_memory(→ None)	Prints the peak memory usage for the specified device during the execution.

dicee.trainer.torch_trainer_ddp.**print_peak_memory**(*prefix: str*, *device: int*) → None

Prints the peak memory usage for the specified device during the execution.

Parameters

- **prefix** (*str*) – A prefix string to include in the print statement for context or identification of the memory usage check point.
- **device** (*int*) – The device index for which to check the peak memory usage. This is typically used for CUDA devices. For example, *device=0* refers to the first CUDA device.

Return type
None

Notes

This function is specifically useful for monitoring the peak memory usage of GPU devices in CUDA context. The memory usage is reported in megabytes (MB). This can help in debugging memory issues or for optimizing memory usage in deep learning models. It requires PyTorch's CUDA utilities to be available and will print the peak allocated memory on the specified CUDA device. If the device is not a CUDA device or if PyTorch is not compiled with CUDA support, this function will not display memory usage.

class dicee.trainer.torch_trainer_ddp.**TorchDDPTrainer**(*args*, *callbacks: List[lightning.Callback]*)

Bases: *dicee.abstracts.AbstractTrainer*

A Trainer class that leverages PyTorch's DistributedDataParallel (DDP) for distributed training across multiple GPUs. This trainer is designed for training models in a distributed fashion using multiple GPUs either on a single machine or across multiple nodes.

Parameters

- **args** (*argparse.Namespace*) – The command-line arguments namespace, containing training hyperparameters and configurations.
- **callbacks** (*List[lightening.Callback]*) – A list of PyTorch Lightning Callbacks to be called during the training process.

train_set_idx

An array of indexed triples for training the model.

Type
np.ndarray

entity_idxs

A dictionary mapping entity names to their corresponding indexes.

Type
Dict[str, int]

relation_idxs

A dictionary mapping relation names to their corresponding indexes.

Type
Dict[str, int]

form

The form of training to be used. This parameter specifies how the training data is presented to the model, e.g., 'EntityPrediction', 'RelationPrediction'.

Type
str

store

The path to where the trained model and other artifacts are stored.

Type
str

label_smoothing_rate

The rate of label smoothing to apply to the loss function. Using label smoothing helps in regularizing the model and preventing overfitting by softening the hard targets.

Type
float

fit(self, *args, **kwargs):

Trains the model using distributed data parallelism. This method initializes the distributed process group, creates a distributed data loader, and starts the training process using a NodeTrainer instance. It handles the setup and teardown of the distributed training environment.

Notes

- This trainer requires the PyTorch library and is designed to work with GPUs.
- Proper setup of the distributed environment variables (e.g., WORLD_SIZE, RANK, LOCAL_RANK) is necessary before using this trainer.
- The 'nccl' backend is used for GPU-based distributed training.
- It's important to ensure that the same number of batches is available across all participating processes to avoid hanging issues.

fit (**args*, ***kwargs*)

Trains the model using Distributed Data Parallel (DDP). This method initializes the distributed environment, creates a distributed sampler for the DataLoader, and starts the training process.

Parameters

- ***args** (*Model*) – The model to be trained. Passed as a positional argument.
- ****kwargs** (*dict*) – Additional keyword arguments, including: - train_dataloaders: DataLoader

  The DataLoader for the training dataset. Must contain a 'dataset' attribute.

Raises

AssertionError – If the number of arguments is not equal to 1 (i.e., the model is not provided).

Return type

None

class dicee.trainer.torch_trainer_ddp.**NodeTrainer**(*trainer*, *model: torch.nn.Module*, *train_dataset_loader: torch.utils.data.DataLoader*, *optimizer: torch.optim.Optimizer*, *callbacks*, *num_epochs: int*)

Manages the training process of a PyTorch model on a single node in a distributed training setup using DistributedDataParallel (DDP). This class orchestrates the training process across multiple GPUs on the node, handling batch processing, loss computation, and optimizer steps.

Parameters

- **trainer** (AbstractTrainer) – The higher-level trainer instance managing the overall training process.
- **model** (*torch.nn.Module*) – The PyTorch model to be trained.
- **train_dataset_loader** (*DataLoader*) – The DataLoader providing access to the training data, properly batched and shuffled.
- **optimizer** (*torch.optim.Optimizer*) – The optimizer used for updating model parameters.
- **callbacks** (*list*) – A list of callbacks to be executed during training, such as model checkpointing.
- **num_epochs** (*int*) – The total number of epochs to train the model.

local_rank

The rank of the GPU on the current node, used for GPU-specific operations.

Type

int

global_rank

The global rank of the process in the distributed training setup.

Type

int

loss_func

The loss function used to compute the difference between the model predictions and targets.

Type

callable

loss_history

A list to record the history of loss values over epochs.

Type
list

_run_batch(self, source, targets):

Processes a single batch of data, performing a forward pass, loss computation, and an optimizer step.

extract_input_outputs(self, z):

Extracts and sends input data and targets to the appropriate device.

_run_epoch(self, epoch):

Performs a single pass over the training dataset, returning the average loss for the epoch.

train(self):

Executes the training process, iterating over epochs and managing DDP-specific configurations.

extract_input_outputs(*z: list*) → tuple

Processes the batch data, ensuring it is on the correct device.

Parameters
z (*list*) – The batch data, which can vary in structure depending on the training setup.

Returns
The processed input and output data, ready for model training.

Return type
tuple

train() → None

The main training loop. Iterates over all epochs, processing each batch of data.

Return type
None

class dicee.trainer.torch_trainer_ddp.**DDPTrainer**(*model: torch.nn.Module*, *train_dataset_loader: torch.utils.data.DataLoader*, *optimizer: torch.optim.Optimizer*, *gpu_id: int*, *callbacks: List[Callable]*, *num_epochs: int*)

Distributed Data Parallel (DDP) Trainer for PyTorch models. Orchestrates the model training across multiple GPUs by wrapping the model with PyTorch's DDP. It manages the training loop, loss computation, and optimization steps.

Parameters

- **model** (*torch.nn.Module*) – The model to be trained in a distributed manner.
- **train_dataset_loader** (*DataLoader*) – DataLoader providing access to the training data, properly batched and shuffled.
- **optimizer** (*torch.optim.Optimizer*) – The optimizer to be used for updating the model's parameters.
- **gpu_id** (*int*) – The GPU identifier where the model is to be placed.
- **callbacks** (*List[Callable]*) – A list of callback functions to be called during training.
- **num_epochs** (*int*) – The number of epochs for which the model will be trained.

loss_history

Records the history of loss values over training epochs.

Type
list

_run_batch(*source: torch.Tensor*, *targets: torch.Tensor*) → float:

Executes a forward pass, computes the loss, performs a backward pass, and updates the model parameters for a single batch of data.

extract_input_outputs(*z: List[torch.Tensor]*) → Tuple[torch.Tensor, torch.Tensor]:

Processes the batch data, ensuring it is on the correct device.

_run_epoch(*epoch: int*) → float:

Completes one full pass over the entire dataset and computes the average loss for the epoch.

train() → None:

Starts the training process, iterating through epochs and managing the distributed training operations.

extract_input_outputs(*z: List[torch.Tensor]*) → Tuple[torch.Tensor, torch.Tensor]

Extracts and moves input and target tensors to the correct device.

Parameters
z (*List[torch.Tensor]*) – A batch of data from the DataLoader.

Returns
Inputs and targets, moved to the correct device.

Return type
Tuple[torch.Tensor, torch.Tensor]

train() → None

Trains the model across specified epochs and GPUs using DDP.

Return type
None

Package Contents

Classes

DICE_Trainer	Implements a training framework for knowledge graph embedding models using [PyTorch Lightning](https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html),

class dicee.trainer.**DICE_Trainer**(*args*, *is_continual_training: bool*, *storage_path: str*, *evaluator: object | None = None*)

Implements a training framework for knowledge graph embedding models using [PyTorch Lightning](https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html), supporting [multi-GPU](https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html) and CPU training. This trainer can handle continual training scenarios and supports different forms of labeling and evaluation methods.

Parameters

- **args** (Namespace) – Command line arguments or configurations specifying training parameters and model settings.

- **is_continual_training** (*bool*) – Flag indicating whether the training session is part of a continual learning process.
- **storage_path** (*str*) – Path to the directory where training checkpoints and models are stored.
- **evaluator** (*object, optional*) – An evaluation object responsible for model evaluation. This can be any object that implements an *eval* method accepting model predictions and returning evaluation metrics.

report

A dictionary to store training reports and metrics.

Type
dict

trainer

The PyTorch Lightning Trainer instance used for model training.

Type
lightening.Trainer or None

form_of_labelling

The form of labeling used during training, which can be "EntityPrediction", "RelationPrediction", or "Pyke".

Type
str or None

continual_start()

Initializes and starts the training process, including model loading and fitting.

initialize_trainer(*callbacks: List*) → lightening.Trainer

Initializes a PyTorch Lightning Trainer instance with the specified callbacks.

initialize_or_load_model()

Initializes or loads a model for training based on the training configuration.

initialize_dataloader(*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

Initializes a DataLoader for the given dataset.

initialize_dataset(*dataset: KG*, *form_of_labelling*) → torch.utils.data.Dataset

Prepares and initializes a dataset for training.

start(*knowledge_graph: KG*) → Tuple[*BaseKGE*, str]

Starts the training process for a given knowledge graph.

k_fold_cross_validation(*dataset*) → Tuple[*BaseKGE*, str]

Performs K-fold cross-validation on the dataset and returns the trained model and form of labelling.

continual_start()

Initializes and starts the training process, including model loading and fitting. This method is specifically designed for continual training scenarios.

Returns

- **model** (*BaseKGE*) – The trained knowledge graph embedding model instance. *BaseKGE* is a placeholder for the actual model class, which should be a subclass of the base model class used in your framework.

- **form_of_labelling** (*str*) – The form of labeling used during the training. This can indicate the type of prediction task the model is trained for, such as "EntityPrediction", "RelationPrediction", or other custom labeling forms defined in your implementation.

initialize_trainer(*callbacks: List*) → lightning.Trainer

Initializes a PyTorch Lightning Trainer instance.

Parameters

callbacks (*List*) – A list of PyTorch Lightning callbacks to be used during training.

Returns

The initialized PyTorch Lightning Trainer instance.

Return type

pl.Trainer

initialize_or_load_model() → Tuple[*dicee.models.base_model.BaseKGE*, str]

Initializes or loads a knowledge graph embedding model based on the training configuration. This method decides whether to start training from scratch or to continue training from a previously saved model state, depending on the *is_continual_training* attribute.

Returns

- **model** (*BaseKGE*) – The model instance that is either initialized from scratch or loaded from a saved state. *BaseKGE* is a generic placeholder for the actual model class, which is a subclass of the base knowledge graph embedding model class used in your implementation.
- **form_of_labelling** (*str*) – A string indicating the type of prediction task the model is configured for. Possible values include "EntityPrediction" and "RelationPrediction", which signify whether the model is trained to predict missing entities or relations in a knowledge graph. The actual values depend on the specific tasks supported by your implementation.

Notes

The method uses the *is_continual_training* attribute to determine if the model should be loaded from a saved state. If *is_continual_training* is True, the method attempts to load the model and its configuration from the specified *storage_path*. If *is_continual_training* is False or the model cannot be loaded, a new model instance is initialized.

This method also sets the *form_of_labelling* attribute based on the model's configuration, which is used to inform downstream training and evaluation processes about the type of prediction task.

initialize_dataloader(*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

Initializes and returns a PyTorch DataLoader object for the given dataset.

This DataLoader is configured based on the training arguments provided, including batch size, shuffle status, and the number of workers.

Parameters

dataset (*torch.utils.data.Dataset*) – The dataset to be loaded into the DataLoader. This dataset should already be processed and ready for training or evaluation.

Returns

A DataLoader instance ready for training or evaluation, configured with the appropriate batch size, shuffle setting, and number of workers.

Return type

torch.utils.data.DataLoader

initialize_dataset(*dataset: dicee.knowledge_graph.KG*, *form_of_labelling: str*) → torch.utils.data.Dataset

Initializes and returns a dataset suitable for training or evaluation, based on the knowledge graph data and the specified form of labelling.

Parameters

- **dataset** (KG) – The knowledge graph data used to construct the dataset. This should include training, validation, and test sets along with any other necessary information like entity and relation mappings.
- **form_of_labelling** (*str*) – The form of labelling to be used for the dataset, indicating the prediction task (e.g., "EntityPrediction", "RelationPrediction").

Returns

A processed dataset ready for use with a PyTorch DataLoader, tailored to the specified form of labelling and containing all necessary data for training or evaluation.

Return type

torch.utils.data.Dataset

start(*knowledge_graph: dicee.knowledge_graph.KG*) → Tuple[*dicee.models.base_model.BaseKGE*, str]

Starts the training process for the selected model using the provided knowledge graph data. The method selects and trains the model based on the configuration specified in the arguments.

Parameters

knowledge_graph (KG) – The knowledge graph data containing entities, relations, and triples, which will be used for training the model.

Returns

A tuple containing the trained model instance and the form of labelling used during training. The form of labelling indicates the type of prediction task.

Return type

Tuple[*BaseKGE*, str]

k_fold_cross_validation(*dataset: dicee.knowledge_graph.KG*) → Tuple[*dicee.models.base_model.BaseKGE*, str]

Conducts K-fold cross-validation on the provided dataset to assess the performance of the model specified in the training arguments. The process involves partitioning the dataset into K distinct subsets, iteratively using one subset for testing and the remainder for training. The model's performance is evaluated on each test split to compute the Mean Reciprocal Rank (MRR) scores.

Steps: 1. The dataset is divided into K train and test splits. 2. For each split: 2.1. A trainer and model are initialized based on the provided configuration. 2.2. The model is trained using the training portion of the split. 2.3. The MRR score of the trained model is computed using the test portion of the split. 3. The process aggregates the MRR scores across all splits to report the mean and standard deviation of the MRR, providing a comprehensive evaluation of the model's performance.

Parameters

dataset (KG) – The dataset to be used for K-fold cross-validation. This dataset should include the triples (head entity, relation, tail entity) for the entire knowledge graph.

Returns

A tuple containing: - The trained model instance from the last fold of the cross-validation. - The form of labelling used during training, indicating the prediction task (e.g., "EntityPrediction", "RelationPrediction").

Return type

Tuple[*BaseKGE*, str]

Notes

The function assumes the presence of a predefined number of folds (K) specified in the training arguments. It utilizes PyTorch Lightning for model training and evaluation, leveraging GPU acceleration if available. The final output includes the model trained on the last fold and a summary of the cross-validation performance metrics.

13.2 Submodules

dicee.abstracts

Module Contents

Classes

AbstractTrainer	Abstract base class for Trainer classes used in training knowledge graph embedding models.
BaseInteractiveKGE	Base class for interactively utilizing knowledge graph embedding models.
AbstractCallback	Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.
AbstractPPECallback	Abstract base class for implementing Parameter Prediction Ensemble (PPE) callbacks for knowledge graph embedding models.

class dicee.abstracts.**AbstractTrainer**(*args*, *callbacks*)

Abstract base class for Trainer classes used in training knowledge graph embedding models. Defines common functionalities and lifecycle hooks for training processes.

Parameters

- **args** (*Namespace or similar*) – A container for various training configurations and hyperparameters.
- **callbacks** (*list of Callback objects*) – A list of callback instances to be invoked at various stages of the training process.

on_fit_start(**args*, ***kwargs*) → None

Invokes the *on_fit_start* method of each registered callback before the training starts.

Parameters

- ***args** – Variable length argument list.
- ****kwargs** – Arbitrary keyword arguments.

Return type
None

on_fit_end(**args*, ***kwargs*) → None

Invokes the *on_fit_end* method of each registered callback after the training ends.

Parameters

- ***args** – Variable length argument list.
- ****kwargs** – Arbitrary keyword arguments.

Return type
None

on_train_epoch_end(**args*, ***kwargs*) → None

Invokes the *on_train_epoch_end* method of each registered callback after each epoch ends.

Parameters

- ***args** – Variable length argument list.
- ****kwargs** – Arbitrary keyword arguments.

Return type
None

on_train_batch_end(**args*, ***kwargs*) → None

Invokes the *on_train_batch_end* method of each registered callback after each training batch ends.

Parameters

- ***args** – Variable length argument list.
- ****kwargs** – Arbitrary keyword arguments.

Return type
None

static save_checkpoint(*full_path: str*, *model: torch.nn.Module*) → None

Saves the model's state dictionary to a file.

Parameters

- **full_path** (*str*) – The file path where the model checkpoint will be saved.
- **model** (*torch.nn.Module*) – The model instance whose parameters are to be saved.

Return type
None

class dicee.abstracts.**BaseInteractiveKGE**(*path: str = None*, *url: str = None*, *construct_ensemble: bool = False*, *model_name: str = None*, *apply_semantic_constraint: bool = False*)

Base class for interactively utilizing knowledge graph embedding models. Supports operations such as loading pretrained models, querying the model, and adding new embeddings.

Parameters

- **path** (*str, optional*) – Path to the directory where the pretrained model is stored. Either *path* or *url* must be provided.
- **url** (*str, optional*) – URL to download the pretrained model. If provided, *path* is ignored and the model is downloaded to a local path.
- **construct_ensemble** (*bool, default=False*) – Whether to construct an ensemble model from the pretrained models available in the specified directory.
- **model_name** (*str, optional*) – Name of the specific model to load. Required if multiple models are present and *construct_ensemble* is False.
- **apply_semantic_constraint** (*bool, default=False*) – Whether to apply semantic constraints based on domain and range information during inference.

model

The loaded or constructed knowledge graph embedding model.

Type
torch.nn.Module

entity_to_idx

Mapping from entity names to their corresponding indices in the embedding matrix.

Type
dict

relation_to_idx

Mapping from relation names to their corresponding indices in the embedding matrix.

Type
dict

num_entities

The number of unique entities in the knowledge graph.

Type
int

num_relations

The number of unique relations in the knowledge graph.

Type
int

configs

Configuration settings and performance metrics of the pretrained model.

Type
dict

property name: str

Property that returns the model's name.

Returns
The name of the model.

Return type
str

get_eval_report() → dict

Retrieves the evaluation report of the pretrained model.

Returns
A dictionary containing evaluation metrics and their values.

Return type
dict

get_bpe_token_representation(*str_entity_or_relation: List[str] | str*)
→ List[List[int]] | List[int]

Converts a string entity or relation name (or a list of them) to its Byte Pair Encoding (BPE) token representation.

Parameters
str_entity_or_relation (*Union[List[str], str]*) – The entity or relation name(s) to be converted.

Returns
The BPE token representation as a list of integers or a list of lists of integers.

Return type
Union[List[List[int]], List[int]]

get_padded_bpe_triple_representation (*triples: List[List[str]]*) → Tuple[List, List, List]

Converts a list of triples to their padded BPE token representations.

Parameters
triples (*List[List[str]]*) – A list of triples, where each triple is a list of strings [head entity, relation, tail entity].

Returns
Three lists corresponding to the padded BPE token representations of head entities, relations, and tail entities.

Return type
Tuple[List, List, List]

get_domain_of_relation (*rel: str*) → List[str]

Retrieves the domain of a given relation.

Parameters
rel (*str*) – The relation name.

Returns
A list of entity names that constitute the domain of the specified relation.

Return type
List[str]

get_range_of_relation (*rel: str*) → List[str]

Retrieves the range of a given relation.

Parameters
rel (*str*) – The relation name.

Returns
A list of entity names that constitute the range of the specified relation.

Return type
List[str]

set_model_train_mode () → None

Sets the model to training mode. This enables gradient computation and backpropagation.

set_model_eval_mode () → None

Sets the model to evaluation mode. This disables gradient computation, making the model read-only and faster for inference.

sample_entity (*n: int*) → List[str]

Randomly samples a specified number of unique entities from the knowledge graph.

Parameters
n (*int*) – The number of entities to sample.

Returns
A list of sampled entity names.

Return type
List[str]

sample_relation (*n: int*) → List[str]

Randomly samples a specified number of unique relations from the knowledge graph.

Parameters
n (*int*) – The number of relations to sample.

Returns
A list of sampled relation names.

Return type
List[str]

is_seen (*entity: str = None*, *relation: str = None*) → bool

Checks if the specified entity or relation is known to the model.

Parameters

- **entity** (*str, optional*) – The entity name to check.
- **relation** (*str, optional*) – The relation name to check.

Returns
True if the entity or relation is known; False otherwise.

Return type
bool

save () → None

Saves the current state of the model to disk. The filename is timestamped.

Return type
None

get_entity_index (*x: str*) → int

Retrieves the index of the specified entity.

Parameters
x (*str*) – The entity name.

Returns
The index of the entity.

Return type
int

get_relation_index (*x: str*) → int

Retrieves the index of the specified relation.

Parameters
x (*str*) – The relation name.

Returns
The index of the relation.

Return type
int

index_triple (*head_entity: List[str]*, *relation: List[str]*, *tail_entity: List[str]*)
→ Tuple[torch.LongTensor, torch.LongTensor, torch.LongTensor]

Converts a list of triples from string representation to tensor indices.

Parameters

- **head_entity** (*List[str]*) – The list of head entities.

- **relation** (*List[str]*) – The list of relations.
- **tail_entity** (*List[str]*) – The list of tail entities.

Returns
The tensor indices of head entities, relations, and tail entities.

Return type
Tuple[torch.LongTensor, torch.LongTensor, torch.LongTensor]

add_new_entity_embeddings (*entity_name: str = None*, *embeddings: torch.FloatTensor = None*) → None

Adds a new entity and its embeddings to the model.

Parameters
- **entity_name** (*str*) – The name of the new entity.
- **embeddings** (*torch.FloatTensor*) – The embedding vector of the new entity.

Return type
None

get_entity_embeddings (*items: List[str]*) → torch.FloatTensor

Retrieves embeddings for a list of entities.

Parameters
items (*List[str]*) – A list of entity names.

Returns
A tensor containing the embeddings of the specified entities.

Return type
torch.FloatTensor

get_relation_embeddings (*items: List[str]*) → torch.FloatTensor

Retrieves embeddings for a list of relations.

Parameters
items (*List[str]*) – A list of relation names.

Returns
A tensor containing the embeddings of the specified relations.

Return type
torch.FloatTensor

construct_input_and_output (*head_entity: List[str]*, *relation: List[str]*, *tail_entity: List[str]*, *labels*) → Tuple[torch.Tensor, torch.Tensor]

Constructs input and output tensors for a given set of triples and labels.

Parameters
- **head_entity** (*List[str]*) – A list of head entities.
- **relation** (*List[str]*) – A list of relations.
- **tail_entity** (*List[str]*) – A list of tail entities.
- **labels** (*List[int] or torch.Tensor*) – The labels associated with each triple.

Returns
The input tensor consisting of indexed triples and the output tensor of labels.

Return type
Tuple[torch.Tensor, torch.Tensor]

parameters()

Retrieves the parameters of the model.

This method is typically used to access the parameters of the model for optimization or inspection.

Returns
An iterator over the model parameters, which are instances of torch.nn.parameter.Parameter.

Return type
Iterator[torch.nn.parameter.Parameter]

class dicee.abstracts.**AbstractCallback**

Bases: abc.ABC, lightning.pytorch.callbacks.Callback

Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.

This class is designed to be subclassed, with methods overridden to perform actions at various points during the training life cycle.

on_init_start(**args*, ***kwargs*)

Called before the trainer initialization starts.

Parameters
trainer (*pl.Trainer*) – The trainer instance.

on_init_end(**args*, ***kwargs*)

Called after the trainer initialization ends.

Parameters
trainer (*pl.Trainer*) – The trainer instance.

on_fit_start(*trainer*, *model*)

Called at the very beginning of fit.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that is being trained.

on_train_epoch_end(*trainer*, *model*)

Called at the end of the training epoch.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that is being trained.

on_train_batch_end(**args*, ***kwargs*)

Call at the end of each mini-batch during the training.

Parameter

trainer:

model:

rtype
None

on_fit_end(**args*, ***kwargs*)

Called at the end of fit.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that has been trained.

class dicee.abstracts.**AbstractPPECallback**(*num_epochs: int*, *path: str*, *epoch_to_start: int | None = None*, *last_percent_to_consider: float | None = None*)

Bases: *AbstractCallback*

Abstract base class for implementing Parameter Prediction Ensemble (PPE) callbacks for knowledge graph embedding models.

This class provides a structure for creating ensemble models by averaging model parameters over epochs, which can potentially improve model performance and robustness.

Parameters

- **num_epochs** (*int*) – Total number of epochs for training.
- **path** (*str*) – Path to save or load the ensemble model.
- **epoch_to_start** (*Optional[int]*) – The epoch number to start creating the ensemble. If None, a percentage of epochs to consider can be specified instead.
- **last_percent_to_consider** (*Optional[float]*) – The last percentage of epochs to consider for creating the ensemble. If both *epoch_to_start* and *last_percent_to_consider* are None, ensemble starts from epoch 1.

on_fit_start(*trainer*, *model*)

Called at the very beginning of fit.

Parameters

- **trainer** (*Trainer instance*) – The trainer instance.
- **model** (*LightningModule*) – The model that is being trained.

on_fit_end(*trainer*, *model*)

Called at the end of fit. It loads the ensemble parameters if they exist.

Parameters

- **trainer** (*Trainer instance*) – The trainer instance.
- **model** (*LightningModule*) – The model that has been trained.

store_ensemble(*param_ensemble: torch.Tensor*) → None

Saves the updated parameter ensemble model to disk.

Parameters
param_ensemble (*torch.Tensor*) – The ensemble of model parameters to be saved.

dicee.analyse_experiments

This script should be moved to dicee/scripts

Module Contents

Classes

Experiment	A class to store and manage data from experiments.

Functions

get_default_arguments()	Returns the default arguments for the script.
analyse(args)	Analyzes and summarizes the results of experiments stored in subdirectories.

dicee.analyse_experiments.**get_default_arguments**()

Returns the default arguments for the script.

Returns

args – The namespace containing the default argument values.

Return type

argparse.Namespace

class dicee.analyse_experiments.**Experiment**

A class to store and manage data from experiments.

model_name

A list storing the names of the models used in experiments.

Type

list

embedding_dim

Dimensions of embeddings used in experiments.

Type

list

num_params

The number of parameters in the models.

Type

list

num_epochs

Number of epochs training was run for each experiment.

Type

list

batch_size

Batch sizes used in experiments.

Type
list

lr

Learning rates used in experiments.

Type
list

byte_pair_encoding

Indicates whether byte pair encoding was used.

Type
list

aswa

Indicates whether adaptive SWA was used.

Type
list

path_dataset_folder

Paths to dataset folders used in experiments.

Type
list

pq

P and Q parameters used in experiments, for models that require them.

Type
list

train_mrr, train_h1, train_h3, train_h10

Training metrics: Mean Reciprocal Rank, Hits@1, Hits@3, and Hits@10.

Type
list

val_mrr, val_h1, val_h3, val_h10

Validation metrics.

Type
list

test_mrr, test_h1, test_h3, test_h10

Test metrics.

Type
list

runtime

Runtime of each experiment.

Type
list

normalization

Indicates whether normalization was applied.

Type
list

scoring_technique

Scoring techniques used in experiments.

Type
list

callbacks

Callbacks used in experiments.

Type
list

save_experiment(*x: dict*)

Saves the data from a single experiment into the class's attributes.

to_df() → pd.DataFrame

Converts the accumulated experiment data into a pandas DataFrame.

save_experiment(*x: dict*)

Saves the data from a single experiment into the class's attributes.

Parameters
x (*dict*) – A dictionary containing the data from a single experiment.

to_df() → pandas.DataFrame

Converts the accumulated experiment data into a pandas DataFrame.

Returns
A DataFrame containing the accumulated data from all experiments.

Return type
pd.DataFrame

dicee.analyse_experiments.**analyse**(*args*)

Analyzes and summarizes the results of experiments stored in subdirectories.

This function reads configurations and evaluation reports from each experiment, compiles a summary, and saves it to a CSV file. It also prints the summary and its LaTeX format to the console.

Parameters
args (*argparse.Namespace*) – Command line arguments passed to the script. Expected to contain: - args.dir: The directory containing subdirectories of experiments.

Return type
None

dicee.callbacks

Module Contents

Classes

AccumulateEpochLossCallback	Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.
PrintCallback	Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.
KGESaveCallback	Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.
PseudoLabellingCallback	Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.
ASWA	Adaptive stochastic weight averaging
Eval	Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.
KronE	Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.
Perturb	A callback for a three-Level Perturbation

Functions

estimate_q(eps)	estimate rate of convergence q from sequence esp
compute_convergence(seq, i)	

class dicee.callbacks.**AccumulateEpochLossCallback**(*path: str*)

Bases: *dicee.abstracts.AbstractCallback*

Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.

This class is designed to be subclassed, with methods overridden to perform actions at various points during the training life cycle.

on_fit_end(*trainer*, *model*) → None

Store epoch loss

Parameter

trainer:

model:

rtype
None

class dicee.callbacks.**PrintCallback**

Bases: *dicee.abstracts.AbstractCallback*

Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.

This class is designed to be subclassed, with methods overridden to perform actions at various points during the training life cycle.

on_fit_start(*trainer*, *pl_module*)

Called at the very beginning of fit.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that is being trained.

on_fit_end(*trainer*, *pl_module*)

Called at the end of fit.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that has been trained.

on_train_batch_end(**args*, ***kwargs*)

Call at the end of each mini-batch during the training.

Parameter

trainer:

model:

rtype
None

on_train_epoch_end(**args*, ***kwargs*)

Called at the end of the training epoch.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that is being trained.

class dicee.callbacks.**KGESaveCallback**(*every_x_epoch: int*, *max_epochs: int*, *path: str*)

Bases: *dicee.abstracts.AbstractCallback*

Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.

This class is designed to be subclassed, with methods overridden to perform actions at various points during the training life cycle.

on_train_batch_end(**args*, ***kwargs*)

Call at the end of each mini-batch during the training.

Parameter

trainer:

model:

rtype
None

on_fit_start(*trainer*, *pl_module*)

Called at the very beginning of fit.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that is being trained.

on_train_epoch_end(**args*, ***kwargs*)

Called at the end of the training epoch.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that is being trained.

on_fit_end(**args*, ***kwargs*)

Called at the end of fit.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that has been trained.

on_epoch_end(*model*, *trainer*, ***kwargs*)

class dicee.callbacks.**PseudoLabellingCallback**(*data_module*, *kg*, *batch_size*)

Bases: *dicee.abstracts.AbstractCallback*

Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.

This class is designed to be subclassed, with methods overridden to perform actions at various points during the training life cycle.

create_random_data()

on_epoch_end(*trainer*, *model*)

dicee.callbacks.**estimate_q**(*eps*)

estimate rate of convergence q from sequence esp

dicee.callbacks.**compute_convergence**(*seq*, *i*)

class dicee.callbacks.**ASWA**(*num_epochs*, *path*)

Bases: *dicee.abstracts.AbstractCallback*

Adaptive stochastic weight averaging ASWE keeps track of the validation performance and update s the ensemble model accordingly.

on_fit_end(*trainer*, *model*)

Called at the end of fit.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that has been trained.

static compute_mrr(*trainer*, *model*) → float

get_aswa_state_dict(*model*)

decide(*running_model_state_dict*, *ensemble_state_dict*, *val_running_model*, *mrr_updated_ensemble_model*)

Perform Hard Update, software or rejection

Parameters

- **running_model_state_dict** –
- **ensemble_state_dict** –
- **val_running_model** –
- **mrr_updated_ensemble_model** –

on_train_epoch_end(*trainer*, *model*)

Called at the end of the training epoch.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that is being trained.

class dicee.callbacks.**Eval**(*path*, *epoch_ratio: int = None*)

Bases: *dicee.abstracts.AbstractCallback*

Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.

This class is designed to be subclassed, with methods overridden to perform actions at various points during the training life cycle.

on_fit_start(*trainer*, *model*)

Called at the very beginning of fit.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that is being trained.

on_fit_end(*trainer*, *model*)

Called at the end of fit.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that has been trained.

on_train_epoch_end (*trainer*, *model*)

Called at the end of the training epoch.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that is being trained.

on_train_batch_end (**args*, ***kwargs*)

Call at the end of each mini-batch during the training.

Parameter

trainer:

model:

rtype
None

class dicee.callbacks.**KronE**

Bases: *dicee.abstracts.AbstractCallback*

Abstract base class for implementing custom callbacks for knowledge graph embedding models during training with PyTorch Lightning.

This class is designed to be subclassed, with methods overridden to perform actions at various points during the training life cycle.

static batch_kronecker_product (*a*, *b*)

Kronecker product of matrices a and b with leading batch dimensions. Batch dimensions are broadcast. The number of them mush :type a: torch.Tensor :type b: torch.Tensor :rtype: torch.Tensor

get_kronecker_triple_representation (*indexed_triple: torch.LongTensor*)

Get kronecker embeddings

on_fit_start (*trainer*, *model*)

Called at the very beginning of fit.

Parameters

- **trainer** (*pl.Trainer*) – The trainer instance.
- **pl_module** (*pl.LightningModule*) – The model that is being trained.

class dicee.callbacks.**Perturb** (*level: str = 'input'*, *ratio: float = 0.0*, *method: str = None*, *scaler: float = None*, *frequency=None*)

Bases: *dicee.abstracts.AbstractCallback*

A callback for a three-Level Perturbation

Input Perturbation: During training an input x is perturbed by randomly replacing its element. In the context of knowledge graph embedding models, x can denote a triple, a tuple of an entity and a relation, or a tuple of two entities. A perturbation means that a component of x is randomly replaced by an entity or a relation.

Parameter Perturbation:

Output Perturbation:

on_train_batch_start(*trainer*, *model*, *batch*, *batch_idx*)

Called when the train batch begins.

dicee.config

Module Contents

Classes

Namespace	Simple object for storing attributes.

class dicee.config.**Namespace**(***kwargs*)

Bases: argparse.Namespace

Simple object for storing attributes.

Implements equality by attribute names and values, and provides a simple string representation.

dataset_dir: str

The path of a folder containing train.txt, and/or valid.txt and/or test.txt

save_embeddings_as_csv: bool = False

Embeddings of entities and relations are stored into CSV files to facilitate easy usage.

storage_path: str = 'Experiments'

A directory named with time of execution under –storage_path that contains related data about embeddings.

path_to_store_single_run: str

A single directory created that contains related data about embeddings.

path_single_kg

Path of a file corresponding to the input knowledge graph

sparql_endpoint

An endpoint of a triple store.

model: str = 'Keci'

KGE model

optim: str = 'Adam'

Optimizer

embedding_dim: int = 64

Size of continuous vector representation of an entity/relation

num_epochs: int = 150

Number of pass over the training data

batch_size: int = 1024

Mini-batch size if it is None, an automatic batch finder technique applied

lr: float = 0.1

Learning rate

add_noise_rate: float

The ratio of added random triples into training dataset

gpus

Number GPUs to be used during training

callbacks

10}}

Type

Callbacks, e.g., {“PPE”

Type

{ “last_percent_to_consider”

backend: str = 'pandas'

Backend to read, process, and index input knowledge graph. pandas, polars and rdflib available

trainer: str = 'torchCPUTrainer'

Trainer for knowledge graph embedding model

scoring_technique: str = 'KvsAll'

Scoring technique for knowledge graph embedding models

neg_ratio: int = 0

Negative ratio for a true triple in NegSample training_technique

weight_decay: float = 0.0

Weight decay for all trainable params

normalization: str = 'None'

LayerNorm, BatchNorm1d, or None

init_param: str

xavier_normal or None

gradient_accumulation_steps: int = 0

Not tested e

num_folds_for_cv: int = 0

Number of folds for CV

eval_model: str = 'train_val_test'

[“None”, “train”, “train_val”, “train_val_test”, “test”]

Type

Evaluate trained model choices

save_model_at_every_epoch: int

Not tested

num_core: int = 0

Number of CPUs to be used in the mini-batch loading process

random_seed: int = 0

Random Seed

sample_triples_ratio: float

Read some triples that are uniformly at random sampled. Ratio being between 0 and 1

read_only_few: int

Read only first few triples

pykeen_model_kwargs

Additional keyword arguments for pykeen models

kernel_size: int = 3

Size of a square kernel in a convolution operation

num_of_output_channels: int = 32

Number of slices in the generated feature map by convolution.

p: int = 0

P parameter of Clifford Embeddings

q: int = 1

Q parameter of Clifford Embeddings

input_dropout_rate: float = 0.0

Dropout rate on embeddings of input triples

hidden_dropout_rate: float = 0.0

Dropout rate on hidden representations of input triples

feature_map_dropout_rate: float = 0.0

Dropout rate on a feature map generated by a convolution operation

byte_pair_encoding: bool = False

Byte pair encoding

Type

WIP

adaptive_swa: bool = False

Adaptive stochastic weight averaging

swa: bool = False

Stochastic weight averaging

block_size: int

block size of LLM

continual_learning

Path of a pretrained model size of LLM

__iter__()

dicee.dataset_classes

Module Contents

Classes

BPE_NegativeSamplingDataset	A PyTorch Dataset for handling negative sampling with Byte Pair Encoding (BPE) entities.
MultiLabelDataset	A dataset class for multi-label knowledge graph embedding tasks. This dataset is designed for models where
MultiClassClassificationDataset	A dataset class for multi-class classification tasks, specifically designed for the 1vsALL training strategy
OnevsAllDataset	A dataset for the One-vs-All (1vsAll) training strategy designed for knowledge graph embedding tasks.
KvsAll	Creates a dataset for K-vs-All training strategy, inheriting from torch.utils.data.Dataset.
AllvsAll	A dataset class for the All-versus-All (AllvsAll) training strategy suitable for knowledge graph embedding models.
KvsSampleDataset	Constructs a dataset for KvsSample training strategy, specifically designed for knowledge graph embedding models.
NegSampleDataset	A dataset for training knowledge graph embedding models using negative sampling.
TriplePredictionDataset	A dataset for triple prediction using negative sampling and label smoothing.
CVDataModule	A LightningDataModule for setting up data loaders for cross-validation training of knowledge graph embedding models.

Functions

reload_dataset(→ torch.utils.data.Dataset)	Reloads the dataset from disk and constructs a PyTorch dataset for training.
construct_dataset(→ torch.utils.data.Dataset)	Constructs a dataset based on the specified parameters and returns a PyTorch Dataset object.

dicee.dataset_classes.**reload_dataset**(*path: str*, *form_of_labelling: str*, *scoring_technique: str*, *neg_ratio: float*, *label_smoothing_rate: float*) → torch.utils.data.Dataset

Reloads the dataset from disk and constructs a PyTorch dataset for training.

Parameters

- **path** (*str*) – The path to the directory where the dataset is stored.
- **form_of_labelling** (*str*) – The form of labelling used in the dataset. Determines how data points are represented.
- **scoring_technique** (*str*) – The scoring technique used for evaluating the embeddings.
- **neg_ratio** (*float*) – The ratio of negative samples to positive samples in the dataset.
- **label_smoothing_rate** (*float*) – The rate of label smoothing applied to the dataset.

Returns

A PyTorch dataset object ready for training.

Return type

torch.utils.data.Dataset

dicee.dataset_classes.**construct_dataset**(*, *train_set: numpy.ndarray | list*, *valid_set=None*, *test_set=None*, *ordered_bpe_entities=None*, *train_target_indices=None*, *target_dim: int = None*, *entity_to_idx: dict*, *relation_to_idx: dict*, *form_of_labelling: str*, *scoring_technique: str*, *neg_ratio: int*, *label_smoothing_rate: float*, *byte_pair_encoding=None*, *block_size: int = None*) → torch.utils.data.Dataset

Constructs a dataset based on the specified parameters and returns a PyTorch Dataset object.

Parameters

- **train_set** (*Union[np.ndarray, list]*) – The training set consisting of triples or tokens.
- **valid_set** (*Optional*) – The validation set. Not currently used in dataset construction.
- **test_set** (*Optional*) – The test set. Not currently used in dataset construction.
- **ordered_bpe_entities** (*Optional*) – Ordered byte pair encoding entities for the dataset.
- **train_target_indices** (*Optional*) – Indices of target entities or relations for training.
- **target_dim** (*int, optional*) – The dimension of target entities or relations.
- **entity_to_idx** (*dict*) – A dictionary mapping entity strings to indices.
- **relation_to_idx** (*dict*) – A dictionary mapping relation strings to indices.
- **form_of_labelling** (*str*) – Specifies the form of labelling, such as 'EntityPrediction' or 'RelationPrediction'.
- **scoring_technique** (*str*) – The scoring technique used for generating negative samples or evaluating the model.
- **neg_ratio** (*int*) – The ratio of negative samples to positive samples.
- **label_smoothing_rate** (*float*) – The rate of label smoothing applied to labels.
- **byte_pair_encoding** (*Optional*) – Indicates if byte pair encoding is used.
- **block_size** (*int, optional*) – The block size for transformer-based models.

Returns

A PyTorch dataset object ready for model training.

Return type

torch.utils.data.Dataset

class dicee.dataset_classes.**BPE_NegativeSamplingDataset**(*train_set: torch.LongTensor*, *ordered_shaped_bpe_entities: torch.LongTensor*, *neg_ratio: int*)

Bases: torch.utils.data.Dataset

A PyTorch Dataset for handling negative sampling with Byte Pair Encoding (BPE) entities.

This dataset extends the PyTorch Dataset class to provide functionality for negative sampling in the context of knowledge graph embeddings. It uses byte pair encoding for entities to handle large vocabularies efficiently.

Parameters

- **train_set** (*torch.LongTensor*) – A tensor containing the training set triples with byte pair encoded entities and relations. The shape of the tensor is [N, 3], where N is the number of triples.
- **ordered_shaped_bpe_entities** (*torch.LongTensor*) – A tensor containing the ordered and shaped byte pair encoded entities.

- **neg_ratio** (*int*) – The ratio of negative samples to generate per positive sample.

num_bpe_entities

The number of byte pair encoded entities.

Type
int

num_datapoints

The number of data points (triples) in the training set.

Type
int

__len__() → int

Returns the total number of data points in the dataset.

Returns
The number of data points.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves the BPE-encoded triple and its corresponding label at the specified index.

Parameters
idx (*int*) – Index of the triple to retrieve.

Returns

A tuple containing the following elements: - The BPE-encoded triple as a torch.Tensor of shape (3,). - The label for the triple, where positive examples have a label of 1 and negative examples have a label

of 0, as a torch.Tensor.

Return type
tuple

collate_fn(*batch_shaped_bpe_triples: List[Tuple[torch.Tensor, torch.Tensor]]*) → Tuple[torch.Tensor, torch.Tensor]

Collate function for the BPE_NegativeSamplingDataset. It processes a batch of byte pair encoded triples, performs negative sampling, and returns the batch along with corresponding labels.

This function is designed to be used with a PyTorch DataLoader. It takes a list of byte pair encoded triples as input and generates negative samples according to the specified negative sampling ratio. The function ensures that the negative samples are combined with the original triples to form a single batch, which is suitable for training a knowledge graph embedding model.

Parameters
batch_shaped_bpe_triples (*List[Tuple[torch.Tensor, torch.Tensor]]*) – A list of tuples, where each tuple contains byte pair encoded representations of head entities, relations, and tail entities for a batch of triples.

Returns
A tuple containing two elements: - The first element is a torch.Tensor of shape [N * (1 + neg_ratio), 3] that contains both the original byte pair encoded triples and the generated negative samples. N is the original number of triples in the batch, and neg_ratio is the negative sampling ratio. - The second element is a torch.Tensor of shape [N * (1 + neg_ratio)] that contains the labels for each triple in the batch. Positive samples are labeled as 1, and negative samples are labeled as 0.

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.dataset_classes.**MultiLabelDataset**(*train_set: torch.LongTensor*, *train_indices_target: torch.LongTensor*, *target_dim: int*, *torch_ordered_shaped_bpe_entities: torch.LongTensor*)

Bases: `torch.utils.data.Dataset`

A dataset class for multi-label knowledge graph embedding tasks. This dataset is designed for models where the output involves predicting multiple labels (entities or relations) for a given input (e.g., predicting all possible tail entities given a head entity and a relation).

Parameters

- **train_set** (*torch.LongTensor*) – A tensor containing the training set triples with byte pair encoding, shaped as [num_triples, 3], where each triple is [head, relation, tail].
- **train_indices_target** (*torch.LongTensor*) – A tensor where each row corresponds to the indices of the target labels for each training example. The length of this tensor must match the number of triples in *train_set*.
- **target_dim** (*int*) – The dimensionality of the target space, typically the total number of possible labels (entities or relations).
- **torch_ordered_shaped_bpe_entities** (*torch.LongTensor*) – A tensor containing ordered byte pair encoded entities used for creating embeddings. This tensor is not directly used in generating targets but may be utilized for additional processing or embedding lookup.

num_datapoints

The number of data points (triples) in the dataset.

Type
int

collate_fn

Optional custom collate function to be used with a PyTorch DataLoader. It's set to None by default and can be specified after initializing the dataset if needed.

Type
None or callable

Note: This dataset is particularly suited for KvsAll (K entities vs. All entities) and AllvsAll training strategies in knowledge graph embedding, where a model predicts a set of possible tail entities given a head entity and a relation (or vice versa), and where each training example can have multiple correct labels.

__len__() → int

Returns the total number of data points in the dataset.

Returns
The number of data points.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves the knowledge graph triple and its corresponding multi-label target vector at the specified index.

Parameters
idx (*int*) – Index of the triple to retrieve.

Returns

A tuple containing the following elements: - The triple as a torch.Tensor of shape (3,). - The multi-label target vector as a torch.Tensor of shape (*target_dim*,), where each element

indicates the presence (1) or absence (0) of a label for the given triple.

Return type
tuple

class dicee.dataset_classes.**MultiClassClassificationDataset**(*subword_units: numpy.ndarray*, *block_size: int = 8*)

Bases: `torch.utils.data.Dataset`

A dataset class for multi-class classification tasks, specifically designed for the 1vsALL training strategy in knowledge graph embedding models. This dataset supports tasks where the model predicts a single correct label from all possible labels for a given input.

Parameters

- **subword_units** (*np.ndarray*) – An array of subword unit indices representing the training data. Each row in the array corresponds to a sequence of subword units (e.g., Byte Pair Encoding tokens) that have been converted to their respective numeric indices.
- **block_size** (*int, optional*) – The size of each sequence of subword units to be used as input to the model. This defines the length of the sequences that the model will receive as input, by default 8.

num_of_data_points

The number of sequences or data points available in the dataset, calculated based on the length of the *subword_units* array and the *block_size*.

Type
int

collate_fn

An optional custom collate function to be used with a PyTorch DataLoader. It's set to None by default and can be specified after initializing the dataset if needed.

Type
None or callable

Note: This dataset is tailored for training knowledge graph embedding models on tasks where the output is a single label out of many possible labels (1vsALL strategy). It is especially suited for models trained with subword tokenization methods like Byte Pair Encoding (BPE), where inputs are sequences of subword unit indices.

__len__() → int

Returns the total number of sequences or data points available in the dataset.

Returns
The number of sequences or data points.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves an input sequence and its subsequent target sequence for next token prediction.

Parameters

idx (*int*) – The starting index for the sequence to be retrieved from the dataset.

Returns

A tuple containing two elements: - *x*: The input sequence as a torch.Tensor of shape (*block_size*,). - *y*: The target sequence as a torch.Tensor of shape (*block_size*,), offset by one position

from the input sequence.

Return type

Tuple[torch.Tensor, torch.Tensor]

class dicee.dataset_classes.**OnevsAllDataset**(*train_set_idx: numpy.ndarray*, *entity_idxs*)

Bases: torch.utils.data.Dataset

A dataset for the One-vs-All (1vsAll) training strategy designed for knowledge graph embedding tasks. This dataset structure is particularly suited for models predicting a single correct label (entity) out of all possible entities for a given pair of head entity and relation.

Parameters

- **train_set_idx** (*np.ndarray*) – An array containing indexed triples from the knowledge graph. Each row represents a triple consisting of indices for the head entity, relation, and tail entity, respectively.
- **entity_idxs** (*dict*) – A dictionary mapping entity names to their corresponding unique integer indices. This is used to determine the dimensionality of the target vector in the 1vsAll setting.

train_data

A tensor version of *train_set_idx*, prepared for use with PyTorch models.

Type

torch.LongTensor

target_dim

The dimensionality of the target vector, equivalent to the total number of unique entities in the dataset.

Type

int

collate_fn

An optional custom collate function for use with a PyTorch DataLoader. By default, it is set to None and can be specified after initializing the dataset.

Type

None or callable

Note: This dataset is optimized for training knowledge graph embedding models using the 1vsAll strategy, where the model aims to correctly predict the tail entity from all possible entities given the head entity and relation.

__len__()

Returns the total number of triples in the dataset.

Returns
The total number of triples.

Return type
int

__getitem__(*idx*)

Retrieves the input data and target vector for the triple at index *idx*.

The input data consists of the indices for the head entity and relation, while the target vector is a one-hot encoded vector with a *1* at the position corresponding to the tail entity's index and `0`s elsewhere.

Parameters
idx (*int*) – The index of the triple to retrieve.

Returns
A tuple containing two elements: - The input data as a torch.Tensor of shape (2,), containing the indices of the head entity and relation. - The target vector as a torch.Tensor of shape (*target_dim*,), a one-hot encoded vector for the tail entity.

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.dataset_classes.**KvsAll**(*train_set_idx: numpy.ndarray*, *entity_idxs*, *relation_idxs*, *form*, *store=None*, *label_smoothing_rate: float = 0.0*)

Bases: torch.utils.data.Dataset

Creates a dataset for K-vs-All training strategy, inheriting from torch.utils.data.Dataset. This dataset is tailored for training scenarios where a model predicts all valid tail entities given a head entity and relation pair or vice versa. The labels are multi-hot encoded to represent the presence of multiple valid entities.

Let (D) denote a dataset for KvsAll training and be defined as (D := {(x, y)_i}_{i=1}^{N}), where: (x: (h, r)) is a unique tuple of an entity (h in E) and a relation (r in R) that has been seen in the input graph. (y) denotes a multi-label vector (in [0, 1]^{|E|}) is a binary label. For all (y_i = 1) s.t. ((h, r, E_i) in KG).

Parameters

- **train_set_idx** (*numpy.ndarray*) – A numpy array of shape *(n, 3)* representing *n* triples, where each triple consists of integer indices corresponding to a head entity, a relation, and a tail entity.
- **entity_idxs** (*dict*) – A dictionary mapping entity names (strings) to their unique integer identifiers.
- **relation_idxs** (*dict*) – A dictionary mapping relation names (strings) to their unique integer identifiers.
- **form** (*str*) – A string indicating the prediction form, either 'RelationPrediction' or 'EntityPrediction'.
- **store** (*dict, optional*) – A precomputed dictionary storing the training data points. If provided, it should map tuples of entity and relation indices to lists of entity indices. If *None*, the store will be constructed from *train_set_idx*.
- **label_smoothing_rate** (*float, default=0.0*) – A float representing the rate of label smoothing to be applied. A value of 0 means no label smoothing is applied.

train_data

Tensor containing the input features for the model, typically consisting of pairs of entity and relation indices.

Type
torch.LongTensor

train_target

Tensor containing the target labels for the model, multi-hot encoded to indicate the presence of multiple valid entities.

Type
torch.LongTensor

target_dim

The dimensionality of the target labels, corresponding to the number of unique entities or relations, depending on the *form*.

Type
int

collate_fn

Placeholder for a custom collate function to be used with a PyTorch DataLoader. This is typically set to *None* and can be overridden as needed.

Type
None

Note: The K-vs-All training strategy is used in scenarios where the task is to predict multiple valid entities given a single entity and relation pair. This dataset supports both predicting multiple valid tail entities given a head entity and relation (EntityPrediction) and predicting multiple valid relations given a pair of entities (RelationPrediction).

The label smoothing rate can be adjusted to control the degree of smoothing applied to the target labels, which can help with regularization and model generalization.

__len__() → int

Returns the number of items in the dataset.

Returns
The total number of items.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves the input pair (head entity, relation) and the corresponding multi-label target vector for the item at index *idx*.

The target vector is a binary vector of length *target_dim*, where each element indicates the presence or absence of a tail entity for the given input pair.

Parameters
idx (*int*) – The index of the item to retrieve.

Returns

A tuple containing two elements: - The input pair as a torch.Tensor of shape (2,), containing the indices of the head entity and relation. - The multi-label target vector as a torch.Tensor of shape (*target_dim*,), indicating the presence or

absence of each possible tail entity.

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.dataset_classes.**AllvsAll**(*train_set_idx: numpy.ndarray*, *entity_idxs*, *relation_idxs*, *label_smoothing_rate=0.0*)

Bases: torch.utils.data.Dataset

A dataset class for the All-versus-All (AllvsAll) training strategy suitable for knowledge graph embedding models. This strategy considers all possible pairs of entities and relations, regardless of whether they exist in the knowledge graph, to predict the associated tail entities.

Let D denote a dataset for AllvsAll training and be defined as D:= {(x,y)_i}_i ^N, where x: (h,r) is a possible unique tuple of an entity h in E and a relation r in R. Hence N = |E| x |R| y: denotes a multi-label vector in [0,1]^{|E|} is a binary label.

orall y_i =1 s.t. (h, r, E_i) in KG.

This setup extends beyond observed triples to include all possible combinations of entities and relations, marking non-existent combinations as negatives. It aims to enrich the training data with hard negatives.

train_set_idx
[numpy.ndarray] An array of shape *(n, 3)*, where each row represents a triple (head entity index, relation index, tail entity index).

entity_idxs
[dict] A dictionary mapping entity names to their unique integer indices.

relation_idxs
[dict] A dictionary mapping relation names to their unique integer indices.

label_smoothing_rate
[float, default=0.0] A parameter for label smoothing to mitigate overfitting by softening the hard labels.

train_data
[torch.LongTensor] A tensor containing all possible pairs of entities and relations derived from the input triples.

train_target
[Union[np.ndarray, list]] A target structure (either a Numpy array or a list) indicating the existence of a tail entity for each head entity and relation pair. It supports multi-label classification where a pair can have multiple correct tail entities.

target_dim
[int] The dimension of the target vector, equal to the total number of unique entities.

collate_fn
[None or callable] An optional function to merge a list of samples into a batch for loading. If not provided, the default collate function of PyTorch's DataLoader will be used.

__len__() → int

Returns the number of items in the dataset, including both existing and potential triples.

Returns
The total number of items.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves the input pair (head entity, relation) and the corresponding multi-label target vector for the item at index *idx*. The target vector is a binary vector of length *target_dim*, where each element indicates the presence or absence of a tail entity for the given input pair, including negative samples.

Parameters
idx (*int*) – The index of the item to retrieve.

Returns

A tuple containing two elements: - The input pair as a torch.Tensor of shape (2,), containing the indices of the head entity and relation. - The multi-label target vector as a torch.Tensor of shape (*target_dim*,), indicating the presence or

absence of each possible tail entity, including negative samples.

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.dataset_classes.**KvsSampleDataset**(*train_set: numpy.ndarray*, *num_entities*, *num_relations*, *neg_sample_ratio: int = None*, *label_smoothing_rate: float = 0.0*)

Bases: `torch.utils.data.Dataset`

Constructs a dataset for KvsSample training strategy, specifically designed for knowledge graph embedding models. This dataset formulation is aimed at handling the imbalance between positive and negative examples for each (head, relation) pair by subsampling tail entities. The subsampling ensures a balanced representation of positive and negative examples in each training batch, according to the specified negative sampling ratio.

The dataset is defined as (D:= {(x,y)_i}_{i=1}^{N}), where:

- (x: (h,r)) is a unique head entity (h in E) and a relation (r in R).
- (y in [0,1]^{|E|}) is a binary label vector. For all (y_i = 1) such that ((h, r, E_i) in KG).

At each mini-batch construction, we subsample (y), hence (|new_y| ll |E|). The new (y) contains all 1's if (sum(y) <) neg_sample_ratio, otherwise, it contains a balanced mix of 1's and 0's.

Parameters

- **train_set** (*np.ndarray*) – An array of shape ((n, 3)), where (n) is the number of triples in the dataset. Each row in the array represents a triple ((h, r, t)), consisting of head entity index (h), relation index (r), and tail entity index (t).
- **num_entities** (*int*) – The total number of unique entities in the dataset.
- **num_relations** (*int*) – The total number of unique relations in the dataset.
- **neg_sample_ratio** (*int*) – The ratio of negative samples to positive samples for each (head, relation) pair. If the number of available positive samples is less than this ratio, additional negative samples are generated to meet the ratio.
- **label_smoothing_rate** (*float, default=0.0*) – A parameter for label smoothing, aiming to mitigate overfitting by softening the hard labels. The labels are adjusted towards a uniform distribution, with the smoothing rate determining the degree of softening.

train_data

A tensor containing the (head, relation) pairs derived from the input triples, used to index the training set.

Type
torch.IntTensor

train_target

A list where each element corresponds to the tail entity indices associated with a given (head, relation) pair.

Type
list of numpy.ndarray

collate_fn

A function to merge a list of samples to form a batch. If None, PyTorch's default collate function is used.

Type
None or callable

__len__()

Returns the total number of unique (head, relation) pairs in the dataset.

Returns
The number of unique (head, relation) pairs.

Return type
int

__getitem__(*idx*)

Retrieves the data for the given index, including the (head, relation) pair, selected tail entity indices, and their labels. Positive examples are sampled from the training set, and negative examples are generated by randomly selecting tail entities not associated with the (head, relation) pair.

Parameters
idx (*int*) – The index of the (head, relation) pair in the dataset.

Returns

A tuple containing the following elements: - x: The (head, relation) pair as a torch.Tensor. - y_idx: The indices of selected tail entities, both positive and negative, as a torch.IntTensor. - y_vec: The labels for the selected tail entities, with 1s indicating positive and 0s indicating negative

examples, as a torch.Tensor.

Return type
tuple

class dicee.dataset_classes.**NegSampleDataset**(*train_set: numpy.ndarray*, *num_entities: int*, *num_relations: int*, *neg_sample_ratio: int = 1*)

Bases: torch.utils.data.Dataset

A dataset for training knowledge graph embedding models using negative sampling. For each positive triple from the knowledge graph, a negative triple is generated by corrupting either the head or the tail entity with a randomly selected entity.

Parameters

- **train_set** (*np.ndarray*) – The training set of triples, where each triple consists of indices of the head entity, relation, and tail entity.
- **num_entities** (*int*) – The total number of unique entities in the knowledge graph.
- **num_relations** (*int*) – The total number of unique relations in the knowledge graph.
- **neg_sample_ratio** (*int, default=1*) – The ratio of negative samples to positive samples. Currently, it generates one negative sample per positive sample.

train_set

The training set converted to a PyTorch tensor and expanded to include a batch dimension.

Type
torch.Tensor

length

The total number of triples in the training set.

Type
int

num_entities

A tensor containing the total number of entities.

Type
torch.tensor

num_relations

A tensor containing the total number of relations.

Type
torch.tensor

neg_sample_ratio

A tensor containing the ratio of negative to positive samples.

Type
torch.tensor

__len__() → int

Returns the total number of triples in the dataset.

Returns
The total number of triples.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves a pair consisting of a positive triple and a generated negative triple along with their labels.

Parameters
idx (*int*) – The index of the triple to retrieve.

Returns
A tuple where the first element is a tensor containing a pair of positive and negative triples, and the second element is a tensor containing their respective labels (1 for positive, 0 for negative).

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.dataset_classes.**TriplePredictionDataset**(*train_set: numpy.ndarray*, *num_entities: int*, *num_relations: int*, *neg_sample_ratio: int = 1*, *label_smoothing_rate: float = 0.0*)

Bases: torch.utils.data.Dataset

A dataset for triple prediction using negative sampling and label smoothing.

D:= {(x)_i}_i ^N, where - x:(h,r, t) in KG is a unique h in E and a relation r in R and - collact_fn => Generates negative triples

collect_fn:

orall (h,r,t) in G obtain, create negative triples{(h,r,x),(,r,t),(h,m,t)}

y: labels are represented in torch.float16

This dataset generates negative triples by corrupting either the head or the tail of each positive triple from the training set. The corruption is performed by randomly replacing the head or the tail with another

entity from the entity set. The dataset supports label smoothing to soften the target labels, which can help improve generalization.

train_set
[np.ndarray] The training set consisting of triples in the form of (head, relation, tail) indices.

num_entities
[int] The total number of unique entities in the knowledge graph.

num_relations
[int] The total number of unique relations in the knowledge graph.

neg_sample_ratio
[int, optional] The ratio of negative samples to generate for each positive sample. Default is 1.

label_smoothing_rate
[float, optional] The rate of label smoothing to apply to the target labels. Default is 0.0.

The *collate_fn* should be passed to the DataLoader's *collate_fn* argument to ensure proper batch processing and negative sample generation.

__len__() → int

Returns the total number of triples in the dataset.

Returns
The total number of triples.

Return type
int

__getitem__(*idx: int*) → torch.Tensor

Retrieves a triple for the given index.

Parameters
idx (*int*) – The index of the triple to retrieve.

Returns
The triple at the specified index.

Return type
torch.Tensor

collate_fn(*batch: List[torch.Tensor]*) → Tuple[torch.Tensor, torch.Tensor]

Custom collate function to generate a batch of positive and negative triples along with their labels.

Parameters
batch (*List[torch.Tensor]*) – A list of tensors representing triples.

Returns
A tuple containing a tensor of triples and a tensor of corresponding labels.

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.dataset_classes.**CVDataModule**(*train_set_idx: numpy.ndarray*, *num_entities: int*, *num_relations: int*, *neg_sample_ratio: int*, *batch_size: int*, *num_workers: int*)

Bases: pytorch_lightning.LightningDataModule

A LightningDataModule for setting up data loaders for cross-validation training of knowledge graph embedding models.

Parameters

- **train_set_idx** (*np.ndarray*) – An array of indexed triples for training, where each triple consists of indices of the head entity, relation, and tail entity.
- **num_entities** (*int*) – The total number of unique entities in the knowledge graph.
- **num_relations** (*int*) – The total number of unique relations in the knowledge graph.
- **neg_sample_ratio** (*int*) – The ratio of negative samples to positive samples for each positive triple.
- **batch_size** (*int*) – The number of samples in each batch of data.
- **num_workers** (*int*) – The number of subprocesses to use for data loading. https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

Returns
A PyTorch DataLoader for the training dataset.

Return type
DataLoader

train_dataloader() → torch.utils.data.DataLoader

Creates a DataLoader for the training dataset.

Returns
A DataLoader object that loads the training data.

Return type
DataLoader

setup(**args*, ***kwargs*)

Called at the beginning of fit (train + validate), validate, test, or predict. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

Parameters
stage – either `'fit'`, `'validate'`, `'test'`, or `'predict'`

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

        # don't do this
        self.something = else

    def setup(self, stage):
        data = load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

transfer_batch_to_device(**args*, ***kwargs*)

Override this hook if your `DataLoader` returns tensors wrapped in a custom data structure.

The data types listed below (and any arbitrary nesting of them) are supported out of the box:

- `torch.Tensor` or anything that implements *.to(...)*
- `list`

- `dict`
- `tuple`

For anything else, you need to define how the data is moved to the target device (CPU, GPU, TPU, …).

Note: This hook should only transfer the data and not modify it, nor should it move the data to any other device than the one passed in as argument (unless you know what you are doing). To check the current state of execution of this hook you can use `self.trainer.training/testing/validating/predicting` so that you can add different logic as per your requirement.

Parameters

- **`batch`** – A batch of data that needs to be transferred to a new device.
- **`device`** – The target device as defined in PyTorch.
- **`dataloader_idx`** – The index of the dataloader to which the batch belongs.

Returns

A reference to the data on the new device.

Example:

```
def transfer_batch_to_device(self, batch, device, dataloader_idx):
    if isinstance(batch, CustomBatch):
        # move all tensors in your custom data structure to the device
        batch.samples = batch.samples.to(device)
        batch.targets = batch.targets.to(device)
    elif dataloader_idx == 0:
        # skip device transfer for the first dataloader or anything you wish
        pass
    else:
        batch = super().transfer_batch_to_device(batch, device, dataloader_
↪idx)
    return batch
```

Raises

`MisconfigurationException` – If using IPUs, `Trainer(accelerator='ipu')`.

See also:

- `move_data_to_device()`
- `apply_to_collection()`

`prepare_data`(**args*, ***kwargs*)

Use this to download and prepare data. Downloading and saving data with multiple processes (distributed settings) will result in corrupted data. Lightning ensures this method is called only within a single process, so you can safely add your downloading logic within.

> **Warning:** DO NOT set state to the model (use `setup` instead) since this is NOT called on every device

Example:

```
def prepare_data(self):
    # good
    download_data()
    tokenize()
    etc()

    # bad
    self.split = data_split
    self.some_state = some_other_state()
```

In a distributed environment, `prepare_data` can be called in two ways (using prepare_data_per_node)

1. Once per node. This is the default and is only called on LOCAL_RANK=0.
2. Once in total. Only called on GLOBAL_RANK=0.

Example:

```
# DEFAULT
# called once per node on LOCAL_RANK=0 of that node
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = True


# call on GLOBAL_RANK=0 (great for shared file systems)
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = False
```

This is called before requesting the dataloaders:

```
model.prepare_data()
initialize_distributed()
model.setup(stage)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
model.predict_dataloader()
```

dicee.eval_static_funcs

Module Contents

Functions

evaluate_link_prediction_performance(→ Dict)	**param model**
evaluate_link_prediction_performance_w	
evaluate_link_prediction_performance_w	
evaluate_link_prediction_performance_w ...)	**param model**
evaluate_lp_bpe_k_vs_all(model, triples[, er_vocab, ...])	

dicee.eval_static_funcs.**evaluate_link_prediction_performance**(*model: dicee.knowledge_graph_embeddings.KGE*, *triples*, *er_vocab: Dict[Tuple, List]*, *re_vocab: Dict[Tuple, List]*) → Dict

Parameters

- **model** –
- **triples** –
- **er_vocab** –
- **re_vocab** –

dicee.eval_static_funcs.**evaluate_link_prediction_performance_with_reciprocals**(*model: dicee.knowledge_graph_embeddings.KGE*, *triples*, *er_vocab: Dict[Tuple, List]*)

dicee.eval_static_funcs.**evaluate_link_prediction_performance_with_bpe_reciprocals**(*model: dicee.knowledge_graph_embeddings.KGE*, *within_entities: List[str]*, *triples: List[List[str]]*, *er_vocab: Dict[Tuple, List]*)

dicee.eval_static_funcs.**evaluate_link_prediction_performance_with_bpe**(*model: dicee.knowledge_graph_embeddings.KGE*, *within_entities: List[str]*, *triples: List[Tuple[str]]*, *er_vocab: Dict[Tuple, List]*, *re_vocab: Dict[Tuple, List]*)

Parameters

- **model** –
- **triples** –
- **within_entities** –
- **er_vocab** –
- **re_vocab** –

dicee.eval_static_funcs.**evaluate_lp_bpe_k_vs_all**(*model*, *triples: List[List[str]]*, *er_vocab=None*, *batch_size=None*, *func_triple_to_bpe_representation: Callable = None*, *str_to_bpe_entity_to_idx=None*)

dicee.evaluator

Module Contents

Classes

Evaluator	Evaluator class to evaluate KGE models in various downstream tasks

class dicee.evaluator.**Evaluator**(*args*, *is_continual_training=None*)

Evaluator class to evaluate KGE models in various downstream tasks

Arguments

vocab_preparation(*dataset*) → None

A function to wait future objects for the attributes of executor

Return type
None

eval(*dataset: dicee.knowledge_graph.KG*, *trained_model*, *form_of_labelling*, *during_training=False*) → None

eval_rank_of_head_and_tail_entity(*, *train_set*, *valid_set=None*, *test_set=None*, *trained_model*)

eval_rank_of_head_and_tail_byte_pair_encoded_entity(*, *train_set=None*, *valid_set=None*, *test_set=None*, *ordered_bpe_entities*, *trained_model*)

eval_with_byte(*, *raw_train_set*, *raw_valid_set=None*, *raw_test_set=None*, *trained_model*, *form_of_labelling*) → None

Evaluate model after reciprocal triples are added

eval_with_bpe_vs_all(*, *raw_train_set*, *raw_valid_set=None*, *raw_test_set=None*, *trained_model*, *form_of_labelling*) → None

Evaluate model after reciprocal triples are added

eval_with_vs_all(*, *train_set*, *valid_set=None*, *test_set=None*, *trained_model*, *form_of_labelling*) → None

Evaluate model after reciprocal triples are added

evaluate_lp_k_vs_all(*model*, *triple_idx*, *info=None*, *form_of_labelling=None*)

Filtered link prediction evaluation. :param model: :param triple_idx: test triples :param info: :param form_of_labelling: :return:

evaluate_lp_with_byte(*model*, *triples: List[List[str]]*, *info=None*)

evaluate_lp_bpe_k_vs_all(*model*, *triples: List[List[str]]*, *info=None*, *form_of_labelling=None*)

Parameters

- **model** –
- **triples** (*List of lists*) –
- **info** –
- **form_of_labelling** –

evaluate_lp(*model*, *triple_idx*, *info: str*)

dummy_eval(*trained_model*, *form_of_labelling: str*)

eval_with_data(*dataset*, *trained_model*, *triple_idx: numpy.ndarray*, *form_of_labelling: str*)

dicee.executer

Module Contents

Classes

Execute	A class for Training, Retraining and Evaluation a model.
ContinuousExecute	A subclass of Execute Class for retraining

class dicee.executer.**Execute**(*args*, *continuous_training=False*)

A class for Training, Retraining and Evaluation a model.

(1) Loading & Preprocessing & Serializing input data.

(2) Training & Validation & Testing

(3) Storing all necessary info

read_or_load_kg()

read_preprocess_index_serialize_data() → None

Read & Preprocess & Index & Serialize Input Data

(1) Read or load the data from disk into memory.

(2) Store the statistics of the data.

Parameter

rtype
None

load_indexed_data() → None

Load the indexed data from disk into memory

Parameter

rtype
None

save_trained_model() → None
Save a knowledge graph embedding model

(1) Send model to eval mode and cpu.

(2) Store the memory footprint of the model.

(3) Save the model into disk.

(4) Update the stats of KG again ?

Parameter

rtype
None

end(*form_of_labelling: str*) → dict
End training

(1) Store trained model.

(2) Report runtimes.

(3) Eval model if required.

Parameter

rtype
A dict containing information about the training and/or evaluation

write_report() → None
Report training related information in a report.json file

start() → dict
Start training

(1) Loading the Data # (2) Create an evaluator object. # (3) Create a trainer object. # (4) Start the training

Parameter

rtype
A dict containing information about the training and/or evaluation

class dicee.executer.**ContinuousExecute**(*args*)
Bases: *Execute*

A subclass of Execute Class for retraining

(1) Loading & Preprocessing & Serializing input data.

(2) Training & Validation & Testing

(3) Storing all necessary info

During the continual learning we can only modify * **num_epochs** * parameter. Trained model stored in the same folder as the seed model for the training. Trained model is noted with the current time.

continual_start() → dict

Start Continual Training

(1) Initialize training.

(2) Start continual training.

(3) Save trained model.

Parameter

rtype
A dict containing information about the training and/or evaluation

dicee.knowledge_graph

Module Contents

Classes

KG	Knowledge Graph

class dicee.knowledge_graph.**KG**(*dataset_dir: str = None*, *byte_pair_encoding: bool = False*, *padding: bool = False*, *add_noise_rate: float = None*, *sparql_endpoint: str = None*, *path_single_kg: str = None*, *path_for_deserialization: str = None*, *add_reciprical: bool = None*, *eval_model: str = None*, *read_only_few: int = None*, *sample_triples_ratio: float = None*, *path_for_serialization: str = None*, *entity_to_idx=None*, *relation_to_idx=None*, *backend=None*, *training_technique: str = None*)

Knowledge Graph

property entities_str: List

property relations_str: List

func_triple_to_bpe_representation(*triple: List[str]*)

dicee.knowledge_graph_embeddings

Module Contents

Classes

KGE	Knowledge Graph Embedding Class for interactive usage of pre-trained models

class dicee.knowledge_graph_embeddings.**KGE**(*path=None*, *url=None*, *construct_ensemble=False*, *model_name=None*, *apply_semantic_constraint=False*)

Bases: *dicee.abstracts.BaseInteractiveKGE*

Knowledge Graph Embedding Class for interactive usage of pre-trained models

get_transductive_entity_embeddings(*indices: torch.LongTensor | List[str]*, *as_pytorch=False*, *as_numpy=False*, *as_list=True*)
→ torch.FloatTensor | numpy.ndarray | List[float]

create_vector_database(*collection_name: str*, *distance: str*, *location: str = 'localhost'*, *port: int = 6333*)

generate(*h=''*, *r=''*)

__str__()

Return str(self).

eval_lp_performance(*dataset=List[Tuple[str, str, str]]*, *filtered=True*)

predict_missing_head_entity(*relation: List[str] | str*, *tail_entity: List[str] | str*, *within=None*)
→ Tuple

Given a relation and a tail entity, return top k ranked head entity.

argmax_{e in E } f(e,r,t), where r in R, t in E.

Parameter

relation: Union[List[str], str]

String representation of selected relations.

tail_entity: Union[List[str], str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_relations(*head_entity: List[str] | str*, *tail_entity: List[str] | str*, *within=None*)
→ Tuple

Given a head entity and a tail entity, return top k ranked relations.

argmax_{r in R } f(h,r,t), where h, t in E.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_tail_entity(*head_entity: List[str] | str*, *relation: List[str] | str*, *within: List[str] = None*) → torch.FloatTensor

Given a head entity and a relation, return top k ranked entities

argmax_{e in E } f(h,r,e), where h in E and r in R.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

scores

predict(*, *h: List[str] | str = None*, *r: List[str] | str = None*, *t: List[str] | str = None*, *within=None*, *logits=True*) → torch.FloatTensor

Parameters

- **logits** –
- **h** –
- **r** –
- **t** –
- **within** –

predict_topk(*, *h: List[str] = None*, *r: List[str] = None*, *t: List[str] = None*, *topk: int = 10*, *within: List[str] = None*)

Predict missing item in a given triple.

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k item.

Returns: Tuple

Highest K scores and items

triple_score (*h: List[str] | str = None*, *r: List[str] | str = None*, *t: List[str] | str = None*, *logits=False*) → torch.FloatTensor

Predict triple score

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

Returns: Tuple

pytorch tensor of triple score

t_norm (*tens_1: torch.Tensor*, *tens_2: torch.Tensor*, *tnorm: str = 'min'*) → torch.Tensor

tensor_t_norm (*subquery_scores: torch.FloatTensor*, *tnorm: str = 'min'*) → torch.FloatTensor

Compute T-norm over [0,1] ^{n imes d} where n denotes the number of hops and d denotes number of entities

t_conorm (*tens_1: torch.Tensor*, *tens_2: torch.Tensor*, *tconorm: str = 'min'*) → torch.Tensor

negnorm (*tens_1: torch.Tensor*, *lambda_: float*, *neg_norm: str = 'standard'*) → torch.Tensor

return_multi_hop_query_results(*aggregated_query_for_all_entities*, *k: int*, *only_scores*)

single_hop_query_answering(*query: tuple*, *only_scores: bool = True*, *k: int = None*)

answer_multi_hop_query(*query_type: str = None*,
query: Tuple[str | Tuple[str, str], Ellipsis] = None,
queries: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, *tnorm: str = 'prod'*,
neg_norm: str = 'standard', *lambda_: float = 0.0*, *k: int = 10*, *only_scores=False*)
→ List[Tuple[str, torch.Tensor]]

@TODO: Refactoring is needed # @TODO: Score computation for each query type should be done in a static function

Find an answer set for EPFO queries including negation and disjunction

Parameter

query_type: str The type of the query, e.g., "2p".

query: Union[str, Tuple[str, Tuple[str, str]]] The query itself, either a string or a nested tuple.

queries: List of Tuple[Union[str, Tuple[str, str]], …]

tnorm: str The t-norm operator.

neg_norm: str The negation norm.

lambda_: float lambda parameter for sugeno and yager negation norms

k: int The top-k substitutions for intermediate variables.

returns

- *List[Tuple[str, torch.Tensor]]*
- *Entities and corresponding scores sorted in the descening order of scores*

find_missing_triples(*confidence: float*, *entities: List[str] = None*, *relations: List[str] = None*,
topk: int = 10, *at_most: int = sys.maxsize*) → Set

Find missing triples

Iterative over a set of entities E and a set of relation R :

orall e in E and orall r in R f(e,r,x)

Return (e,r,x)

otin G and f(e,r,x) > confidence

confidence: float

A threshold for an output of a sigmoid function given a triple.

topk: int

Highest ranked k item to select triples with f(e,r,x) > confidence .

at_most: int

Stop after finding at_most missing triples

{(e,r,x) | f(e,r,x) > confidence land (e,r,x)

otin G

deploy (*share: bool = False*, *top_k: int = 10*)

train_triples (*h: List[str]*, *r: List[str]*, *t: List[str]*, *labels: List[float]*, *iteration=2*, *optimizer=None*)

train_k_vs_all (*h*, *r*, *iteration=1*, *lr=0.001*)

Train k vs all :param head_entity: :param relation: :param iteration: :param lr: :return:

train (*kg*, *lr=0.1*, *epoch=10*, *batch_size=32*, *neg_sample_ratio=10*, *num_workers=1*) → None

Retrained a pretrain model on an input KG via negative sampling.

dicee.query_generator

Module Contents

Classes

QueryGenerator

class dicee.query_generator.**QueryGenerator** (*train_path*, *val_path: str*, *test_path: str*, *ent2id: Dict = None*, *rel2id: Dict = None*, *seed: int = 1*, *gen_valid: bool = False*, *gen_test: bool = True*)

list2tuple (*list_data*)

tuple2list (*x: List | Tuple*) → List | Tuple

Convert a nested tuple to a nested list.

set_global_seed (*seed: int*)

Set seed

construct_graph (*paths: List[str]*) → Tuple[Dict, Dict]

Construct graph from triples Returns dicts with incoming and outgoing edges

fill_query (*query_structure: List[str | List]*, *ent_in: Dict*, *ent_out: Dict*, *answer: int*) → bool

Private method for fill_query logic.

achieve_answer (*query: List[str | List]*, *ent_in: Dict*, *ent_out: Dict*) → set

Private method for achieve_answer logic. @TODO: Document the code

write_links (*ent_out*, *small_ent_out*)

ground_queries (*query_structure: List[str | List]*, *ent_in: Dict*, *ent_out: Dict*, *small_ent_in: Dict*, *small_ent_out: Dict*, *gen_num: int*, *query_name: str*)

Generating queries and achieving answers

unmap (*query_type*, *queries*, *tp_answers*, *fp_answers*, *fn_answers*)

unmap_query (*query_structure*, *query*, *id2ent*, *id2rel*)

generate_queries (*query_struct: List*, *gen_num: int*, *query_type: str*)

Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting queries and answers in return @ TODO: create a class for each single query struct

save_queries(*query_type: str*, *gen_num: int*, *save_path: str*)

abstract load_queries(*path*)

get_queries(*query_type: str*, *gen_num: int*)

static save_queries_and_answers(*path: str*,
data: List[Tuple[str, Tuple[collections.defaultdict]]]) → None

Save Queries into Disk

static load_queries_and_answers(*path: str*)
→ List[Tuple[str, Tuple[collections.defaultdict]]]

Load Queries from Disk to Memory

dicee.sanity_checkers

Module Contents

Functions

is_sparql_endpoint_alive([sparql_endpoint])	
validate_knowledge_graph(args)	Validating the source of knowledge graph
sanity_checking_with_arguments(args)	

dicee.sanity_checkers.**is_sparql_endpoint_alive**(*sparql_endpoint: str = None*)

dicee.sanity_checkers.**validate_knowledge_graph**(*args*)

Validating the source of knowledge graph

dicee.sanity_checkers.**sanity_checking_with_arguments**(*args*)

dicee.static_funcs

Module Contents

Functions

create_recipriocal_triples(x)	Add inverse triples into dask dataframe
get_er_vocab(data[, file_path])	
get_re_vocab(data[, file_path])	
get_ee_vocab(data[, file_path])	
timeit(func)	

continues on next page

Table 2 – continued from previous page

save_pickle(*[, data, file_path])	
load_pickle([file_path])	
select_model(args[, is_continual_training, storage_path])	
load_model(→ Tuple[object, Tuple[dict, dict]])	Load weights and initialize pytorch module from namespace arguments
load_model_ensemble(...)	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
save_numpy_ndarray(*, data, file_path)	
numpy_data_type_changer(→ numpy.ndarray)	Detect most efficient data type for a given triples
save_checkpoint_model(→ None)	Store Pytorch model into disk
store(→ None)	Store trained_model model and save embeddings into csv file.
add_noisy_triples(→ pandas.DataFrame)	Add randomly constructed triples
read_or_load_kg(args, cls)	
intialize_model(→ Tuple[object, str])	
load_json(→ dict)	
save_embeddings(→ None)	Save it as CSV if memory allows.
random_prediction(pre_trained_kge)	
deploy_triple_prediction(pre_trained_kge, str_subject, ...)	
deploy_tail_entity_prediction(pre_trained_ ...)	
deploy_head_entity_prediction(pre_trained_ ...)	
deploy_relation_prediction(pre_trained_kge, ...)	
vocab_to_parquet(vocab_to_idx, name, ...)	
create_experiment_folder([folder_name])	
continual_training_setup_executor(→ None)	storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data
exponential_function(→ torch.FloatTensor)	
load_numpy(→ numpy.ndarray)	
evaluate(entity_to_idx, scores, easy_answers, hard_answers)	# @TODO: CD: Renamed this function
download_file(url[, destination_folder])	
download_files_from_url(→ None)	**param base_url**

continues on next page

Table 2 – continued from previous page

download_pretrained_model(→ str)

dicee.static_funcs.**create_recipriocal_triples**(*x*)

Add inverse triples into dask dataframe :param x: :return:

dicee.static_funcs.**get_er_vocab**(*data*, *file_path: str = None*)

dicee.static_funcs.**get_re_vocab**(*data*, *file_path: str = None*)

dicee.static_funcs.**get_ee_vocab**(*data*, *file_path: str = None*)

dicee.static_funcs.**timeit**(*func*)

dicee.static_funcs.**save_pickle**(*, *data: object = None*, *file_path=str*)

dicee.static_funcs.**load_pickle**(*file_path=str*)

dicee.static_funcs.**select_model**(*args: dict*, *is_continual_training: bool = None*, *storage_path: str = None*)

dicee.static_funcs.**load_model**(*path_of_experiment_folder: str*, *model_name='model.pt'*, *verbose=0*) → Tuple[object, Tuple[dict, dict]]

Load weights and initialize pytorch module from namespace arguments

dicee.static_funcs.**load_model_ensemble**(*path_of_experiment_folder: str*) → Tuple[*dicee.models.base_model.BaseKGE*, Tuple[pandas.DataFrame, pandas.DataFrame]]

Construct Ensemble Of weights and initialize pytorch module from namespace arguments

(1) Detect models under given path

(2) Accumulate parameters of detected models

(3) Normalize parameters

(4) Insert (3) into model.

dicee.static_funcs.**save_numpy_ndarray**(*, *data: numpy.ndarray*, *file_path: str*)

dicee.static_funcs.**numpy_data_type_changer**(*train_set: numpy.ndarray*, *num: int*) → numpy.ndarray

Detect most efficient data type for a given triples :param train_set: :param num: :return:

dicee.static_funcs.**save_checkpoint_model**(*model*, *path: str*) → None

Store Pytorch model into disk

dicee.static_funcs.**store**(*trainer*, *trained_model*, *model_name: str = 'model'*, *full_storage_path: str = None*, *save_embeddings_as_csv=False*) → None

Store trained_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param full_storage_path: path to save parameters. :param model_name: string representation of the name of the model. :param trained_model: an instance of BaseKGE see core.models.base_model . :param save_embeddings_as_csv: for easy access of embeddings. :return:

dicee.static_funcs.**add_noisy_triples**(*train_set: pandas.DataFrame*, *add_noise_rate: float*) → pandas.DataFrame

Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

dicee.static_funcs.**read_or_load_kg**(*args*, *cls*)

dicee.static_funcs.**intialize_model**(*args: dict*, *verbose=0*) → Tuple[object, str]

dicee.static_funcs.**load_json**(*p: str*) → dict

dicee.static_funcs.**save_embeddings**(*embeddings: numpy.ndarray*, *indexes*, *path: str*) → None

Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:

dicee.static_funcs.**random_prediction**(*pre_trained_kge*)

dicee.static_funcs.**deploy_triple_prediction**(*pre_trained_kge*, *str_subject*, *str_predicate*, *str_object*)

dicee.static_funcs.**deploy_tail_entity_prediction**(*pre_trained_kge*, *str_subject*, *str_predicate*, *top_k*)

dicee.static_funcs.**deploy_head_entity_prediction**(*pre_trained_kge*, *str_object*, *str_predicate*, *top_k*)

dicee.static_funcs.**deploy_relation_prediction**(*pre_trained_kge*, *str_subject*, *str_object*, *top_k*)

dicee.static_funcs.**vocab_to_parquet**(*vocab_to_idx*, *name*, *path_for_serialization*, *print_into*)

dicee.static_funcs.**create_experiment_folder**(*folder_name='Experiments'*)

dicee.static_funcs.**continual_training_setup_executor**(*executor*) → None

storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data

full_storage_path:str A path leading to a subdirectory containing KGE related data

dicee.static_funcs.**exponential_function**(*x: numpy.ndarray*, *lam: float*, *ascending_order=True*) → torch.FloatTensor

dicee.static_funcs.**load_numpy**(*path*) → numpy.ndarray

dicee.static_funcs.**evaluate**(*entity_to_idx*, *scores*, *easy_answers*, *hard_answers*)

@TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

dicee.static_funcs.**download_file**(*url*, *destination_folder='.'*)

dicee.static_funcs.**download_files_from_url**(*base_url: str*, *destination_folder='.'*) → None

Parameters

- **base_url** (e.g. "https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll") –
- **destination_folder**(*e.g. "KINSHIP-Keci-dim128-epoch256-KvsAll"*) –

dicee.static_funcs.**download_pretrained_model**(*url: str*) → str

dicee.static_funcs_training

Module Contents

Functions

evaluate_lp(model, triple_idx, num_entities, er_vocab, ...)	Evaluate model in a standard link prediction task
evaluate_bpe_lp(model, triple_idx, ...[, info])	
efficient_zero_grad(model)	

dicee.static_funcs_training.**evaluate_lp**(*model*, *triple_idx*, *num_entities*, *er_vocab: Dict[Tuple, List]*, *re_vocab: Dict[Tuple, List]*, *info='Eval Starts'*)

Evaluate model in a standard link prediction task

for each triple the rank is computed by taking the mean of the filtered missing head entity rank and the filtered missing tail entity rank :param model: :param triple_idx: :param info: :return:

dicee.static_funcs_training.**evaluate_bpe_lp**(*model*, *triple_idx: List[Tuple]*, *all_bpe_shaped_entities*, *er_vocab: Dict[Tuple, List]*, *re_vocab: Dict[Tuple, List]*, *info='Eval Starts'*)

dicee.static_funcs_training.**efficient_zero_grad**(*model*)

dicee.static_preprocess_funcs

Module Contents

Functions

timeit(func)	
preprocesses_input_args(args)	Sanity Checking in input arguments
create_constraints(→ Tuple[dict, dict, dict, dict])	(1) Extract domains and ranges of relations
get_er_vocab(data)	
get_re_vocab(data)	
get_ee_vocab(data)	
mapping_from_first_two_cols_to_third(tra	

Attributes

enable_log

dicee.static_preprocess_funcs.**enable_log = False**

dicee.static_preprocess_funcs.**timeit**(*func*)

dicee.static_preprocess_funcs.**preprocesses_input_args**(*args*)

Sanity Checking in input arguments

dicee.static_preprocess_funcs.**create_constraints**(*triples: numpy.ndarray*) → Tuple[dict, dict, dict, dict]

(1) Extract domains and ranges of relations

(2) Store a mapping from relations to entities that are outside of the domain and range. Create constraints entities based on the range of relations :param triples: :return:

dicee.static_preprocess_funcs.**get_er_vocab**(*data*)

dicee.static_preprocess_funcs.**get_re_vocab**(*data*)

dicee.static_preprocess_funcs.**get_ee_vocab**(*data*)

dicee.static_preprocess_funcs.**mapping_from_first_two_cols_to_third**(*train_set_idx*)

13.3 Package Contents

Classes

CMult	The CMult class represents a specific kind of mathematical object used in knowledge graph embeddings,
Pyke	Pyke is a physical embedding model for knowledge graphs, emphasizing the geometric relationships
DistMult	DistMult model for learning and inference in knowledge bases. It represents both entities
KeciBase	Without learning dimension scaling
Keci	The Keci class is a knowledge graph embedding model that incorporates Clifford algebra for embeddings.
TransE	TransE model for learning embeddings in multi-relational data. It is based on the idea of translating
DeCaL	Base class for all neural network modules.
ComplEx	ComplEx (Complex Embeddings for Knowledge Graphs) is a model that extends
AConEx	AConEx (Additive Convolutional ComplEx) extends the ConEx model by incorporating
AConvO	Additive Convolutional Octonion(AConvO) extends the base knowledge graph embedding model by integrating additive convolutional

continues on next page

Table 3 – continued from previous page

AConvQ	Additive Convolutional Quaternion Knowledge Graph Embeddings (AConvQ) model integrates
ConvQ	Convolutional Quaternion Knowledge Graph Embeddings (ConvQ) is a model that extends
ConvO	ConvO extends the base knowledge graph embedding model by integrating convolutional
ConEx	ConEx (Convolutional ComplEx) is a Knowledge Graph Embedding model that extends ComplEx embeddings with convolutional layers.
QMult	QMult extends the base knowledge graph embedding model by integrating quaternion
OMult	OMult extends the base knowledge graph embedding model by integrating octonion
Shallom	Shallom is a shallow neural model designed for relation prediction in knowledge graphs.
LFMult	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
PykeenKGE	A class for using knowledge graph embedding models implemented in Pykeen.
BytE	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
DICE_Trainer	Implements a training framework for knowledge graph embedding models using [PyTorch Lightning](https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html),
KGE	Knowledge Graph Embedding Class for interactive usage of pre-trained models
Execute	A class for Training, Retraining and Evaluation a model.
BPE_NegativeSamplingDataset	A PyTorch Dataset for handling negative sampling with Byte Pair Encoding (BPE) entities.
MultiLabelDataset	A dataset class for multi-label knowledge graph embedding tasks. This dataset is designed for models where
MultiClassClassificationDataset	A dataset class for multi-class classification tasks, specifically designed for the 1vsALL training strategy
OnevsAllDataset	A dataset for the One-vs-All (1vsAll) training strategy designed for knowledge graph embedding tasks.
KvsAll	Creates a dataset for K-vs-All training strategy, inheriting from torch.utils.data.Dataset.
AllvsAll	A dataset class for the All-versus-All (AllvsAll) training strategy suitable for knowledge graph embedding models.
KvsSampleDataset	Constructs a dataset for KvsSample training strategy, specifically designed for knowledge graph embedding models.
NegSampleDataset	A dataset for training knowledge graph embedding models using negative sampling.
TriplePredictionDataset	A dataset for triple prediction using negative sampling and label smoothing.
CVDataModule	A LightningDataModule for setting up data loaders for cross-validation training of knowledge graph embedding models.

continues on next page

Table 3 – continued from previous page

QueryGenerator	

Functions

create_recipriocal_triples(x)	Add inverse triples into dask dataframe
get_er_vocab(data[, file_path])	
get_re_vocab(data[, file_path])	
get_ee_vocab(data[, file_path])	
timeit(func)	
save_pickle(*[, data, file_path])	
load_pickle([file_path])	
select_model(args[, is_continual_training, storage_path])	
load_model(→ Tuple[object, Tuple[dict, dict]])	Load weights and initialize pytorch module from namespace arguments
load_model_ensemble(...)	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
save_numpy_ndarray(*, data, file_path)	
numpy_data_type_changer(→ numpy.ndarray)	Detect most efficient data type for a given triples
save_checkpoint_model(→ None)	Store Pytorch model into disk
store(→ None)	Store trained_model model and save embeddings into csv file.
add_noisy_triples(→ pandas.DataFrame)	Add randomly constructed triples
read_or_load_kg(args, cls)	
intialize_model(→ Tuple[object, str])	
load_json(→ dict)	
save_embeddings(→ None)	Save it as CSV if memory allows.
random_prediction(pre_trained_kge)	
deploy_triple_prediction(pre_trained_kge, str_subject, ...)	
deploy_tail_entity_prediction(pre_trained_k ...)	
deploy_head_entity_prediction(pre_trained_k ...)	
deploy_relation_prediction(pre_trained_kge, ...)	
vocab_to_parquet(vocab_to_idx, name, ...)	

continues on next page

Table 4 – continued from previous page

create_experiment_folder([folder_name])	
continual_training_setup_executor(→ None)	storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data
exponential_function(→ torch.FloatTensor)	
load_numpy(→ numpy.ndarray)	
evaluate(entity_to_idx, scores, easy_answers, hard_answers)	# @TODO: CD: Renamed this function
download_file(url[, destination_folder])	
download_files_from_url(→ None)	**param base_url**
download_pretrained_model(→ str)	
mapping_from_first_two_cols_to_third(tra	
timeit(func)	
load_pickle([file_path])	
reload_dataset(→ torch.utils.data.Dataset)	Reloads the dataset from disk and constructs a PyTorch dataset for training.
construct_dataset(→ torch.utils.data.Dataset)	Constructs a dataset based on the specified parameters and returns a PyTorch Dataset object.

Attributes

__version__

class dicee.**CMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

The CMult class represents a specific kind of mathematical object used in knowledge graph embeddings, involving Clifford algebra multiplication. It defines several algebraic structures based on the signature (p, q), such as Real Numbers, Complex Numbers, Quaternions, and others. The class provides functionality for performing Clifford multiplication, a generalization of the geometric product for vectors in a Clifford algebra.

TODO: Add mathematical format for sphinx.

Cl_(0,0) => Real Numbers

Cl_(0,1) =>

A multivector mathbf{a} = a_0 + a_1 e_1 A multivector mathbf{b} = b_0 + b_1 e_1

multiplication is isomorphic to the product of two complex numbers

mathbf{a} imes mathbf{b} = a_0 b_0 + a_0b_1 e1 + a_1 b_1 e_1 e_1
= (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) e_1

Cl_(2,0) =>

A multivector mathbf{a} = a_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 e_2 A multivector mathbf{b} = b_0 + b_1 e_1 + b_2 e_2 + b_{12} e_1 e_2

mathbf{a} imes mathbf{b} = a_0b_0 + a_0b_1 e_1 + a_0b_2e_2 + a_0 b_12 e_1 e_2

- a_1 b_0 e_1 + a_1b_1 e_1_e1 ..

Cl_(0,2) => Quaternions

`name`

The name identifier for the CMult class.

Type
str

`entity_embeddings`

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

`relation_embeddings`

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

`p`

Non-negative integer representing the number of positive square terms in the Clifford algebra.

Type
int

`q`

Non-negative integer representing the number of negative square terms in the Clifford algebra.

Type
int

`clifford_mul`(*x: torch.FloatTensor*, *y: torch.FloatTensor*, *p: int*, *q: int*) → tuple

Performs Clifford multiplication based on the given signature (p, q).

`score`(*head_ent_emb*, *rel_ent_emb*, *tail_ent_emb*) → torch.FloatTensor

Computes a scoring function for a head entity, relation, and tail entity embeddings.

`forward_triples`(*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples.

`forward_k_vs_all`(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities in the knowledge graph.

`clifford_mul`(*x: torch.FloatTensor*, *y: torch.FloatTensor*, *p: int*, *q: int*) → tuple

Performs Clifford multiplication in the Clifford algebra Cl_{p,q}. This method generalizes the geometric product of vectors in a Clifford algebra, handling different algebraic structures like real numbers, complex numbers, quaternions, etc., based on the signature (p, q).

Clifford multiplication Cl_{p,q} (mathbb{R})

ei ^2 = +1 for i =< i =< p ej ^2 = -1 for p < j =< p+q ei ej = -eje1 for i

eq j

x
[torch.FloatTensor] The first multivector operand with shape (n, d).

y
[torch.FloatTensor] The second multivector operand with shape (n, d).

p
[int] A non-negative integer representing the number of positive square terms in the Clifford algebra.

q
[int] A non-negative integer representing the number of negative square terms in the Clifford algebra.

tuple
The result of Clifford multiplication, a tuple of tensors representing the components of the resulting multivector.

score (*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes a scoring function for a given triple of head entity, relation, and tail entity embeddings. The method involves Clifford multiplication of the head entity and relation embeddings, followed by a calculation of the score with the tail entity embedding.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – Embedding of the head entity.
- **rel_ent_emb** (*torch.FloatTensor*) – Embedding of the relation.
- **tail_ent_emb** (*torch.FloatTensor*) – Embedding of the tail entity.

Returns
A tensor representing the score of the given triple.

Return type
torch.FloatTensor

forward_triples (*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples. This method is typically used in training or evaluation of knowledge graph embedding models. It applies Clifford multiplication to the embeddings of head entities and relations and then calculates the score with respect to the tail entity embeddings.

Parameters
x (*torch.LongTensor*) – A tensor with shape (n, 3) representing a batch of triples, where each triple consists of indices for a head entity, a relation, and a tail entity.

Returns
A tensor with shape (n,) containing the scores for each triple in the batch.

Return type
torch.FloatTensor

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities in the knowledge graph, often used in KvsAll evaluation. This method retrieves embeddings for heads and relations, performs Clifford multiplication, and then computes the inner product with all entity embeddings to get scores for every possible triple involving the given heads and relations.

Parameters
x (*torch.Tensor*) – A tensor with shape (n, 3) representing a batch of triples, where each

triple consists of indices for a head entity and a relation. The tail entity is to be compared against all possible entities.

Returns
A tensor with shape (n,) containing scores for each triple against all possible tail entities.

Return type
torch.FloatTensor

class dicee.**Pyke**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

Pyke is a physical embedding model for knowledge graphs, emphasizing the geometric relationships in the embedding space. The model aims to represent entities and relations in a way that captures the underlying structure of the knowledge graph.

name

The name identifier for the Pyke model.

Type
str

dist_func

A pairwise distance function to compute distances in the embedding space.

Type
torch.nn.PairwiseDistance

margin

The margin value used in the scoring function.

Type
float

forward_triples(*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples based on the physical embedding approach.

forward_triples(*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for a batch of triples based on the physical embedding approach.

The method calculates the Euclidean distance between the head and relation embeddings, and between the relation and tail embeddings. The average of these distances is subtracted from the margin to compute the score for each triple.

Parameters
x (*torch.LongTensor*) – A tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples. Lower scores indicate more likely triples according to the geometric arrangement of embeddings.

Return type
torch.FloatTensor

class dicee.**DistMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

DistMult model for learning and inference in knowledge bases. It represents both entities and relations using embeddings and uses a simple bilinear form to compute scores for triples.

This implementation of the DistMult model is based on the paper: 'Embedding Entities and Relations for Learning and Inference in Knowledge Bases' (https://arxiv.org/abs/1412.6575).

name

The name identifier for the DistMult model.

Type
str

k_vs_all_score (*emb_h: torch.FloatTensor*, *emb_r: torch.FloatTensor*, *emb_E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using embeddings for a batch of head entities and relations.

forward_k_vs_all (*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for all entities given a batch of head entities and relations.

forward_k_vs_sample (*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Computes scores for a sampled subset of entities given a batch of head entities and relations.

score (*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Computes the score of triples using DistMult's scoring function.

k_vs_all_score (*emb_h: torch.FloatTensor*, *emb_r: torch.FloatTensor*, *emb_E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using embeddings for a batch of head entities and relations.

This method multiplies the head entity and relation embeddings, applies a dropout and a normalization, and then computes the dot product with all tail entity embeddings.

Parameters

- **emb_h** (*torch.FloatTensor*) – Embeddings of head entities.
- **emb_r** (*torch.FloatTensor*) – Embeddings of relations.
- **emb_E** (*torch.FloatTensor*) – Embeddings of all entities.

Returns
Scores for all possible triples formed with the given head entities and relations against all entities.

Return type
torch.FloatTensor

forward_k_vs_all (*x: torch.LongTensor*) → torch.FloatTensor

Computes scores for all entities given a batch of head entities and relations.

This method is used for K-vs-All scoring, where the model predicts the likelihood of each entity being the tail entity in a triple with each head entity and relation pair in the batch.

Parameters
x (*torch.LongTensor*) – Tensor containing indices for head entities and relations.

Returns
Scores for all entities for each head entity and relation pair in the batch.

Return type
torch.FloatTensor

forward_k_vs_sample (*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Computes scores for a sampled subset of entities given a batch of head entities and relations.

This method is particularly useful when the full set of entities is too large to score with every batch and only a subset of entities is required.

Parameters

- **x** (*torch.LongTensor*) – Tensor containing indices for head entities and relations.
- **target_entity_idx** (*torch.LongTensor*) – Indices of the target entities against which the scores are to be computed.

Returns
Scores for each head entity and relation pair against the sampled subset of entities.

Return type
torch.FloatTensor

score(*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Computes the score of triples using DistMult's scoring function.

The scoring function multiplies head entity and relation embeddings, applies dropout and normalization, and computes the dot product with the tail entity embeddings.

Parameters

- **h** (*torch.FloatTensor*) – Embedding of the head entity.
- **r** (*torch.FloatTensor*) – Embedding of the relation.
- **t** (*torch.FloatTensor*) – Embedding of the tail entity.

Returns
The score of the triple.

Return type
torch.FloatTensor

class dicee.**KeciBase**(*args*)

Bases: *Keci*

Without learning dimension scaling

class dicee.**Keci**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

The Keci class is a knowledge graph embedding model that incorporates Clifford algebra for embeddings. It supports different dimensions of Clifford algebra by setting the parameters p and q. The class utilizes Clifford multiplication for embedding interactions and computes scores for knowledge graph triples.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model.

name

The name identifier for the Keci class.

Type
str

p

The parameter 'p' in Clifford algebra, representing the number of positive square terms.

Type
int

q

The parameter ‘q’ in Clifford algebra, representing the number of negative square terms.

Type
int

r

A derived attribute for dimension scaling based on ‘p’ and ‘q’.

Type
int

p_coefficients

Embedding for scaling coefficients of ‘p’ terms, if ‘p’ > 0.

Type
torch.nn.Embedding (optional)

q_coefficients

Embedding for scaling coefficients of ‘q’ terms, if ‘q’ > 0.

Type
torch.nn.Embedding (optional)

compute_sigma_pp(*hp: torch.Tensor*, *rp: torch.Tensor*) → torch.Tensor

Computes the sigma_pp component in Clifford multiplication.

compute_sigma_qq(*hq: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_qq component in Clifford multiplication.

compute_sigma_pq(*hp: torch.Tensor*, *hq: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_pq component in Clifford multiplication.

apply_coefficients(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → tuple

Applies scaling coefficients to the base vectors in Clifford algebra.

clifford_multiplication(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*) → tuple

Performs Clifford multiplication of head and relation embeddings.

construct_cl_multivector(*x: torch.FloatTensor*, *r: int*, *p: int*, *q: int*) → tuple

Constructs a multivector in Clifford algebra Cl_{p,q}(mathbb{R}^d).

forward_k_vs_with_explicit(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities using explicit Clifford multiplication.

k_vs_all_score(*bpe_head_ent_emb: torch.Tensor*, *bpe_rel_ent_emb: torch.Tensor*, *E: torch.Tensor*) → torch.FloatTensor

Computes scores for all triples using Clifford multiplication in a K-vs-All setup.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Wrapper function for K-vs-All scoring.

forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Computes scores for a sampled subset of entities.

score (*h: torch.Tensor*, *r: torch.Tensor*, *t: torch.Tensor*) → torch.FloatTensor

Computes the score for a given triple using Clifford multiplication.

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples.

Notes

The class is designed to work with embeddings in the context of knowledge graph completion tasks, leveraging the properties of Clifford algebra for embedding interactions.

compute_sigma_pp (*hp: torch.Tensor*, *rp: torch.Tensor*) → torch.Tensor

Computes the sigma_pp component in Clifford multiplication, representing the interactions between the positive square terms in the Clifford algebra.

sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k, TODO: Add mathematical format for sphinx.

sigma_{pp} captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

for k in range(i + 1, p):
results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

Parameters

- **hp** (*torch.Tensor*) – The 'p' part of the head entity embedding in Clifford algebra.
- **rp** (*torch.Tensor*) – The 'p' part of the relation embedding in Clifford algebra.

Returns
sigma_pp – The sigma_pp component of the Clifford multiplication.

Return type
torch.Tensor

compute_sigma_qq (*hq: torch.Tensor*, *rq: torch.Tensor*) → torch.Tensor

Computes the sigma_qq component in Clifford multiplication, representing the interactions between the negative square terms in the Clifford algebra.

TODO: Add mathematical format for sphinx.

sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k sigma_{q} captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

for k in range(j + 1, q):
results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

Parameters

- **hq** (*torch.Tensor*) – The 'q' part of the head entity embedding in Clifford algebra.
- **rq** (*torch.Tensor*) – The 'q' part of the relation embedding in Clifford algebra.

Returns
sigma_qq – The sigma_qq component of the Clifford multiplication.

Return type
torch.Tensor

compute_sigma_pq(*, *hp: torch.Tensor*, *hq: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*)
→ torch.Tensor

Computes the sigma_pq component in Clifford multiplication, representing the interactions between the positive and negative square terms in the Clifford algebra.

TODO: Add mathematical format for sphinx.

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] # sigma_pq = torch.zeros(b, r, p, q) # for i in range(p): # for j in range(q): # sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i] # print(sigma_pq.shape)

Parameters

- **hp** (*torch.Tensor*) – The 'p' part of the head entity embedding in Clifford algebra.
- **hq** (*torch.Tensor*) – The 'q' part of the head entity embedding in Clifford algebra.
- **rp** (*torch.Tensor*) – The 'p' part of the relation embedding in Clifford algebra.
- **rq** (*torch.Tensor*) – The 'q' part of the relation embedding in Clifford algebra.

Returns
sigma_pq – The sigma_pq component of the Clifford multiplication.

Return type
torch.Tensor

apply_coefficients(*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTe

Applies scaling coefficients to the base vectors in the Clifford algebra. This method is used for adjusting the contributions of different components in the algebra.

Parameters

- **h0** (*torch.Tensor*) – The scalar part of the head entity embedding.
- **hp** (*torch.Tensor*) – The 'p' part of the head entity embedding.
- **hq** (*torch.Tensor*) – The 'q' part of the head entity embedding.
- **r0** (*torch.Tensor*) – The scalar part of the relation embedding.
- **rp** (*torch.Tensor*) – The 'p' part of the relation embedding.
- **rq** (*torch.Tensor*) – The 'q' part of the relation embedding.

Returns
Tuple containing the scaled components of the head and relation embeddings.

Return type
tuple

clifford_multiplication (*h0: torch.Tensor*, *hp: torch.Tensor*, *hq: torch.Tensor*, *r0: torch.Tensor*, *rp: torch.Tensor*, *rq: torch.Tensor*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTe

Performs Clifford multiplication of head and relation embeddings. This method computes the various components of the Clifford product, combining the scalar, 'p', and 'q' parts of the embeddings.

TODO: Add mathematical format for sphinx.

h = h_0 + sum_{i=1}^p h_i e_i + sum_{j=p+1}^{p+q} h_j e_j r = r_0 + sum_{i=1}^p r_i e_i + sum_{j=p+1}^{p+q} r_j e_j

ei ^2 = +1 for i =< i =< p ej ^2 = -1 for p < j =< p+q ei ej = -eje1 for i

eq j

h r = sigma_0 + sigma_p + sigma_q + sigma_{pp} + sigma_{q}+ sigma_{pq} where

(1) sigma_0 = h_0 r_0 + sum_{i=1}^p (h_0 r_i) e_i - sum_{j=p+1}^{p+q} (h_j r_j) e_j

(2) sigma_p = sum_{i=1}^p (h_0 r_i + h_i r_0) e_i

(3) sigma_q = sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j

(4) sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k

(5) sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k

(6) sigma_{pq} = sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

h0
[torch.Tensor] The scalar part of the head entity embedding.

hp
[torch.Tensor] The 'p' part of the head entity embedding.

hq
[torch.Tensor] The 'q' part of the head entity embedding.

r0
[torch.Tensor] The scalar part of the relation embedding.

rp
[torch.Tensor] The 'p' part of the relation embedding.

rq
[torch.Tensor] The 'q' part of the relation embedding.

tuple
Tuple containing the components of the Clifford product.

construct_cl_multivector (*x: torch.FloatTensor*, *r: int*, *p: int*, *q: int*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors Cl_{p,q}(mathbb{R}^d)

Parameter

x
[torch.FloatTensor] The embedding vector with shape (n, d).

r
[int] The dimension of the scalar part.

p
[int] The number of positive square terms.

q
[int] The number of negative square terms.

returns

- **a0** (*torch.FloatTensor*) – Tensor with (n,r) shape
- **ap** (*torch.FloatTensor*) – Tensor with (n,r,p) shape
- **aq** (*torch.FloatTensor*) – Tensor with (n,r,q) shape

forward_k_vs_with_explicit (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples against all entities using explicit Clifford multiplication. This method is used for K-vs-All training and evaluation.

Parameters
x (*torch.Tensor*) – Tensor representing a batch of head entities and relations.

Returns
A tensor containing scores for each triple against all entities.

Return type
torch.FloatTensor

k_vs_all_score (*bpe_head_ent_emb: torch.Tensor*, *bpe_rel_ent_emb: torch.Tensor*, *E: torch.Tensor*) → torch.FloatTensor

Computes scores for all triples using Clifford multiplication in a K-vs-All setup. This method involves constructing multivectors for head entities and relations in Clifford algebra, applying coefficients, and computing interaction scores based on different components of the Clifford algebra.

Parameters

- **bpe_head_ent_emb** (*torch.Tensor*) – Batch of head entity embeddings in BPE (Byte Pair Encoding) format. Tensor shape: (batch_size, embedding_dim).
- **bpe_rel_ent_emb** (*torch.Tensor*) – Batch of relation embeddings in BPE format. Tensor shape: (batch_size, embedding_dim).
- **E** (*torch.Tensor*) – Tensor containing all entity embeddings. Tensor shape: (num_entities, embedding_dim).

Returns
Tensor containing the scores for each triple in the K-vs-All setting. Tensor shape: (batch_size, num_entities).

Return type
torch.FloatTensor

Notes

The method computes scores based on the basis of 1 (scalar part), the bases of 'p' (positive square terms), and the bases of 'q' (negative square terms). Additional computations involve sigma_pp, sigma_qq, and sigma_pq components in Clifford multiplication, corresponding to different interaction terms.

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

TODO: Add mathematical format for sphinx. Performs the forward pass for K-vs-All training and evaluation in knowledge graph embeddings. This method involves retrieving real-valued embedding vectors for head entities and relations mathbb{R}^d, constructing Clifford algebra multivectors for these embeddings according to Cl_{p,q}(mathbb{R}^d), performing Clifford multiplication, and computing the inner product with all entity embeddings.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of head entities and relations for the K-vs-All evaluation. Expected tensor shape: (n, 2), where 'n' is the batch size and '2' represents head entity and relation pairs.

Returns

A tensor containing the scores for each head entity and relation pair against all possible tail entities in the knowledge graph. Tensor shape: (n, |E|), where '|E|' is the number of entities in the knowledge graph.

Return type

torch.FloatTensor

Notes

This method is similar to the 'forward_k_vs_with_explicit' function in functionality. It is typically used in scenarios where every possible combination of a head entity and a relation is scored against all tail entities, commonly used in knowledge graph completion tasks.

forward_k_vs_sample (*x: torch.LongTensor*, *target_entity_idx: torch.LongTensor*) → torch.FloatTensor

TODO: Add mathematical format for sphinx.

Performs the forward pass for K-vs-Sample training in knowledge graph embeddings. This method involves retrieving real-valued embedding vectors for head entities and relations mathbb{R}^d, constructing Clifford algebra multivectors for these embeddings according to Cl_{p,q}(mathbb{R}^d), performing Clifford multiplication, and computing the inner product with a sampled subset of entity embeddings.

Parameters

- **x** (*torch.LongTensor*) – A tensor representing a batch of head entities and relations for the K-vs-Sample evaluation. Expected tensor shape: (n, 2), where 'n' is the batch size and '2' represents head entity and relation pairs.
- **target_entity_idx** (*torch.LongTensor*) – A tensor of target entity indices for sampling in the K-vs-Sample evaluation. Tensor shape: (n, sample_size), where 'sample_size' is the number of entities sampled.

Returns

A tensor containing the scores for each head entity and relation pair against the sampled subset of tail entities. Tensor shape: (n, sample_size).

Return type

torch.FloatTensor

Notes

This method is used in scenarios where every possible combination of a head entity and a relation is scored against a sampled subset of tail entities, commonly used in knowledge graph completion tasks with a large number of entities.

score (*h: torch.Tensor*, *r: torch.Tensor*, *t: torch.Tensor*) → torch.FloatTensor

Computes the score for a given triple using Clifford multiplication in the context of knowledge graph embeddings. This method involves constructing Clifford algebra multivectors for head entities, relations, and tail entities, applying coefficients, and computing interaction scores based on different components of the Clifford algebra.

Parameters

- **h** (*torch.Tensor*) – Tensor representing the embeddings of head entities. Expected shape: (n, d), where 'n' is the number of triples and 'd' is the embedding dimension.
- **r** (*torch.Tensor*) – Tensor representing the embeddings of relations. Expected shape: (n, d).
- **t** (*torch.Tensor*) – Tensor representing the embeddings of tail entities. Expected shape: (n, d).

Returns

Tensor containing the scores for each triple. Tensor shape: (n,).

Return type

torch.FloatTensor

Notes

The method computes scores based on the scalar part, the bases of 'p' (positive square terms), and the bases of 'q' (negative square terms) in Clifford algebra. It includes additional computations involving sigma_pp, sigma_qq, and sigma_pq components, which correspond to different interaction terms in the Clifford product.

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using Clifford multiplication. This method is involved in the forward pass of the model during training or evaluation. It retrieves embeddings for head entities, relations, and tail entities, constructs Clifford algebra multivectors, applies coefficients, and computes interaction scores based on different components of Clifford algebra.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of triples. Each triple consists of indices for a head entity, a relation, and a tail entity. Expected tensor shape: (n, 3), where 'n' is the number of triples.

Returns

A tensor containing the scores for each triple in the batch. Tensor shape: (n,), where 'n' is the number of triples.

Return type

torch.FloatTensor

Notes

The method computes scores based on the scalar part, the bases of 'p' (positive square terms), and the bases of 'q' (negative square terms) in Clifford algebra. It includes additional computations involving sigma_pp, sigma_qq, and sigma_pq components, corresponding to different interaction terms in the Clifford product.

class dicee.**TransE**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

TransE model for learning embeddings in multi-relational data. It is based on the idea of translating embeddings for head entities by the relation vector to approach the tail entity embeddings in the embedding space.

This implementation of TransE is based on the paper: 'Translating Embeddings for Modeling Multi-relational Data' (https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf).

name

The name identifier for the TransE model.

Type
str

_norm

The norm used for computing pairwise distances in the embedding space.

Type
int

margin

The margin value used in the scoring function.

Type
int

score(*head_ent_emb: torch.Tensor*, *rel_ent_emb: torch.Tensor*, *tail_ent_emb: torch.Tensor*)
→ torch.Tensor

Computes the score of triples using the TransE scoring function.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for all entities given a head entity and a relation.

score(*head_ent_emb: torch.Tensor*, *rel_ent_emb: torch.Tensor*, *tail_ent_emb: torch.Tensor*)
→ torch.Tensor

Computes the score of triples using the TransE scoring function.

The scoring function computes the L2 distance between the translated head entity and the tail entity embeddings and subtracts this distance from the margin.

Parameters

- **head_ent_emb** (*torch.Tensor*) – Embedding of the head entity.
- **rel_ent_emb** (*torch.Tensor*) – Embedding of the relation.
- **tail_ent_emb** (*torch.Tensor*) – Embedding of the tail entity.

Returns
The score of the triple.

Return type
torch.Tensor

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for all entities given a head entity and a relation.

This method is used for K-vs-All scoring, where the model predicts the likelihood of each entity being the tail entity in a triple with each head entity and relation.

Parameters

x (*torch.Tensor*) – Tensor containing indices for head entities and relations.

Returns

Scores for all entities for each head entity and relation pair.

Return type

torch.FloatTensor

class dicee.**DeCaL**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,3) shape

rtype
torch.FloatTensor with (n) shape

cl_pqr (*a*)

Input: tensor(batch_size, emb_dim) —-> output: tensor with 1+p+q+r components with size (batch_size, emb_dim/(1+p+q+r)) each.

1) takes a tensor of size (batch_size, emb_dim), split it into 1 + p + q +r components, hence 1+p+q+r must be a divisor of the emb_dim. 2) Return a list of the 1+p+q+r components vectors, each are tensors of size (batch_size, emb_dim/(1+p+q+r))

compute_sigmas_single (*list_h_emb*, *list_r_emb*, *list_t_emb*)

here we compute all the sums with no others vectors interaction taken with the scalar product with t, that is, 1) s0 = h_0r_0t_0 2) s1 = sum_{i=1}^{p}h_ir_it_0 3) s2 = sum_{j=p+1}^{p+q}h_jr_jt_0 4) s3 = sum_{i=1}^{q}(h_0r_it_i + h_ir_0t_i) 5) s4 = sum_{i=p+1}^{p+q}(h_0r_it_i + h_ir_0t_i) 5) s5 = sum_{i=p+q+1}^{p+q+r}(h_0r_it_i + h_ir_0t_i)

and return:

*) sigma_0t = sigma_0 cdot t_0 = s0 + s1 -s2 *) s3, s4 and s5

compute_sigmas_multivect (*list_h_emb*, *list_r_emb*)

Here we compute and return all the sums with vectors interaction for the same and different bases.

For same bases vectors interaction we have

1) sigma_pp = sum_{i=1}^{p-1}sum_{i'=i+1}^{p}(h_ir_{i'}-h_{i'}r_i) (models the interactions between e_i and e_i' for 1 <= i, i' <= p)
2) sigma_qq = sum_{j=p+1}^{p+q-1}sum_{j'=j+1}^{p+q}(h_jr_{j'}-h_{j'} (models the interactions between e_j and e_j' for p+1 <= j, j' <= p+q)
3) sigma_rr = sum_{k=p+q+1}^{p+q+r-1}sum_{k'=k+1}^{p}(h_kr_{k'}-h_{k'}r_k) (models the interactions between e_k and e_k' for p+q+1 <= k, k' <= p+q+r)

For different base vector interactions, we have

4) sigma_pq = sum_{i=1}^{p}sum_{j=p+1}^{p+q}(h_ir_j - h_jr_i) (interactionsn between e_i and e_j for 1<=i <=p and p+1<= j <= p+q)
5) sigma_pr = sum_{i=1}^{p}sum_{k=p+q+1}^{p+q+r}(h_ir_k - h_kr_i) (interactionsn between e_i and e_k for 1<=i <=p and p+q+1<= k <= p+q+r)
6) sigma_qr = sum_{j=p+1}^{p+q}sum_{j=p+q+1}^{p+q+r}(h_jr_k - h_kr_j) (interactionsn between e_j and e_k for p+1 <= j <=p+q and p+q+1<= j <= p+q+r)

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Kvsall training

(1) Retrieve real-valued embedding vectors for heads and relations mathbb{R}^d .
(2) Construct head entity and relation embeddings according to Cl_{p,q}(mathbb{R}^d) .
(3) Perform Cl multiplication
(4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this funcitons are identical Parameter ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, |E|) shape

apply_coefficients (*h0*, *hp*, *hq*, *hk*, *r0*, *rp*, *rq*, *rk*)

Multiplying a base vector with its scalar coefficient

construct_cl_multivector (*x: torch.FloatTensor*, *re: int*, *p: int*, *q: int*, *r: int*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors Cl_{p,q,r}(mathbb{R}^d)

Parameter

x: torch.FloatTensor with (n,d) shape

returns

- **a0** (*torch.FloatTensor*)
- **ap** (*torch.FloatTensor*)
- **aq** (*torch.FloatTensor*)
- **ar** (*torch.FloatTensor*)

compute_sigma_pp (*hp*, *rp*)

sigma_{p,p}^* = sum_{i=1}^{p-1}sum_{i'=i+1}^{p}(x_iy_{i'}-x_{i'}y_i)

sigma_{pp} captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

for k in range(i + 1, p):
results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

compute_sigma_qq (*hq*, *rq*)

Compute sigma_{q,q}^* = sum_{j=p+1}^{p+q-1}sum_{j'=j+1}^{p+q}(x_jy_{j'}-x_{j'}y_j) Eq. 16 sigma_{q} captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3 , and e_2 e_3 This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

for k in range(j + 1, q):
results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

compute_sigma_rr (*hk*, *rk*)

sigma_{r,r}^* = sum_{k=p+q+1}^{p+q+r-1}sum_{k'=k+1}^{p}(x_ky_{k'}-x_{k'}y_k)

compute_sigma_pq(*, *hp*, *hq*, *rp*, *rq*)

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):
sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

print(sigma_pq.shape)

compute_sigma_pr(*, *hp*, *hk*, *rp*, *rk*)

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):
sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

print(sigma_pq.shape)

compute_sigma_qr(*, *hq*, *hk*, *rq*, *rk*)

sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):
sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

print(sigma_pq.shape)

class dicee.**ComplEx**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

ComplEx (Complex Embeddings for Knowledge Graphs) is a model that extends the base knowledge graph embedding approach by using complex-valued embeddings. It emphasizes the interaction of real and imaginary components of embeddings to capture the asymmetric relationships often found in knowledge graphs.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, learning rate, and regularization methods.

name

The name identifier for the ComplEx model.

Type
str

score(head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor,

tail_ent_emb: torch.FloatTensor) -> torch.FloatTensor

Computes the score of a triple using the ComplEx scoring function.

k_vs_all_score(emb_h: torch.FloatTensor, emb_r: torch.FloatTensor,

emb_E: torch.FloatTensor) -> torch.FloatTensor

Computes scores in a K-vs-All setting using complex-valued embeddings.

forward_k_vs_all(*x: torch.LongTensor*) → torch.FloatTensor

Performs a forward pass for K-vs-All scoring, returning scores for all entities.

Notes

ComplEx is particularly suited for modeling asymmetric relations and has been shown to perform well on various knowledge graph benchmarks. The use of complex numbers allows the model to encode additional information compared to real-valued models.

static score (*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Compute the scoring function for a given triple using complex-valued embeddings.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – The complex embedding of the head entity.
- **rel_ent_emb** (*torch.FloatTensor*) – The complex embedding of the relation.
- **tail_ent_emb** (*torch.FloatTensor*) – The complex embedding of the tail entity.

Returns
The score of the triple calculated using the Hermitian dot product of complex embeddings.

Return type
torch.FloatTensor

Notes

The scoring function exploits the complex vector space to model the interactions between entities and relations. It involves element-wise multiplication and summation of real and imaginary parts.

static k_vs_all_score (*emb_h: torch.FloatTensor*, *emb_r: torch.FloatTensor*, *emb_E: torch.FloatTensor*) → torch.FloatTensor

Compute scores for a head entity and relation against all entities in a K-vs-All scenario.

Parameters

- **emb_h** (*torch.FloatTensor*) – The complex embedding of the head entity.
- **emb_r** (*torch.FloatTensor*) – The complex embedding of the relation.
- **emb_E** (*torch.FloatTensor*) – The complex embeddings of all possible tail entities.

Returns
Scores for all possible triples formed with the given head entity and relation.

Return type
torch.FloatTensor

Notes

This method is useful for tasks like link prediction where the model predicts the likelihood of a relation between a given entity pair.

forward_k_vs_all (*x: torch.LongTensor*) → torch.FloatTensor

Perform a forward pass for K-vs-all scoring using complex-valued embeddings.

Parameters
x (*torch.LongTensor*) – Tensor containing indices for head entities and relations.

Returns
Scores for all triples formed with the given head entities and relations against all entities.

Return type
torch.FloatTensor

Notes

This method is typically used in training and evaluation of the model in a link prediction setting, where the goal is to rank all possible tail entities for a given head entity and relation.

class dicee.**AConEx**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

AConEx (Additive Convolutional ComplEx) extends the ConEx model by incorporating additive connections in the convolutional operations. This model integrates convolutional neural networks with complex-valued embeddings, emphasizing additive feature interactions for knowledge graph embeddings.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, kernel size, number of output channels, and dropout rates.

name

The name identifier for the AConEx model.

Type
str

conv2d

A 2D convolutional layer used for processing complex-valued embeddings.

Type
torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

norm_fc1

Normalization layer applied after the fully connected layer.

Type
Normalizer

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(C_1: Tuple[torch.Tensor, torch.Tensor],

C_2: Tuple[torch.Tensor, torch.Tensor]) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Performs a residual convolution operation on two complex-valued embeddings.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional operations on embeddings.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*)

Computes scores against a sampled subset of entities using convolutional operations.

Notes

AConEx aims to enhance the modeling capabilities of knowledge graph embeddings by adding more complex interaction patterns through convolutional layers, potentially improving performance on tasks like link prediction.

residual_convolution(*C_1: Tuple[torch.Tensor, torch.Tensor]*,
C_2: Tuple[torch.Tensor, torch.Tensor])
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Computes the residual convolution of two complex-valued embeddings. This method is a core part of the AConEx model, applying convolutional neural network techniques to complex-valued embeddings to capture intricate relationships in the data.

Parameters

- **C_1** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple of two PyTorch tensors representing the real and imaginary components of the first complex-valued embedding.
- **C_2** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple of two PyTorch tensors representing the real and imaginary components of the second complex-valued embedding.

Returns

A tuple of four tensors, each representing a component of the convolutionally transformed embeddings. These components correspond to the modified real and imaginary parts of the input embeddings.

Return type

Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Notes

The method concatenates the real and imaginary components of the embeddings and applies a 2D convolution, followed by batch normalization, ReLU activation, dropout, and a fully connected layer. This convolutional process is designed to enhance the model's ability to capture complex patterns in knowledge graph embeddings.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional and additive operations on complex-valued embeddings. This method evaluates the performance of the model by computing scores for each head entity and relation pair against all possible tail entities.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of head entities and relations. Expected tensor shape: (batch_size, 2), where 'batch_size' is the number of head entity and relation pairs.

Returns

A tensor containing the scores for each head entity and relation pair against all possible tail entities. Tensor shape: (batch_size, |E|), where '|E|' is the number of entities in the knowledge graph.

Return type

torch.FloatTensor

Notes

The method first retrieves embeddings for head entities and relations, splits them into real and imaginary parts, and applies a convolutional operation. It then computes the Hermitian inner product with all tail entity embeddings, using an additive approach that combines the convolutional results with the original embeddings. This technique aims to capture complex relational patterns in the knowledge graph.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations and additive connections on complex-valued embeddings. This method is key for evaluating the model's performance on individual triples within the knowledge graph.

Parameters

x (*torch.Tensor*) – A tensor representing a batch of triples. Each triple consists of indices for a head entity, a relation, and a tail entity. Expected tensor shape: (n, 3), where 'n' is the number of triples.

Returns

A tensor containing the scores for each triple in the batch. Tensor shape: (n,), where 'n' is the number of triples.

Return type

torch.FloatTensor

Notes

The method retrieves embeddings for head entities, relations, and tail entities, and splits them into real and imaginary parts. It then applies a convolution operation on these embeddings and computes the Hermitian inner product, enhanced with an additive connection. This approach allows the model to capture complex relational patterns within the knowledge graph, potentially improving prediction accuracy and interpretability.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of samples (entity pairs) given a batch of queries. This method is used to predict the scores for different tail entities for a set of query triples.

Parameters

- **x** (*torch.Tensor*) – A tensor representing a batch of query triples. Each triple consists of indices for a head entity, a relation, and a dummy tail entity (used for scoring). Expected tensor shape: (n, 3), where 'n' is the number of query triples.
- **target_entity_idx** (*torch.Tensor*) – A tensor containing the indices of the target tail entities for which scores are to be predicted. Expected tensor shape: (n, m), where 'n' is the number of queries and 'm' is the number of target entities.

Returns

A tensor containing the scores for each query-triple and target-entity pair. Tensor shape: (n, m), where 'n' is the number of queries and 'm' is the number of target entities.

Return type

torch.FloatTensor

Notes

This method retrieves embeddings for the head entities and relations in the query triples, splits them into real and imaginary parts, and applies convolutional operations with additive connections to capture complex patterns. It also retrieves embeddings for the target tail entities and computes Hermitian inner products to obtain scores, allowing the model to rank the tail entities based on their relevance to the queries.

class dicee.**AConvO**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Octonion(AConvO) extends the base knowledge graph embedding model by integrating additive convolutional operations with octonion algebra. This model applies convolutional neural networks to octonion-based embeddings, capturing complex interactions in knowledge graphs.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, number of output channels, kernel size, and dropout rates.

name

The name identifier for the AConvO model.

Type

str

conv2d

A 2D convolutional layer used for processing octonion-based embeddings.

Type

torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type

int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type

torch.nn.Linear

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type

torch.nn.BatchNorm2d

norm_fc1

Normalization layer applied after the fully connected layer.

Type

Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, ..., *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, ...]

Normalizes octonion components to unit length.

residual_convolution(*self*, *O_1: Tuple[torch.Tensor, ...]*, *O_2: Tuple[torch.Tensor, ...]*) → Tuple[torch.Tensor, ...]

Performs a residual convolution operation on two octonion embeddings.

forward_triples(*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_all(*x: torch.Tensor*)

Computes scores against a sampled subset of entities using convolutional operations.

Notes

AConvO aims to enhance the modeling capabilities of knowledge graph embeddings by adding more complex interaction patterns through convolutional layers, potentially improving performance on tasks like link prediction.

static octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, *emb_rel_e2: torch.Tensor*, *emb_rel_e3: torch.Tensor*, *emb_rel_e4: torch.Tensor*, *emb_rel_e5: torch.Tensor*, *emb_rel_e6: torch.Tensor*, *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Normalizes the components of an octonion to unit length.

Each component of the octonion is divided by the square root of the sum of the squares of all components.

Parameters

- **emb_rel_e0** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e1** (*torch.Tensor*) – The eight components of an octonion.
- **...** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e7** (*torch.Tensor*) – The eight components of an octonion.

Returns
The normalized components of the octonion.

Return type
Tuple[torch.Tensor, …]

residual_convolution(
O_1: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
O_2: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Performs a residual convolution operation on two sets of octonion embeddings.

The method combines two octonion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **O_1** (*Tuple[torch.Tensor, ...]*) – The first set of octonion embeddings.

- **O_2** (*Tuple[torch.Tensor, ...]*) – The second set of octonion embeddings.

Returns
The resulting octonion embeddings after the convolutional operation.

Return type
Tuple[torch.Tensor, …]

forward_triples(*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

The method processes head, relation, and tail embeddings using convolutional layers and computes the scores of the triples.

Parameters
x (*torch.Tensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.Tensor

forward_k_vs_all(*x: torch.Tensor*) → torch.Tensor

Compute scores for a head entity and a relation (h,r) against all entities in the knowledge graph.

Given a head entity and a relation (h, r), this method computes scores for (h, r, x) for all entities x in the knowledge graph.

Parameters
x (*torch.Tensor*) – A tensor containing indices for head entities and relations.

Returns
A tensor of scores representing the compatibility of (h, r, x) for all entities x in the knowledge graph.

Return type
torch.Tensor

Notes

This method supports batch processing, allowing the input tensor *x* to contain multiple head entities and relations.

The scores indicate how well each entity x in the knowledge graph fits the (h, r) pattern, with higher scores indicating better compatibility.

class dicee.**AConvQ**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Quaternion Knowledge Graph Embeddings (AConvQ) model integrates quaternion algebra with convolutional neural networks for knowledge graph embeddings. This model is designed to capture complex interactions in knowledge graphs by applying additive convolutions to quaternion-based entity and relation embeddings.

name

The name identifier for the AConvQ model.

Type
str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

conv2d

A 2D convolutional layer used for processing quaternion embeddings.

Type
torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

bn_conv1

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

bn_conv2

Normalization layer applied after the fully connected layer.

Type
Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(*Q_1*, *Q_2*)

Performs an additive residual convolution operation on two sets of quaternion embeddings.

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using additive convolutional operations on quaternion embeddings.

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

residual_convolution(
Q_1: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor],
Q_2: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor])
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Performs a residual convolution operation on two sets of quaternion embeddings.

The method combines two quaternion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **Q_1** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The first set of quaternion embeddings.
- **Q_2** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The second set of quaternion embeddings.

Returns
The resulting quaternion embeddings after the convolutional operation.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on quaternion embeddings.

The method processes head, relation, and tail embeddings using quaternion algebra and convolutional layers and computes the scores of the triples.

Parameters
indexed_triple (*torch.FloatTensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.FloatTensor

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

This method retrieves embeddings for the head entities and relations from the input tensor *x*, applies necessary dropout and normalization, and then computes scores against all entities in the knowledge graph.

Parameters
x (*torch.FloatTensor*) – A tensor containing indices for head entities and relations.

Returns
Scores for all entities for the given batch of head entities and relations.

Return type
torch.FloatTensor

class dicee.**ConvQ**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings (ConvQ) is a model that extends the base knowledge graph embedding approach by using quaternion algebra and convolutional neural networks. This model aims to capture complex interactions in knowledge graphs by applying convolutions to quaternion-based entity and relation embeddings.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, number of output channels, kernel size, and dropout rates.

name

The name identifier for the ConvQ model.

Type
str

entity_embeddings

Embedding layer for entities in the knowledge graph.

Type
torch.nn.Embedding

relation_embeddings

Embedding layer for relations in the knowledge graph.

Type
torch.nn.Embedding

conv2d

A 2D convolutional layer used for processing quaternion embeddings.

Type
torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type
int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

bn_conv1

First batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

bn_conv2

Second normalization layer applied after the fully connected layer.

Type
Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(*Q_1*, *Q_2*)

Performs a residual convolution operation on two sets of quaternion embeddings.

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on quaternion embeddings.

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

Notes

ConvQ leverages the properties of quaternions, a number system that extends complex numbers, to represent and process the embeddings of entities and relations. The convolutional layers aim to capture spatial relationships and complex patterns in the embeddings.

residual_convolution(
Q_1: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor],
Q_2: Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor])
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Performs a residual convolution operation on two sets of quaternion embeddings.

The method combines two quaternion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **Q_1** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The first set of quaternion embeddings.
- **Q_2** (*Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]*) – The second set of quaternion embeddings.

Returns
The resulting quaternion embeddings after the convolutional operation.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

forward_triples(*indexed_triple: torch.FloatTensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on quaternion embeddings.

The method processes head, relation, and tail embeddings using quaternion algebra and convolutional layers and computes the scores of the triples.

Parameters
indexed_triple (*torch.FloatTensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.FloatTensor

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

This method retrieves embeddings for the head entities and relations from the input tensor *x*, applies necessary dropout and normalization, and then computes scores against all entities in the knowledge graph.

Parameters
x (*torch.FloatTensor*) – A tensor containing indices for head entities and relations.

Returns
Scores for all entities for the given batch of head entities and relations.

Return type
torch.FloatTensor

class dicee.**ConvO**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

ConvO extends the base knowledge graph embedding model by integrating convolutional operations with octonion algebra. This model applies convolutional neural networks to octonion-based embeddings, capturing complex interactions in knowledge graphs.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, number of output channels, kernel size, and dropout rates.

name

The name identifier for the ConvO model.

Type

str

conv2d

A 2D convolutional layer used for processing octonion-based embeddings.

Type

torch.nn.Conv2d

fc_num_input

The number of input features for the fully connected layer.

Type

int

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type

torch.nn.Linear

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type

torch.nn.BatchNorm2d

norm_fc1

Normalization layer applied after the fully connected layer.

Type

Normalizer

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type

torch.nn.Dropout2d

octonion_normalizer(*emb_rel_e0*, *emb_rel_e1*, ..., *emb_rel_e7*)

Normalizes octonion components to unit length.

residual_convolution(*O_1*, *O_2*)

Performs a residual convolution operation on two octonion embeddings.

forward_triples(*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_all(*x: torch.Tensor*)

Computes scores against a sampled subset of entities using convolutional operations.

Notes

ConvO aims to enhance the modeling capabilities of knowledge graph embeddings by adding more complex interaction patterns through convolutional layers, potentially improving performance on tasks like link prediction.

static octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, *emb_rel_e2: torch.Tensor*, *emb_rel_e3: torch.Tensor*, *emb_rel_e4: torch.Tensor*, *emb_rel_e5: torch.Tensor*, *emb_rel_e6: torch.Tensor*, *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Normalizes the components of an octonion to unit length.

Each component of the octonion is divided by the square root of the sum of the squares of all components.

Parameters

- **emb_rel_e0** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e1** (*torch.Tensor*) – The eight components of an octonion.
- **...** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e7** (*torch.Tensor*) – The eight components of an octonion.

Returns

The normalized components of the octonion.

Return type

Tuple[torch.Tensor, …]

residual_convolution(*O_1: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens*
O_2: Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Performs a residual convolution operation on two sets of octonion embeddings.

The method combines two octonion embeddings and applies a convolutional operation followed by batch normalization, dropout, and a fully connected layer.

Parameters

- **O_1** (*Tuple[torch.Tensor, ...]*) – The first set of octonion embeddings.
- **O_2** (*Tuple[torch.Tensor, ...]*) – The second set of octonion embeddings.

Returns

The resulting octonion embeddings after the convolutional operation.

Return type

Tuple[torch.Tensor, …]

forward_triples(*x: torch.Tensor*) → torch.Tensor

Computes scores for a batch of triples using convolutional operations.

The method processes head, relation, and tail embeddings using convolutional layers and computes the scores of the triples.

Parameters
x (*torch.Tensor*) – Tensor containing indices for head entities, relations, and tail entities.

Returns
Scores for the given batch of triples.

Return type
torch.Tensor

forward_k_vs_all(*x: torch.Tensor*) → torch.Tensor

Given a batch of head entities and relations (h,r), this method computes scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,…,0.8], shape=> (1, |Entities|) Given a batch of head entities and relations => shape (size of batch,| Entities|)

Parameters
x (*torch.Tensor*) – A tensor representing a batch of input triples in the form of (head entities, relations).

Returns
Scores for the input triples against all possible tail entities.

Return type
torch.Tensor

Notes

- The input *x* is a tensor of shape (batch_size, 2), where each row represents a pair of head entities and relations.
- **The method follows the following steps:**
  (1) Retrieve embeddings & Apply Dropout & Normalization.
  (2) Split the embeddings into real and imaginary parts.
  (3) Apply convolution operation on the real and imaginary parts.
  (4) Perform quaternion multiplication.
  (5) Compute scores for all entities.

The method returns a tensor of shape (batch_size, num_entities) where each row contains scores for each entity in the knowledge graph.

class dicee.**ConEx**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

ConEx (Convolutional ComplEx) is a Knowledge Graph Embedding model that extends ComplEx embeddings with convolutional layers. It integrates convolutional neural networks into the embedding process to capture complex patterns in the data.

Parameters
args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, kernel size, number of output channels, and dropout rates.

name

The name identifier for the ConEx model.

Type
str

conv2d

A 2D convolutional layer used for processing complex-valued embeddings.

Type
torch.nn.Conv2d

fc1

A fully connected linear layer for compressing the output of the convolutional layer.

Type
torch.nn.Linear

norm_fc1

Normalization layer applied after the fully connected layer.

Type
Normalizer

bn_conv2d

Batch normalization layer applied after the convolutional operation.

Type
torch.nn.BatchNorm2d

feature_map_dropout

Dropout layer applied to the output of the convolutional layer.

Type
torch.nn.Dropout2d

residual_convolution(*C_1: Tuple[torch.Tensor, torch.Tensor]*, *C_2: Tuple[torch.Tensor, torch.Tensor]*) → Tuple[torch.Tensor, torch.Tensor]

Performs a residual convolution operation on two complex-valued embeddings.

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional operations on embeddings.

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*) → torch.Tensor

Computes scores against a sampled subset of entities using convolutional operations.

Notes

ConEx combines complex-valued embeddings with convolutional neural networks to capture intricate patterns and interactions in the knowledge graph, potentially leading to improved performance on tasks like link prediction.

residual_convolution(*C_1: Tuple[torch.Tensor, torch.Tensor]*, *C_2: Tuple[torch.Tensor, torch.Tensor]*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Computes the residual score of two complex-valued embeddings by applying convolutional operations. This method is a key component of the ConEx model, combining complex embeddings with convolutional neural networks.

Parameters

- **C_1** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple consisting of two PyTorch tensors representing the real and imaginary components of the first complex-valued embedding.

- **C_2** (*Tuple[torch.Tensor, torch.Tensor]*) – A tuple consisting of two PyTorch tensors representing the real and imaginary components of the second complex-valued embedding.

Returns
A tuple of two tensors, representing the real and imaginary parts of the convolutionally transformed embeddings.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

Notes

The method involves concatenating the real and imaginary components of the embeddings, applying a 2D convolution, followed by batch normalization, ReLU activation, dropout, and a fully connected layer. This process is intended to capture complex interactions between the embeddings in a convolutional manner.

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using convolutional operations on complex-valued embeddings. This method is used for evaluating the performance of the model by computing scores for each head entity and relation pair against all possible tail entities.

Parameters
x (*torch.Tensor*) – A tensor representing a batch of head entities and relations. Expected tensor shape: (n, 2), where 'n' is the batch size and '2' represents head entity and relation pairs.

Returns
A tensor containing the scores for each head entity and relation pair against all possible tail entities. Tensor shape: (n, |E|), where '|E|' is the number of entities in the knowledge graph.

Return type
torch.FloatTensor

Notes

The method retrieves embeddings for head entities and relations, splits them into real and imaginary parts, and applies a convolution operation. It then computes the Hermitian product of the transformed embeddings with all tail entity embeddings to generate scores. This approach allows for capturing complex relational patterns in the knowledge graph.

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Computes scores for a batch of triples using convolutional operations on complex-valued embeddings. This method is crucial for evaluating the performance of the model on individual triples in the knowledge graph.

Parameters
x (*torch.Tensor*) – A tensor representing a batch of triples. Each triple consists of indices for a head entity, a relation, and a tail entity. Expected tensor shape: (n, 3), where 'n' is the number of triples.

Returns
A tensor containing the scores for each triple in the batch. Tensor shape: (n,), where 'n' is the number of triples.

Return type
torch.FloatTensor

Notes

The method retrieves embeddings for head entities, relations, and tail entities, and splits them into real and imaginary parts. It then applies a convolution operation on these embeddings and computes the Hermitian inner product, which involves a combination of real and imaginary parts of the embeddings. This process is designed to capture complex relational patterns and interactions within the knowledge graph, leveraging the power of convolutional neural networks.

forward_k_vs_sample(*x: torch.Tensor*, *target_entity_idx: torch.Tensor*) → torch.Tensor

Computes scores against a sampled subset of entities using convolutional operations on complex-valued embeddings. This method is particularly useful for large knowledge graphs where computing scores against all entities is computationally expensive.

Parameters

- **x** (*torch.Tensor*) – A tensor representing a batch of head entities and relations. Expected tensor shape: (batch_size, 2), where 'batch_size' is the number of head entity and relation pairs.
- **target_entity_idx** (*torch.Tensor*) – A tensor of target entity indices for sampling. Tensor shape: (batch_size, num_selected_entities).

Returns

A tensor containing the scores for each head entity and relation pair against the sampled subset of tail entities. Tensor shape: (batch_size, num_selected_entities).

Return type

torch.Tensor

Notes

The method first retrieves and processes the embeddings for head entities and relations. It then applies a convolution operation and computes the Hermitian inner product with the embeddings of the sampled tail entities. This process enables capturing complex relational patterns in a computationally efficient manner.

class dicee.**QMult**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

QMult extends the base knowledge graph embedding model by integrating quaternion algebra. This model leverages the properties of quaternions to represent and process the embeddings of entities and relations in a knowledge graph, aiming to capture complex interactions and patterns.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions and learning rate.

name

The name identifier for the QMult model.

Type

str

quaternion_normalizer(*x: torch.FloatTensor*) → torch.FloatTensor

Normalizes the length of relation vectors.

score(*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes the score of a triple using quaternion multiplication.

k_vs_all_score (*bpe_head_ent_emb: torch.FloatTensor*, *bpe_rel_ent_emb: torch.FloatTensor*, *E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using quaternion embeddings.

forward_k_vs_all (*x: torch.FloatTensor*) → torch.FloatTensor

Performs a forward pass for K-vs-All scoring, returning scores for all entities.

forward_k_vs_sample (*x: torch.FloatTensor*, *target_entity_idx: int*) → torch.FloatTensor

Performs a forward pass for K-vs-Sample scoring, returning scores for the specified entities.

quaternion_multiplication_followed_by_inner_product (*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Performs quaternion multiplication followed by inner product, returning triple scores.

quaternion_multiplication_followed_by_inner_product (*h: torch.FloatTensor*, *r: torch.FloatTensor*, *t: torch.FloatTensor*) → torch.FloatTensor

Performs quaternion multiplication followed by inner product.

Parameters

- **h** (*torch.FloatTensor*) – The head representations. Shape: (**batch_dims*, dim)
- **r** (*torch.FloatTensor*) – The relation representations. Shape: (**batch_dims*, dim)
- **t** (*torch.FloatTensor*) – The tail representations. Shape: (**batch_dims*, dim)

Returns

Triple scores.

Return type

torch.FloatTensor

static quaternion_normalizer (*x: torch.FloatTensor*) → torch.FloatTensor

TODO: Add mathematical format for sphinx. Normalize the length of relation vectors, if the forward constraint has not been applied yet.

The absolute value of a quaternion is calculated as follows: .. math:

```
|a + bi + cj + dk| = \sqrt{a^2 + b^2 + c^2 + d^2}
```

The L2 norm of a quaternion vector is computed as: .. math:

```
\|x\|^2 = \sum_{i=1}^d |x_i|^2
        = \sum_{i=1}^d (x_i.re^2 + x_i.im_1^2 + x_i.im_2^2 + x_i.im_3^2)
```

Parameters

x (*torch.FloatTensor*) – The vector containing quaternion values.

Returns

The normalized vector.

Return type

torch.FloatTensor

Notes

This function normalizes the length of relation vectors represented as quaternions. It ensures that the absolute value of each quaternion in the vector is equal to 1, preserving the unit length.

score (*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Compute scores for a batch of triples using octonion-based embeddings.

This method computes scores for a batch of triples using octonion-based embeddings of head entities, relation embeddings, and tail entities. It supports both explicit and non-explicit scoring methods.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – Tensor containing the octonion-based embeddings of head entities.
- **rel_ent_emb** (*torch.FloatTensor*) – Tensor containing the octonion-based embeddings of relations.
- **tail_ent_emb** (*torch.FloatTensor*) – Tensor containing the octonion-based embeddings of tail entities.

Returns

Scores for the given batch of triples.

Return type

torch.FloatTensor

Notes

If no normalization is set, this method applies quaternion normalization to relation embeddings.

If the scoring method is explicit, it computes the scores using quaternion multiplication followed by an inner product of the real and imaginary parts of the resulting quaternions.

If the scoring method is non-explicit, it directly computes the inner product of the real and imaginary parts of the octonion-based embeddings.

k_vs_all_score (*bpe_head_ent_emb: torch.FloatTensor*, *bpe_rel_ent_emb: torch.FloatTensor*, *E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using quaternion embeddings for a batch of head entities and relations.

This method involves splitting the head entity and relation embeddings into quaternion components, optionally normalizing the relation embeddings, performing quaternion multiplication, and then calculating the score by performing an inner product with all tail entity embeddings.

Parameters

- **bpe_head_ent_emb** (*torch.FloatTensor*) – Batched embeddings of head entities, each represented as a quaternion.
- **bpe_rel_ent_emb** (*torch.FloatTensor*) – Batched embeddings of relations, each represented as a quaternion.
- **E** (*torch.FloatTensor*) – Embeddings of all possible tail entities.

Returns

Scores for all possible triples formed with the given head entities and relations against all entities. The shape of the output is (size of batch, number of entities).

Return type
torch.FloatTensor

Notes

The method is particularly useful in scenarios like link prediction, where the goal is to rank all possible tail entities for a given head entity and relation. Quaternion algebra is used to enhance the interaction modeling between entities and relations.

forward_k_vs_all(*x: torch.FloatTensor*) → torch.FloatTensor

Computes scores for all entities in a K-vs-All setting given a batch of head entities and relations.

This method retrieves embeddings for the head entities and relations from the input tensor *x*, applies necessary dropout and normalization, and then uses the *k_vs_all_score* method to compute the scores against all possible tail entities in the knowledge graph.

Parameters
x (*torch.FloatTensor*) – A tensor containing indices for head entities and relations. The tensor is expected to have a specific format suitable for the model's embedding retrieval process.

Returns
A tensor of scores, where each row corresponds to the scores of all tail entities for a single head entity and relation pair. The shape of the tensor is (size of the batch, number of entities).

Return type
torch.FloatTensor

Notes

This method is typically used in evaluating the model's performance in link prediction tasks, where it's important to rank the likelihood of every possible tail entity for a given head entity and relation.

forward_k_vs_sample(*x: torch.FloatTensor*, *target_entity_idx: int*) → torch.FloatTensor

Computes scores for a batch of triples against a sampled subset of entities in a K-vs-Sample setting.

Given a batch of head entities and relations (h,r), this method computes the scores for all possible triples formed with these head entities and relations against a subset of entities, i.e., [score(h,r,x)|x in Entities] => [0.0,0.1,…,0.8], shape=> (1, |Entities|). TODO: Add mathematical format for sphinx. The subset of entities is specified by the *target_entity_idx*, which is an integer index representing a specific entity. Given a batch of head entities and relations => shape (size of batch,| Entities|).

Parameters

- **x** (*torch.FloatTensor*) – A tensor containing indices for head entities and relations. The tensor is expected to have a specific format suitable for the model's embedding retrieval process.
- **target_entity_idx** (*int*) – Index of the target entity against which the scores are to be computed.

Returns
A tensor of scores where each element corresponds to the score of the target entity for a single head entity and relation pair. The shape of the tensor is (size of the batch, 1).

Return type
torch.FloatTensor

Notes

This method is particularly useful in scenarios like link prediction, where it's necessary to evaluate the likelihood of a specific relationship between a given head entity and a particular target entity.

class dicee.**OMult**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

OMult extends the base knowledge graph embedding model by integrating octonion algebra. This model leverages the properties of octonions to represent and process the embeddings of entities and relations in a knowledge graph, aiming to capture complex interactions and patterns.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions and learning rate.

name

The name identifier for the OMult model.

Type

str

octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, ..., *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, ...]

Normalizes octonion components to unit length.

score(*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes the score of a triple using octonion multiplication.

k_vs_all_score(*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*) → torch.FloatTensor

Computes scores in a K-vs-All setting using octonion embeddings.

forward_k_vs_all(*x*) → torch.FloatTensor

Performs a forward pass for K-vs-All scoring, returning scores for all entities.

static octonion_normalizer(*emb_rel_e0: torch.Tensor*, *emb_rel_e1: torch.Tensor*, *emb_rel_e2: torch.Tensor*, *emb_rel_e3: torch.Tensor*, *emb_rel_e4: torch.Tensor*, *emb_rel_e5: torch.Tensor*, *emb_rel_e6: torch.Tensor*, *emb_rel_e7: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tens

Normalizes the components of an octonion.

Each component of the octonion is divided by the square root of the sum of the squares of all components, normalizing it to unit length.

Parameters

- **emb_rel_e0** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e1** (*torch.Tensor*) – The eight components of an octonion.
- **...** (*torch.Tensor*) – The eight components of an octonion.
- **emb_rel_e7** (*torch.Tensor*) – The eight components of an octonion.

Returns

The normalized components of the octonion.

Return type

Tuple[torch.Tensor, …]

score (*head_ent_emb: torch.FloatTensor*, *rel_ent_emb: torch.FloatTensor*, *tail_ent_emb: torch.FloatTensor*) → torch.FloatTensor

Computes the score of a triple using octonion multiplication.

The method involves splitting the embeddings into real and imaginary parts, normalizing the relation embeddings, performing octonion multiplication, and then calculating the score based on the inner product.

Parameters

- **head_ent_emb** (*torch.FloatTensor*) – Embedding of the head entity.
- **rel_ent_emb** (*torch.FloatTensor*) – Embedding of the relation.
- **tail_ent_emb** (*torch.FloatTensor*) – Embedding of the tail entity.

Returns

The score of the triple.

Return type

torch.FloatTensor

k_vs_all_score (*bpe_head_ent_emb: torch.FloatTensor*, *bpe_rel_ent_emb: torch.FloatTensor*, *E: torch.FloatTensor*) → torch.FloatTensor

Computes scores in a K-vs-All setting using octonion embeddings for a batch of head entities and relations.

This method splits the head entity and relation embeddings into their octonion components, normalizes the relation embeddings if necessary, and then applies octonion multiplication. It computes the score by performing an inner product with all tail entity embeddings.

Parameters

- **bpe_head_ent_emb** (*torch.FloatTensor*) – Batched embeddings of head entities, each represented as an octonion.
- **bpe_rel_ent_emb** (*torch.FloatTensor*) – Batched embeddings of relations, each represented as an octonion.
- **E** (*torch.FloatTensor*) – Embeddings of all possible tail entities.

Returns

Scores for all possible triples formed with the given head entities and relations against all entities. The shape of the output is (size of batch, number of entities).

Return type

torch.FloatTensor

Notes

The method is particularly useful in scenarios like link prediction, where the goal is to rank all possible tail entities for a given head entity and relation.

forward_k_vs_all (*x*)

Performs a forward pass for K-vs-All scoring.

TODO: Add mathematical format for sphinx.

Given a head entity and a relation (h,r), this method computes scores for all possible triples, i.e., [score(h,r,x)|x in Entities] => [0.0,0.1,…,0.8], shape=> (1, **|Entities|**), returning a score for each entity in the knowledge graph.

Parameters

x (*Tensor*) – Tensor containing indices for head entities and relations.

Returns
Scores for all triples formed with the given head entities and relations against all entities.

Return type
torch.FloatTensor

class dicee.**Shallom**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

Shallom is a shallow neural model designed for relation prediction in knowledge graphs. The model combines entity embeddings and passes them through a neural network to predict the likelihood of different relations. It's based on the paper: 'A Shallow Neural Model for Relation Prediction' (https://arxiv.org/abs/2101.09090).

name

The name identifier for the Shallom model.

Type
str

shallom

A sequential neural network model used for predicting relations.

Type
torch.nn.Sequential

get_embeddings() → Tuple[np.ndarray, None]

Retrieves the entity embeddings.

forward_k_vs_all(*x*) → torch.FloatTensor

Computes relation scores for all pairs of entities in the batch.

forward_triples(*x*) → torch.FloatTensor

Computes relation scores for a batch of triples.

get_embeddings() → Tuple[numpy.ndarray, None]

Retrieves the entity embeddings from the model.

Returns
A tuple containing the entity embeddings as a NumPy array and None for the relation embeddings.

Return type
Tuple[np.ndarray, None]

forward_k_vs_all(*x: torch.Tensor*) → torch.FloatTensor

Computes relation scores for all pairs of entities in the batch.

Each pair of entities is passed through the Shallom neural network to predict the likelihood of various relations between them.

Parameters
x (*torch.Tensor*) – A tensor of entity pairs.

Returns
A tensor of relation scores for each pair of entities in the batch.

Return type
torch.FloatTensor

forward_triples(*x: torch.Tensor*) → torch.FloatTensor

Computes relation scores for a batch of triples.

This method first computes relation scores for all possible relations for each pair of entities and then selects the scores corresponding to the actual relations in the triples.

Parameters
x (*torch.Tensor*) – A tensor containing a batch of triples.

Returns
A flattened tensor of relation scores for the given batch of triples.

Return type
torch.FloatTensor

class dicee.**LFMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Embedding with polynomial functions. We represent all entities and relations in the polynomial space as: f(x) = sum_{i=0}^{d-1} a_k x^{i%d} and use the three differents scoring function as in the paper to evaluate the score. We also consider combining with Neural Networks.

forward_triples(*idx_triple*)

Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

construct_multi_coeff(*x*)

poly_NN(*x*, *coefh*, *coefr*, *coeft*)

Constructing a 2 layers NN to represent the embeddings. h = sigma(wh^T x + bh), r = sigma(wr^T x + br), t = sigma(wt^T x + bt)

linear(*x*, *w*, *b*)

scalar_batch_NN(*a*, *b*, *c*)

element wise multiplication between a,b and c: Inputs : a, b, c ====> torch.tensor of size batch_size x m x d Output : a tensor of size batch_size x d

tri_score(*coeff_h*, *coeff_r*, *coeff_t*)

this part implement the trilinear scoring techniques:

score(h,r,t) = int_{0}{1} h(x)r(x)t(x) dx = sum_{i,j,k = 0}^{d-1} dfrac{a_i*b_j*c_k}{1+(i+j+k)%d}

1. generate the range for i,j and k from [0 d-1]
2. perform dfrac{a_i*b_j*c_k}{1+(i+j+k)%d} in parallel for every batch
3. take the sum over each batch

vtp_score(*h*, *r*, *t*)

this part implement the vector triple product scoring techniques:

score(h,r,t) = int_{0}{1} h(x)r(x)t(x) dx = sum_{i,j,k = 0}^{d-1} dfrac{a_i*c_j*b_k - b_i*c_j*a_k}{(1+(i+j)%d)(1+k)}

1. generate the range for i,j and k from [0 d-1]
2. Compute the first and second terms of the sum
3. Multiply with then denominator and take the sum
4. take the sum over each batch

comp_func(*h*, *r*, *t*)

this part implement the function composition scoring techniques: i.e. score = <hor, t>

polynomial(*coeff*, *x*, *degree*)

This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,…d] and return a vector tensor (coeff[0][0] + coeff[0][1]x +…+ coeff[0][d]x^d,

coeff[1][0] + coeff[1][1]x +…+ coeff[1][d]x^d)

pop(*coeff*, *x*, *degree*)

This function allow us to evaluate the composition of two polynomes without for loops :) it takes a matrix tensor of coefficients (coeff), a matrix tensor of points x and range of integer [0,1,…d]

and return a tensor (coeff[0][0] + coeff[0][1]x +…+ coeff[0][d]x^d,

coeff[1][0] + coeff[1][1]x +…+ coeff[1][d]x^d)

class dicee.**PykeenKGE**(*args: dict*)

Bases: *dicee.models.base_model.BaseKGE*

A class for using knowledge graph embedding models implemented in Pykeen.

Parameters

args (*dict*) – A dictionary of arguments containing hyperparameters and settings for the model, such as embedding dimensions, random seed, and model-specific kwargs.

name

The name identifier for the PykeenKGE model.

Type

str

model

The Pykeen model instance.

Type

pykeen.models.base.Model

loss_history

A list to store the training loss history.

Type

list

args

The arguments used to initialize the model.

Type

dict

entity_embeddings

Entity embeddings learned by the model.

Type

torch.nn.Embedding

relation_embeddings

Relation embeddings learned by the model.

Type
torch.nn.Embedding

interaction

Interaction module used by the Pykeen model.

Type
pykeen.nn.modules.Interaction

forward_k_vs_all (*x: torch.LongTensor*) → torch.FloatTensor

Compute scores for all entities given a batch of head entities and relations.

forward_triples (*x: torch.LongTensor*) → torch.FloatTensor

Compute scores for a batch of triples.

forward_k_vs_sample (*x: torch.LongTensor*, *target_entity_idx: int*)

Compute scores against a sampled subset of entities.

Notes

This class provides an interface for using knowledge graph embedding models implemented in Pykeen. It initializes Pykeen models based on the provided arguments and allows for scoring triples and conducting knowledge graph embedding experiments.

forward_k_vs_all (*x: torch.LongTensor*)

TODO: Format in Numpy-style documentation

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads and relations + apply Dropout & Normalization if given. h, r = self.get_head_relation_representation(x) # (2) Reshape (1). if self.last_dim > 0:

h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim)

(3) Reshape all entities. if self.last_dim > 0:

t = self.entity_embeddings.weight.reshape(self.num_entities, self.embedding_dim, self.last_dim)

else:
t = self.entity_embeddings.weight

(4) Call the score_t from interactions to generate triple scores. return self.interaction.score_t(h=h, r=r, all_entities=t, slice_size=1)

forward_triples (*x: torch.LongTensor*) → torch.FloatTensor

TODO: Format in Numpy-style documentation

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads, relations and tails and apply Dropout & Normalization if given. h, r, t = self.get_triple_representation(x) # (2) Reshape (1). if self.last_dim > 0:

h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim) t = t.reshape(len(x), self.embedding_dim, self.last_dim)

(3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice_size=None, slice_dim=0)

abstract forward_k_vs_sample(*x: torch.LongTensor*, *target_entity_idx: int*)

Forward pass for K vs. Sample.

Raises

ValueError – This function is not implemented in the current model.

class dicee.**BytE**(**args*, ***kwargs*)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

loss_function(*yhat_batch*, *y_batch*)

Parameters

- **yhat_batch** –
- **y_batch** –

forward(*x: torch.LongTensor*)

Parameters

x (*B by T tensor*) –

generate(*idx*, *max_new_tokens*, *temperature=1.0*, *top_k=None*)

Take a conditioning sequence of indices idx (LongTensor of shape (b,t)) and complete the sequence max_new_tokens times, feeding the predictions back into the model each time. Most likely you'll want to make sure to be in model.eval() mode of operation for this.

training_step(*batch*, *batch_idx=None*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** – The output of your data iterable, normally a `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple dataloaders used)

Returns

- `Tensor` - The loss tensor
- `dict` - A dictionary which can include any keys, but must include the key `'loss'` in the case of automatic optimization.
- `None` - In automatic optimization, this will skip to the next batch (but is not supported for multi-GPU, TPU, or DeepSpeed). For manual optimization, this has no special meaning, as returning the loss is not required.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

To use multiple optimizers, you can switch to 'manual optimization' and control their stepping:

```
def __init__(self):
    super().__init__()
    self.automatic_optimization = False


# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

    # do training_step with encoder
    ...
    opt1.step()
    # do training_step with decoder
    ...
    opt2.step()
```

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

class dicee.**BaseKGE**(*args: dict*)

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_byte_pair_encoded_k_vs_all(*x: torch.LongTensor*)

Parameters

x (*B x 2 x T*) –

forward_byte_pair_encoded_triple(*x: Tuple[torch.LongTensor, torch.LongTensor]*)

Perform the forward pass for byte pair encoded triples.

Parameters

x (*Tuple[torch.LongTensor, torch.LongTensor]*) – The input tuple containing byte pair encoded entities and relations.

Returns

The output tensor containing the scores for the byte pair encoded triples.

Return type

torch.Tensor

init_params_with_sanity_checking()

forward(*x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor]*, *y_idx: torch.LongTensor = None*)

Perform the forward pass of the model.

Parameters

- **x** (*Union[torch.LongTensor, Tuple[torch.LongTensor, torch.LongTensor]]*) – The input tensor or a tuple containing the input tensor and target entity indexes.

- **y_idx** (*torch.LongTensor, optional*) – The target entity indexes (default is None).

Returns
The output of the forward pass.

Return type
Any

forward_triples(*x: torch.LongTensor*) → torch.Tensor
Perform the forward pass for triples.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The output tensor containing the scores for the input triples.

Return type
torch.Tensor

forward_k_vs_all(**args*, ***kwargs*)
Forward pass for K vs. All.

Raises
ValueError – This function is not implemented in the current model.

forward_k_vs_sample(**args*, ***kwargs*)
Forward pass for K vs. Sample.

Raises
ValueError – This function is not implemented in the current model.

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor]
Get the representation for the head and relation entities.

Parameters
indexed_triple (*torch.LongTensor*) – The indexes of the head and relation entities.

Returns
The representation for the head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_sentence_representation(*x: torch.LongTensor*)
→ Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]
Get the representation for a sentence.

Parameters
x (*torch.LongTensor*) – The input tensor containing the indexes of head, relation, and tail entities.

Returns
The representation for the input sentence.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

get_bpe_head_and_relation_representation(*x: torch.LongTensor*) → Tuple[torch.FloatTensor, torch.FloatTensor]

Get the representation for BPE head and relation entities.

Parameters
x (*B x 2 x T*) –

Returns
The representation for BPE head and relation entities.

Return type
Tuple[torch.FloatTensor, torch.FloatTensor]

get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]

Get the entity and relation embeddings.

Returns
The entity and relation embeddings.

Return type
Tuple[np.ndarray, np.ndarray]

dicee.**create_recipriocal_triples**(*x*)

Add inverse triples into dask dataframe :param x: :return:

dicee.**get_er_vocab**(*data*, *file_path: str = None*)

dicee.**get_re_vocab**(*data*, *file_path: str = None*)

dicee.**get_ee_vocab**(*data*, *file_path: str = None*)

dicee.**timeit**(*func*)

dicee.**save_pickle**(*, *data: object = None*, *file_path=str*)

dicee.**load_pickle**(*file_path=str*)

dicee.**select_model**(*args: dict*, *is_continual_training: bool = None*, *storage_path: str = None*)

dicee.**load_model**(*path_of_experiment_folder: str*, *model_name='model.pt'*, *verbose=0*) → Tuple[object, Tuple[dict, dict]]

Load weights and initialize pytorch module from namespace arguments

dicee.**load_model_ensemble**(*path_of_experiment_folder: str*) → Tuple[*dicee.models.base_model.BaseKGE*, Tuple[pandas.DataFrame, pandas.DataFrame]]

Construct Ensemble Of weights and initialize pytorch module from namespace arguments

(1) Detect models under given path

(2) Accumulate parameters of detected models

(3) Normalize parameters

(4) Insert (3) into model.

dicee.**save_numpy_ndarray**(*, *data: numpy.ndarray*, *file_path: str*)

dicee.**numpy_data_type_changer**(*train_set: numpy.ndarray*, *num: int*) → numpy.ndarray

Detect most efficient data type for a given triples :param train_set: :param num: :return:

dicee.**save_checkpoint_model**(*model*, *path: str*) → None

Store Pytorch model into disk

dicee.**store**(*trainer*, *trained_model*, *model_name: str = 'model'*, *full_storage_path: str = None*, *save_embeddings_as_csv=False*) → None

Store trained_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param full_storage_path: path to save parameters. :param model_name: string representation of the name of the model. :param trained_model: an instance of BaseKGE see core.models.base_model . :param save_embeddings_as_csv: for easy access of embeddings. :return:

dicee.**add_noisy_triples**(*train_set: pandas.DataFrame*, *add_noise_rate: float*) → pandas.DataFrame

Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

dicee.**read_or_load_kg**(*args*, *cls*)

dicee.**intialize_model**(*args: dict*, *verbose=0*) → Tuple[object, str]

dicee.**load_json**(*p: str*) → dict

dicee.**save_embeddings**(*embeddings: numpy.ndarray*, *indexes*, *path: str*) → None

Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:

dicee.**random_prediction**(*pre_trained_kge*)

dicee.**deploy_triple_prediction**(*pre_trained_kge*, *str_subject*, *str_predicate*, *str_object*)

dicee.**deploy_tail_entity_prediction**(*pre_trained_kge*, *str_subject*, *str_predicate*, *top_k*)

dicee.**deploy_head_entity_prediction**(*pre_trained_kge*, *str_object*, *str_predicate*, *top_k*)

dicee.**deploy_relation_prediction**(*pre_trained_kge*, *str_subject*, *str_object*, *top_k*)

dicee.**vocab_to_parquet**(*vocab_to_idx*, *name*, *path_for_serialization*, *print_into*)

dicee.**create_experiment_folder**(*folder_name='Experiments'*)

dicee.**continual_training_setup_executor**(*executor*) → None

storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data

full_storage_path:str A path leading to a subdirectory containing KGE related data

dicee.**exponential_function**(*x: numpy.ndarray*, *lam: float*, *ascending_order=True*) → torch.FloatTensor

dicee.**load_numpy**(*path*) → numpy.ndarray

dicee.**evaluate**(*entity_to_idx*, *scores*, *easy_answers*, *hard_answers*)

@TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

dicee.**download_file**(*url*, *destination_folder='.'*)

dicee.**download_files_from_url**(*base_url: str*, *destination_folder='.'*) → None

Parameters

- **base_url** (e.g. "https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll") –
- **destination_folder**(*e.g. "KINSHIP-Keci-dim128-epoch256-KvsAll"*) –

dicee.**download_pretrained_model**(*url: str*) → str

class dicee.**DICE_Trainer**(*args*, *is_continual_training: bool*, *storage_path: str*, *evaluator: object | None = None*)

Implements a training framework for knowledge graph embedding models using [PyTorch Lightning](https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html), supporting [multi-GPU](https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html) and CPU training. This trainer can handle continual training scenarios and supports different forms of labeling and evaluation methods.

Parameters

- **args** (*Namespace*) – Command line arguments or configurations specifying training parameters and model settings.
- **is_continual_training** (*bool*) – Flag indicating whether the training session is part of a continual learning process.
- **storage_path** (*str*) – Path to the directory where training checkpoints and models are stored.
- **evaluator** (*object, optional*) – An evaluation object responsible for model evaluation. This can be any object that implements an *eval* method accepting model predictions and returning evaluation metrics.

report

A dictionary to store training reports and metrics.

Type
dict

trainer

The PyTorch Lightning Trainer instance used for model training.

Type
lightening.Trainer or None

form_of_labelling

The form of labeling used during training, which can be "EntityPrediction", "RelationPrediction", or "Pyke".

Type
str or None

continual_start()

Initializes and starts the training process, including model loading and fitting.

initialize_trainer(*callbacks: List*) → lightening.Trainer

Initializes a PyTorch Lightning Trainer instance with the specified callbacks.

initialize_or_load_model()

Initializes or loads a model for training based on the training configuration.

initialize_dataloader(*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

Initializes a DataLoader for the given dataset.

initialize_dataset(*dataset: KG*, *form_of_labelling*) → torch.utils.data.Dataset

Prepares and initializes a dataset for training.

start(*knowledge_graph: KG*) → Tuple[*BaseKGE*, str]

Starts the training process for a given knowledge graph.

k_fold_cross_validation(*dataset*) → Tuple[*BaseKGE*, str]

Performs K-fold cross-validation on the dataset and returns the trained model and form of labelling.

continual_start()

Initializes and starts the training process, including model loading and fitting. This method is specifically designed for continual training scenarios.

Returns

- **model** (*BaseKGE*) – The trained knowledge graph embedding model instance. *BaseKGE* is a placeholder for the actual model class, which should be a subclass of the base model class used in your framework.
- **form_of_labelling** (*str*) – The form of labeling used during the training. This can indicate the type of prediction task the model is trained for, such as "EntityPrediction", "RelationPrediction", or other custom labeling forms defined in your implementation.

initialize_trainer(*callbacks: List*) → lightning.Trainer

Initializes a PyTorch Lightning Trainer instance.

Parameters

callbacks (*List*) – A list of PyTorch Lightning callbacks to be used during training.

Returns

The initialized PyTorch Lightning Trainer instance.

Return type

pl.Trainer

initialize_or_load_model() → Tuple[*dicee.models.base_model.BaseKGE*, str]

Initializes or loads a knowledge graph embedding model based on the training configuration. This method decides whether to start training from scratch or to continue training from a previously saved model state, depending on the *is_continual_training* attribute.

Returns

- **model** (*BaseKGE*) – The model instance that is either initialized from scratch or loaded from a saved state. *BaseKGE* is a generic placeholder for the actual model class, which is a subclass of the base knowledge graph embedding model class used in your implementation.
- **form_of_labelling** (*str*) – A string indicating the type of prediction task the model is configured for. Possible values include "EntityPrediction" and "RelationPrediction", which signify whether the model is trained to predict missing entities or relations in a knowledge graph. The actual values depend on the specific tasks supported by your implementation.

Notes

The method uses the *is_continual_training* attribute to determine if the model should be loaded from a saved state. If *is_continual_training* is True, the method attempts to load the model and its configuration from the specified *storage_path*. If *is_continual_training* is False or the model cannot be loaded, a new model instance is initialized.

This method also sets the *form_of_labelling* attribute based on the model's configuration, which is used to inform downstream training and evaluation processes about the type of prediction task.

initialize_dataloader(*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

Initializes and returns a PyTorch DataLoader object for the given dataset.

This DataLoader is configured based on the training arguments provided, including batch size, shuffle status, and the number of workers.

Parameters
dataset (*torch.utils.data.Dataset*) – The dataset to be loaded into the DataLoader. This dataset should already be processed and ready for training or evaluation.

Returns
A DataLoader instance ready for training or evaluation, configured with the appropriate batch size, shuffle setting, and number of workers.

Return type
torch.utils.data.DataLoader

initialize_dataset(*dataset: dicee.knowledge_graph.KG*, *form_of_labelling: str*) → torch.utils.data.Dataset

Initializes and returns a dataset suitable for training or evaluation, based on the knowledge graph data and the specified form of labelling.

Parameters

- **dataset** (KG) – The knowledge graph data used to construct the dataset. This should include training, validation, and test sets along with any other necessary information like entity and relation mappings.
- **form_of_labelling** (*str*) – The form of labelling to be used for the dataset, indicating the prediction task (e.g., "EntityPrediction", "RelationPrediction").

Returns
A processed dataset ready for use with a PyTorch DataLoader, tailored to the specified form of labelling and containing all necessary data for training or evaluation.

Return type
torch.utils.data.Dataset

start(*knowledge_graph: dicee.knowledge_graph.KG*) → Tuple[*dicee.models.base_model.BaseKGE*, str]

Starts the training process for the selected model using the provided knowledge graph data. The method selects and trains the model based on the configuration specified in the arguments.

Parameters
knowledge_graph (KG) – The knowledge graph data containing entities, relations, and triples, which will be used for training the model.

Returns
A tuple containing the trained model instance and the form of labelling used during training. The form of labelling indicates the type of prediction task.

Return type
Tuple[*BaseKGE*, str]

k_fold_cross_validation(*dataset: dicee.knowledge_graph.KG*) → Tuple[*dicee.models.base_model.BaseKGE*, str]

Conducts K-fold cross-validation on the provided dataset to assess the performance of the model specified in the training arguments. The process involves partitioning the dataset into K distinct subsets, iteratively using one subset for testing and the remainder for training. The model's performance is evaluated on each test split to compute the Mean Reciprocal Rank (MRR) scores.

Steps: 1. The dataset is divided into K train and test splits. 2. For each split: 2.1. A trainer and model are initialized based on the provided configuration. 2.2. The model is trained using the training portion of the split. 2.3. The MRR score of the trained model is computed using the test portion of the split. 3. The process aggregates the MRR scores across all splits to report the mean and standard deviation of the MRR, providing a comprehensive evaluation of the model's performance.

Parameters

dataset (KG) – The dataset to be used for K-fold cross-validation. This dataset should include the triples (head entity, relation, tail entity) for the entire knowledge graph.

Returns

A tuple containing: - The trained model instance from the last fold of the cross-validation. - The form of labelling used during training, indicating the prediction task (e.g., "EntityPrediction", "RelationPrediction").

Return type

Tuple[*BaseKGE*, str]

Notes

The function assumes the presence of a predefined number of folds (K) specified in the training arguments. It utilizes PyTorch Lightning for model training and evaluation, leveraging GPU acceleration if available. The final output includes the model trained on the last fold and a summary of the cross-validation performance metrics.

class dicee.**KGE**(*path=None*, *url=None*, *construct_ensemble=False*, *model_name=None*, *apply_semantic_constraint=False*)

Bases: *dicee.abstracts.BaseInteractiveKGE*

Knowledge Graph Embedding Class for interactive usage of pre-trained models

get_transductive_entity_embeddings(*indices: torch.LongTensor | List[str]*, *as_pytorch=False*, *as_numpy=False*, *as_list=True*)
→ torch.FloatTensor | numpy.ndarray | List[float]

create_vector_database(*collection_name: str*, *distance: str*, *location: str = 'localhost'*, *port: int = 6333*)

generate(*h=''*, *r=''*)

__str__()

Return str(self).

eval_lp_performance(*dataset=List[Tuple[str, str, str]]*, *filtered=True*)

predict_missing_head_entity(*relation: List[str] | str*, *tail_entity: List[str] | str*, *within=None*)
→ Tuple

Given a relation and a tail entity, return top k ranked head entity.

argmax_{e in E } f(e,r,t), where r in R, t in E.

Parameter

relation: Union[List[str], str]

String representation of selected relations.

tail_entity: Union[List[str], str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_relations(*head_entity: List[str] | str*, *tail_entity: List[str] | str*, *within=None*) → Tuple

Given a head entity and a tail entity, return top k ranked relations.

argmax_{r in R } f(h,r,t), where h, t in E.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_tail_entity(*head_entity: List[str] | str*, *relation: List[str] | str*, *within: List[str] = None*) → torch.FloatTensor

Given a head entity and a relation, return top k ranked entities

argmax_{e in E } f(h,r,e), where h in E and r in R.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

scores

predict(*, *h: List[str] | str = None*, *r: List[str] | str = None*, *t: List[str] | str = None*, *within=None*, *logits=True*) → torch.FloatTensor

Parameters

- **logits** –
- **h** –

- **r** –
- **t** –
- **within** –

predict_topk(*, *h: List[str] = None*, *r: List[str] = None*, *t: List[str] = None*, *topk: int = 10*, *within: List[str] = None*)

Predict missing item in a given triple.

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k item.

Returns: Tuple

Highest K scores and items

triple_score(*h: List[str] | str = None*, *r: List[str] | str = None*, *t: List[str] | str = None*, *logits=False*) → torch.FloatTensor

Predict triple score

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

Returns: Tuple

pytorch tensor of triple score

t_norm (*tens_1: torch.Tensor*, *tens_2: torch.Tensor*, *tnorm: str = 'min'*) → torch.Tensor

tensor_t_norm (*subquery_scores: torch.FloatTensor*, *tnorm: str = 'min'*) → torch.FloatTensor

Compute T-norm over [0,1] ^{n imes d} where n denotes the number of hops and d denotes number of entities

t_conorm (*tens_1: torch.Tensor*, *tens_2: torch.Tensor*, *tconorm: str = 'min'*) → torch.Tensor

negnorm (*tens_1: torch.Tensor*, *lambda_: float*, *neg_norm: str = 'standard'*) → torch.Tensor

return_multi_hop_query_results (*aggregated_query_for_all_entities*, *k: int*, *only_scores*)

single_hop_query_answering (*query: tuple*, *only_scores: bool = True*, *k: int = None*)

answer_multi_hop_query (*query_type: str = None*,
query: Tuple[str | Tuple[str, str], Ellipsis] = None,
queries: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, *tnorm: str = 'prod'*,
neg_norm: str = 'standard', *lambda_: float = 0.0*, *k: int = 10*, *only_scores=False*)
→ List[Tuple[str, torch.Tensor]]

@TODO: Refactoring is needed # @TODO: Score computation for each query type should be done in a static function

Find an answer set for EPFO queries including negation and disjunction

Parameter

query_type: str The type of the query, e.g., “2p”.

query: Union[str, Tuple[str, Tuple[str, str]]] The query itself, either a string or a nested tuple.

queries: List of Tuple[Union[str, Tuple[str, str]], …]

tnorm: str The t-norm operator.

neg_norm: str The negation norm.

lambda_: float lambda parameter for sugeno and yager negation norms

k: int The top-k substitutions for intermediate variables.

returns

- *List[Tuple[str, torch.Tensor]]*
- *Entities and corresponding scores sorted in the descening order of scores*

find_missing_triples (*confidence: float*, *entities: List[str] = None*, *relations: List[str] = None*,
topk: int = 10, *at_most: int = sys.maxsize*) → Set

Find missing triples

Iterative over a set of entities E and a set of relation R :

orall e in E and orall r in R f(e,r,x)

Return (e,r,x)

otin G and f(e,r,x) > confidence

confidence: float

A threshold for an output of a sigmoid function given a triple.

topk: int

Highest ranked k item to select triples with f(e,r,x) > confidence .

at_most: int

Stop after finding at_most missing triples

{(e,r,x) | f(e,r,x) > confidence land (e,r,x)

otin G

deploy(*share: bool = False*, *top_k: int = 10*)

train_triples(*h: List[str]*, *r: List[str]*, *t: List[str]*, *labels: List[float]*, *iteration=2*, *optimizer=None*)

train_k_vs_all(*h*, *r*, *iteration=1*, *lr=0.001*)

Train k vs all :param head_entity: :param relation: :param iteration: :param lr: :return:

train(*kg*, *lr=0.1*, *epoch=10*, *batch_size=32*, *neg_sample_ratio=10*, *num_workers=1*) → None

Retrained a pretrain model on an input KG via negative sampling.

class dicee.**Execute**(*args*, *continuous_training=False*)

A class for Training, Retraining and Evaluation a model.

(1) Loading & Preprocessing & Serializing input data.

(2) Training & Validation & Testing

(3) Storing all necessary info

read_or_load_kg()

read_preprocess_index_serialize_data() → None

Read & Preprocess & Index & Serialize Input Data

(1) Read or load the data from disk into memory.

(2) Store the statistics of the data.

Parameter

rtype
None

load_indexed_data() → None

Load the indexed data from disk into memory

Parameter

rtype
None

save_trained_model() → None
Save a knowledge graph embedding model

(1) Send model to eval mode and cpu.

(2) Store the memory footprint of the model.

(3) Save the model into disk.

(4) Update the stats of KG again ?

Parameter

rtype
None

end(*form_of_labelling: str*) → dict
End training

(1) Store trained model.

(2) Report runtimes.

(3) Eval model if required.

Parameter

rtype
A dict containing information about the training and/or evaluation

write_report() → None
Report training related information in a report.json file

start() → dict
Start training

(1) Loading the Data # (2) Create an evaluator object. # (3) Create a trainer object. # (4) Start the training

Parameter

rtype
A dict containing information about the training and/or evaluation

dicee.**mapping_from_first_two_cols_to_third**(*train_set_idx*)

dicee.**timeit**(*func*)

dicee.**load_pickle**(*file_path=str*)

dicee.**reload_dataset**(*path: str*, *form_of_labelling: str*, *scoring_technique: str*, *neg_ratio: float*, *label_smoothing_rate: float*) → torch.utils.data.Dataset

Reloads the dataset from disk and constructs a PyTorch dataset for training.

Parameters

- **path** (*str*) – The path to the directory where the dataset is stored.
- **form_of_labelling** (*str*) – The form of labelling used in the dataset. Determines how data points are represented.
- **scoring_technique** (*str*) – The scoring technique used for evaluating the embeddings.
- **neg_ratio** (*float*) – The ratio of negative samples to positive samples in the dataset.
- **label_smoothing_rate** (*float*) – The rate of label smoothing applied to the dataset.

Returns

A PyTorch dataset object ready for training.

Return type

torch.utils.data.Dataset

dicee.**construct_dataset**(*, *train_set: numpy.ndarray | list*, *valid_set=None*, *test_set=None*, *ordered_bpe_entities=None*, *train_target_indices=None*, *target_dim: int = None*, *entity_to_idx: dict*, *relation_to_idx: dict*, *form_of_labelling: str*, *scoring_technique: str*, *neg_ratio: int*, *label_smoothing_rate: float*, *byte_pair_encoding=None*, *block_size: int = None*) → torch.utils.data.Dataset

Constructs a dataset based on the specified parameters and returns a PyTorch Dataset object.

Parameters

- **train_set** (*Union[np.ndarray, list]*) – The training set consisting of triples or tokens.
- **valid_set** (*Optional*) – The validation set. Not currently used in dataset construction.
- **test_set** (*Optional*) – The test set. Not currently used in dataset construction.
- **ordered_bpe_entities** (*Optional*) – Ordered byte pair encoding entities for the dataset.
- **train_target_indices** (*Optional*) – Indices of target entities or relations for training.
- **target_dim** (*int, optional*) – The dimension of target entities or relations.
- **entity_to_idx** (*dict*) – A dictionary mapping entity strings to indices.
- **relation_to_idx** (*dict*) – A dictionary mapping relation strings to indices.
- **form_of_labelling** (*str*) – Specifies the form of labelling, such as 'EntityPrediction' or 'RelationPrediction'.
- **scoring_technique** (*str*) – The scoring technique used for generating negative samples or evaluating the model.
- **neg_ratio** (*int*) – The ratio of negative samples to positive samples.
- **label_smoothing_rate** (*float*) – The rate of label smoothing applied to labels.
- **byte_pair_encoding** (*Optional*) – Indicates if byte pair encoding is used.
- **block_size** (*int, optional*) – The block size for transformer-based models.

Returns
A PyTorch dataset object ready for model training.

Return type
torch.utils.data.Dataset

class dicee.**BPE_NegativeSamplingDataset**(*train_set: torch.LongTensor*, *ordered_shaped_bpe_entities: torch.LongTensor*, *neg_ratio: int*)

Bases: `torch.utils.data.Dataset`

A PyTorch Dataset for handling negative sampling with Byte Pair Encoding (BPE) entities.

This dataset extends the PyTorch Dataset class to provide functionality for negative sampling in the context of knowledge graph embeddings. It uses byte pair encoding for entities to handle large vocabularies efficiently.

Parameters

- **train_set** (*torch.LongTensor*) – A tensor containing the training set triples with byte pair encoded entities and relations. The shape of the tensor is [N, 3], where N is the number of triples.
- **ordered_shaped_bpe_entities** (*torch.LongTensor*) – A tensor containing the ordered and shaped byte pair encoded entities.
- **neg_ratio** (*int*) – The ratio of negative samples to generate per positive sample.

num_bpe_entities

The number of byte pair encoded entities.

Type
int

num_datapoints

The number of data points (triples) in the training set.

Type
int

__len__() → int

Returns the total number of data points in the dataset.

Returns
The number of data points.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves the BPE-encoded triple and its corresponding label at the specified index.

Parameters
idx (*int*) – Index of the triple to retrieve.

Returns

A tuple containing the following elements: - The BPE-encoded triple as a torch.Tensor of shape (3,). - The label for the triple, where positive examples have a label of 1 and negative examples have a label

of 0, as a torch.Tensor.

Return type
tuple

collate_fn(*batch_shaped_bpe_triples: List[Tuple[torch.Tensor, torch.Tensor]]*)
→ Tuple[torch.Tensor, torch.Tensor]

Collate function for the BPE_NegativeSamplingDataset. It processes a batch of byte pair encoded triples, performs negative sampling, and returns the batch along with corresponding labels.

This function is designed to be used with a PyTorch DataLoader. It takes a list of byte pair encoded triples as input and generates negative samples according to the specified negative sampling ratio. The function ensures that the negative samples are combined with the original triples to form a single batch, which is suitable for training a knowledge graph embedding model.

Parameters

batch_shaped_bpe_triples (*List[Tuple[torch.Tensor, torch.Tensor]]*) – A list of tuples, where each tuple contains byte pair encoded representations of head entities, relations, and tail entities for a batch of triples.

Returns

A tuple containing two elements: - The first element is a torch.Tensor of shape [N * (1 + neg_ratio), 3] that contains both the original byte pair encoded triples and the generated negative samples. N is the original number of triples in the batch, and neg_ratio is the negative sampling ratio. - The second element is a torch.Tensor of shape [N * (1 + neg_ratio)] that contains the labels for each triple in the batch. Positive samples are labeled as 1, and negative samples are labeled as 0.

Return type

Tuple[torch.Tensor, torch.Tensor]

class dicee.**MultiLabelDataset**(*train_set: torch.LongTensor*, *train_indices_target: torch.LongTensor*, *target_dim: int*, *torch_ordered_shaped_bpe_entities: torch.LongTensor*)

Bases: `torch.utils.data.Dataset`

A dataset class for multi-label knowledge graph embedding tasks. This dataset is designed for models where the output involves predicting multiple labels (entities or relations) for a given input (e.g., predicting all possible tail entities given a head entity and a relation).

Parameters

- **train_set** (*torch.LongTensor*) – A tensor containing the training set triples with byte pair encoding, shaped as [num_triples, 3], where each triple is [head, relation, tail].
- **train_indices_target** (*torch.LongTensor*) – A tensor where each row corresponds to the indices of the target labels for each training example. The length of this tensor must match the number of triples in *train_set*.
- **target_dim** (*int*) – The dimensionality of the target space, typically the total number of possible labels (entities or relations).
- **torch_ordered_shaped_bpe_entities** (*torch.LongTensor*) – A tensor containing ordered byte pair encoded entities used for creating embeddings. This tensor is not directly used in generating targets but may be utilized for additional processing or embedding lookup.

num_datapoints

The number of data points (triples) in the dataset.

Type

int

collate_fn

Optional custom collate function to be used with a PyTorch DataLoader. It's set to None by default and can be specified after initializing the dataset if needed.

Type
None or callable

Note: This dataset is particularly suited for KvsAll (K entities vs. All entities) and AllvsAll training strategies in knowledge graph embedding, where a model predicts a set of possible tail entities given a head entity and a relation (or vice versa), and where each training example can have multiple correct labels.

__len__() → int

Returns the total number of data points in the dataset.

Returns
The number of data points.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves the knowledge graph triple and its corresponding multi-label target vector at the specified index.

Parameters
idx (*int*) – Index of the triple to retrieve.

Returns

A tuple containing the following elements: - The triple as a torch.Tensor of shape (3,). - The multi-label target vector as a torch.Tensor of shape (*target_dim*,), where each element

indicates the presence (1) or absence (0) of a label for the given triple.

Return type
tuple

class dicee.**MultiClassClassificationDataset**(*subword_units: numpy.ndarray*, *block_size: int = 8*)

Bases: torch.utils.data.Dataset

A dataset class for multi-class classification tasks, specifically designed for the 1vsALL training strategy in knowledge graph embedding models. This dataset supports tasks where the model predicts a single correct label from all possible labels for a given input.

Parameters

- **subword_units** (*np.ndarray*) – An array of subword unit indices representing the training data. Each row in the array corresponds to a sequence of subword units (e.g., Byte Pair Encoding tokens) that have been converted to their respective numeric indices.
- **block_size** (*int, optional*) – The size of each sequence of subword units to be used as input to the model. This defines the length of the sequences that the model will receive as input, by default 8.

num_of_data_points

The number of sequences or data points available in the dataset, calculated based on the length of the *subword_units* array and the *block_size*.

Type
int

collate_fn

An optional custom collate function to be used with a PyTorch DataLoader. It's set to None by default and can be specified after initializing the dataset if needed.

Type
None or callable

Note: This dataset is tailored for training knowledge graph embedding models on tasks where the output is a single label out of many possible labels (1vsALL strategy). It is especially suited for models trained with subword tokenization methods like Byte Pair Encoding (BPE), where inputs are sequences of subword unit indices.

__len__() → int

Returns the total number of sequences or data points available in the dataset.

Returns
The number of sequences or data points.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves an input sequence and its subsequent target sequence for next token prediction.

Parameters
idx (*int*) – The starting index for the sequence to be retrieved from the dataset.

Returns

A tuple containing two elements: - *x*: The input sequence as a torch.Tensor of shape (*block_size*,). - *y*: The target sequence as a torch.Tensor of shape (*block_size*,), offset by one position

from the input sequence.

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.**OnevsAllDataset**(*train_set_idx: numpy.ndarray*, *entity_idxs*)

Bases: torch.utils.data.Dataset

A dataset for the One-vs-All (1vsAll) training strategy designed for knowledge graph embedding tasks. This dataset structure is particularly suited for models predicting a single correct label (entity) out of all possible entities for a given pair of head entity and relation.

Parameters

- **train_set_idx** (*np.ndarray*) – An array containing indexed triples from the knowledge graph. Each row represents a triple consisting of indices for the head entity, relation, and tail entity, respectively.
- **entity_idxs** (*dict*) – A dictionary mapping entity names to their corresponding unique integer indices. This is used to determine the dimensionality of the target vector in the 1vsAll setting.

train_data

A tensor version of *train_set_idx*, prepared for use with PyTorch models.

Type
torch.LongTensor

target_dim

The dimensionality of the target vector, equivalent to the total number of unique entities in the dataset.

Type
int

collate_fn

An optional custom collate function for use with a PyTorch DataLoader. By default, it is set to None and can be specified after initializing the dataset.

Type
None or callable

Note: This dataset is optimized for training knowledge graph embedding models using the 1vsAll strategy, where the model aims to correctly predict the tail entity from all possible entities given the head entity and relation.

__len__()

Returns the total number of triples in the dataset.

Returns
The total number of triples.

Return type
int

__getitem__(*idx*)

Retrieves the input data and target vector for the triple at index *idx*.

The input data consists of the indices for the head entity and relation, while the target vector is a one-hot encoded vector with a *1* at the position corresponding to the tail entity's index and `0`s elsewhere.

Parameters
idx (*int*) – The index of the triple to retrieve.

Returns
A tuple containing two elements: - The input data as a torch.Tensor of shape (2,), containing the indices of the head entity and relation. - The target vector as a torch.Tensor of shape (*target_dim*,), a one-hot encoded vector for the tail entity.

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.**KvsAll**(*train_set_idx: numpy.ndarray*, *entity_idxs*, *relation_idxs*, *form*, *store=None*, *label_smoothing_rate: float = 0.0*)

Bases: torch.utils.data.Dataset

Creates a dataset for K-vs-All training strategy, inheriting from torch.utils.data.Dataset. This dataset is tailored for training scenarios where a model predicts all valid tail entities given a head entity and relation pair or vice versa. The labels are multi-hot encoded to represent the presence of multiple valid entities.

Let (D) denote a dataset for KvsAll training and be defined as (D := {(x, y)_i}_{i=1}^{N}), where: (x: (h, r)) is a unique tuple of an entity (h in E) and a relation (r in R) that has been seen in the input graph. (y) denotes a multi-label vector (in [0, 1]^{|E|}) is a binary label. For all (y_i = 1) s.t. ((h, r, E_i) in KG).

Parameters

- **train_set_idx** (*numpy.ndarray*) – A numpy array of shape *(n, 3)* representing *n* triples, where each triple consists of integer indices corresponding to a head entity, a relation, and a tail entity.
- **entity_idxs** (*dict*) – A dictionary mapping entity names (strings) to their unique integer identifiers.

- **relation_idxs** (*dict*) – A dictionary mapping relation names (strings) to their unique integer identifiers.
- **form** (*str*) – A string indicating the prediction form, either 'RelationPrediction' or 'EntityPrediction'.
- **store** (*dict, optional*) – A precomputed dictionary storing the training data points. If provided, it should map tuples of entity and relation indices to lists of entity indices. If *None*, the store will be constructed from *train_set_idx*.
- **label_smoothing_rate** (*float, default=0.0*) – A float representing the rate of label smoothing to be applied. A value of 0 means no label smoothing is applied.

train_data

Tensor containing the input features for the model, typically consisting of pairs of entity and relation indices.

Type
torch.LongTensor

train_target

Tensor containing the target labels for the model, multi-hot encoded to indicate the presence of multiple valid entities.

Type
torch.LongTensor

target_dim

The dimensionality of the target labels, corresponding to the number of unique entities or relations, depending on the *form*.

Type
int

collate_fn

Placeholder for a custom collate function to be used with a PyTorch DataLoader. This is typically set to *None* and can be overridden as needed.

Type
None

Note: The K-vs-All training strategy is used in scenarios where the task is to predict multiple valid entities given a single entity and relation pair. This dataset supports both predicting multiple valid tail entities given a head entity and relation (EntityPrediction) and predicting multiple valid relations given a pair of entities (RelationPrediction).

The label smoothing rate can be adjusted to control the degree of smoothing applied to the target labels, which can help with regularization and model generalization.

__len__() → int

Returns the number of items in the dataset.

Returns
The total number of items.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves the input pair (head entity, relation) and the corresponding multi-label target vector for the item at index *idx*.

The target vector is a binary vector of length *target_dim*, where each element indicates the presence or absence of a tail entity for the given input pair.

Parameters

idx (*int*) – The index of the item to retrieve.

Returns

A tuple containing two elements: - The input pair as a torch.Tensor of shape (2,), containing the indices of the head entity and relation. - The multi-label target vector as a torch.Tensor of shape (*target_dim*,), indicating the presence or

absence of each possible tail entity.

Return type

Tuple[torch.Tensor, torch.Tensor]

class dicee.**AllvsAll**(*train_set_idx: numpy.ndarray*, *entity_idxs*, *relation_idxs*, *label_smoothing_rate=0.0*)

Bases: `torch.utils.data.Dataset`

A dataset class for the All-versus-All (AllvsAll) training strategy suitable for knowledge graph embedding models. This strategy considers all possible pairs of entities and relations, regardless of whether they exist in the knowledge graph, to predict the associated tail entities.

Let D denote a dataset for AllvsAll training and be defined as D:= {(x,y)_i}_i ^N, where x: (h,r) is a possible unique tuple of an entity h in E and a relation r in R. Hence N = |E| x |R| y: denotes a multi-label vector in [0,1]^{|E|} is a binary label.

orall y_i =1 s.t. (h, r, E_i) in KG.

This setup extends beyond observed triples to include all possible combinations of entities and relations, marking non-existent combinations as negatives. It aims to enrich the training data with hard negatives.

train_set_idx

[numpy.ndarray] An array of shape *(n, 3)*, where each row represents a triple (head entity index, relation index, tail entity index).

entity_idxs

[dict] A dictionary mapping entity names to their unique integer indices.

relation_idxs

[dict] A dictionary mapping relation names to their unique integer indices.

label_smoothing_rate

[float, default=0.0] A parameter for label smoothing to mitigate overfitting by softening the hard labels.

train_data

[torch.LongTensor] A tensor containing all possible pairs of entities and relations derived from the input triples.

train_target

[Union[np.ndarray, list]] A target structure (either a Numpy array or a list) indicating the existence of a tail entity for each head entity and relation pair. It supports multi-label classification where a pair can have multiple correct tail entities.

target_dim

[int] The dimension of the target vector, equal to the total number of unique entities.

collate_fn

[None or callable] An optional function to merge a list of samples into a batch for loading. If not provided, the default collate function of PyTorch's DataLoader will be used.

__len__() → int

Returns the number of items in the dataset, including both existing and potential triples.

Returns
The total number of items.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves the input pair (head entity, relation) and the corresponding multi-label target vector for the item at index *idx*. The target vector is a binary vector of length *target_dim*, where each element indicates the presence or absence of a tail entity for the given input pair, including negative samples.

Parameters
idx (*int*) – The index of the item to retrieve.

Returns

A tuple containing two elements: - The input pair as a torch.Tensor of shape (2,), containing the indices of the head entity and relation. - The multi-label target vector as a torch.Tensor of shape (*target_dim*,), indicating the presence or

absence of each possible tail entity, including negative samples.

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.**KvsSampleDataset**(*train_set: numpy.ndarray*, *num_entities*, *num_relations*, *neg_sample_ratio: int = None*, *label_smoothing_rate: float = 0.0*)

Bases: torch.utils.data.Dataset

Constructs a dataset for KvsSample training strategy, specifically designed for knowledge graph embedding models. This dataset formulation is aimed at handling the imbalance between positive and negative examples for each (head, relation) pair by subsampling tail entities. The subsampling ensures a balanced representation of positive and negative examples in each training batch, according to the specified negative sampling ratio.

The dataset is defined as (D:= {(x,y)_i}_{i=1}^{N}), where:

- (x: (h,r)) is a unique head entity (h in E) and a relation (r in R).
- (y in [0,1]^{|E|}) is a binary label vector. For all (y_i = 1) such that ((h, r, E_i) in KG).

At each mini-batch construction, we subsample (y), hence (|new_y| ll |E|). The new (y) contains all 1's if (sum(y) <) neg_sample_ratio, otherwise, it contains a balanced mix of 1's and 0's.

Parameters

- **train_set** (*np.ndarray*) – An array of shape ((n, 3)), where (n) is the number of triples in the dataset. Each row in the array represents a triple ((h, r, t)), consisting of head entity index (h), relation index (r), and tail entity index (t).
- **num_entities** (*int*) – The total number of unique entities in the dataset.
- **num_relations** (*int*) – The total number of unique relations in the dataset.
- **neg_sample_ratio** (*int*) – The ratio of negative samples to positive samples for each (head, relation) pair. If the number of available positive samples is less than this ratio, additional negative samples are generated to meet the ratio.
- **label_smoothing_rate** (*float, default=0.0*) – A parameter for label smoothing, aiming to mitigate overfitting by softening the hard labels. The labels are adjusted towards a uniform distribution, with the smoothing rate determining the degree of softening.

train_data

A tensor containing the (head, relation) pairs derived from the input triples, used to index the training set.

Type
torch.IntTensor

train_target

A list where each element corresponds to the tail entity indices associated with a given (head, relation) pair.

Type
list of numpy.ndarray

collate_fn

A function to merge a list of samples to form a batch. If None, PyTorch's default collate function is used.

Type
None or callable

__len__()

Returns the total number of unique (head, relation) pairs in the dataset.

Returns
The number of unique (head, relation) pairs.

Return type
int

__getitem__(*idx*)

Retrieves the data for the given index, including the (head, relation) pair, selected tail entity indices, and their labels. Positive examples are sampled from the training set, and negative examples are generated by randomly selecting tail entities not associated with the (head, relation) pair.

Parameters
idx (*int*) – The index of the (head, relation) pair in the dataset.

Returns

A tuple containing the following elements: - x: The (head, relation) pair as a torch.Tensor. - y_idx: The indices of selected tail entities, both positive and negative, as a torch.IntTensor. - y_vec: The labels for the selected tail entities, with 1s indicating positive and 0s indicating negative

examples, as a torch.Tensor.

Return type
tuple

class dicee.**NegSampleDataset**(*train_set: numpy.ndarray*, *num_entities: int*, *num_relations: int*, *neg_sample_ratio: int = 1*)

Bases: torch.utils.data.Dataset

A dataset for training knowledge graph embedding models using negative sampling. For each positive triple from the knowledge graph, a negative triple is generated by corrupting either the head or the tail entity with a randomly selected entity.

Parameters

- **train_set** (*np.ndarray*) – The training set of triples, where each triple consists of indices of the head entity, relation, and tail entity.
- **num_entities** (*int*) – The total number of unique entities in the knowledge graph.
- **num_relations** (*int*) – The total number of unique relations in the knowledge graph.

- **neg_sample_ratio** (*int, default=1*) – The ratio of negative samples to positive samples. Currently, it generates one negative sample per positive sample.

train_set

The training set converted to a PyTorch tensor and expanded to include a batch dimension.

Type
torch.Tensor

length

The total number of triples in the training set.

Type
int

num_entities

A tensor containing the total number of entities.

Type
torch.tensor

num_relations

A tensor containing the total number of relations.

Type
torch.tensor

neg_sample_ratio

A tensor containing the ratio of negative to positive samples.

Type
torch.tensor

__len__() → int

Returns the total number of triples in the dataset.

Returns
The total number of triples.

Return type
int

__getitem__(*idx: int*) → Tuple[torch.Tensor, torch.Tensor]

Retrieves a pair consisting of a positive triple and a generated negative triple along with their labels.

Parameters
idx (*int*) – The index of the triple to retrieve.

Returns
A tuple where the first element is a tensor containing a pair of positive and negative triples, and the second element is a tensor containing their respective labels (1 for positive, 0 for negative).

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.**TriplePredictionDataset**(*train_set: numpy.ndarray*, *num_entities: int*, *num_relations: int*, *neg_sample_ratio: int = 1*, *label_smoothing_rate: float = 0.0*)

Bases: torch.utils.data.Dataset

A dataset for triple prediction using negative sampling and label smoothing.

D:= {(x)_i}_i ^N, where - x:(h,r, t) in KG is a unique h in E and a relation r in R and - collact_fn => Generates negative triples

collect_fn:

orall (h,r,t) in G obtain, create negative triples{(h,r,x),(,r,t),(h,m,t)}

y: labels are represented in torch.float16

This dataset generates negative triples by corrupting either the head or the tail of each positive triple from the training set. The corruption is performed by randomly replacing the head or the tail with another entity from the entity set. The dataset supports label smoothing to soften the target labels, which can help improve generalization.

train_set
[np.ndarray] The training set consisting of triples in the form of (head, relation, tail) indices.

num_entities
[int] The total number of unique entities in the knowledge graph.

num_relations
[int] The total number of unique relations in the knowledge graph.

neg_sample_ratio
[int, optional] The ratio of negative samples to generate for each positive sample. Default is 1.

label_smoothing_rate
[float, optional] The rate of label smoothing to apply to the target labels. Default is 0.0.

The *collate_fn* should be passed to the DataLoader's *collate_fn* argument to ensure proper batch processing and negative sample generation.

__len__() → int

Returns the total number of triples in the dataset.

Returns
The total number of triples.

Return type
int

__getitem__(*idx: int*) → torch.Tensor

Retrieves a triple for the given index.

Parameters
idx (*int*) – The index of the triple to retrieve.

Returns
The triple at the specified index.

Return type
torch.Tensor

collate_fn(*batch: List[torch.Tensor]*) → Tuple[torch.Tensor, torch.Tensor]

Custom collate function to generate a batch of positive and negative triples along with their labels.

Parameters
batch (*List[torch.Tensor]*) – A list of tensors representing triples.

Returns
A tuple containing a tensor of triples and a tensor of corresponding labels.

Return type
Tuple[torch.Tensor, torch.Tensor]

class dicee.**CVDataModule**(*train_set_idx: numpy.ndarray*, *num_entities: int*, *num_relations: int*, *neg_sample_ratio: int*, *batch_size: int*, *num_workers: int*)

Bases: `pytorch_lightning.LightningDataModule`

A LightningDataModule for setting up data loaders for cross-validation training of knowledge graph embedding models.

Parameters

- **train_set_idx** (*np.ndarray*) – An array of indexed triples for training, where each triple consists of indices of the head entity, relation, and tail entity.
- **num_entities** (*int*) – The total number of unique entities in the knowledge graph.
- **num_relations** (*int*) – The total number of unique relations in the knowledge graph.
- **neg_sample_ratio** (*int*) – The ratio of negative samples to positive samples for each positive triple.
- **batch_size** (*int*) – The number of samples in each batch of data.
- **num_workers** (*int*) – The number of subprocesses to use for data loading. https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

Returns
A PyTorch DataLoader for the training dataset.

Return type
DataLoader

train_dataloader() → torch.utils.data.DataLoader

Creates a DataLoader for the training dataset.

Returns
A DataLoader object that loads the training data.

Return type
DataLoader

setup(**args*, ***kwargs*)

Called at the beginning of fit (train + validate), validate, test, or predict. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

Parameters
stage – either `'fit'`, `'validate'`, `'test'`, or `'predict'`

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

        # don't do this
        self.something = else

    def setup(self, stage):
```

(continues on next page)

(continued from previous page)

```
        data = load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

transfer_batch_to_device(**args*, ***kwargs*)

Override this hook if your `DataLoader` returns tensors wrapped in a custom data structure.

The data types listed below (and any arbitrary nesting of them) are supported out of the box:

- `torch.Tensor` or anything that implements *.to(...)*
- `list`
- `dict`
- `tuple`

For anything else, you need to define how the data is moved to the target device (CPU, GPU, TPU, …).

Note: This hook should only transfer the data and not modify it, nor should it move the data to any other device than the one passed in as argument (unless you know what you are doing). To check the current state of execution of this hook you can use `self.trainer.training/testing/validating/predicting` so that you can add different logic as per your requirement.

Parameters

- **batch** – A batch of data that needs to be transferred to a new device.
- **device** – The target device as defined in PyTorch.
- **dataloader_idx** – The index of the dataloader to which the batch belongs.

Returns

A reference to the data on the new device.

Example:

```
def transfer_batch_to_device(self, batch, device, dataloader_idx):
    if isinstance(batch, CustomBatch):
        # move all tensors in your custom data structure to the device
        batch.samples = batch.samples.to(device)
        batch.targets = batch.targets.to(device)
    elif dataloader_idx == 0:
        # skip device transfer for the first dataloader or anything you wish
        pass
    else:
        batch = super().transfer_batch_to_device(batch, device, dataloader_
↪idx)
    return batch
```

Raises

MisconfigurationException – If using IPUs, `Trainer(accelerator='ipu')`.

See also:

- `move_data_to_device()`
- `apply_to_collection()`

prepare_data(**args*, ***kwargs*)

Use this to download and prepare data. Downloading and saving data with multiple processes (distributed settings) will result in corrupted data. Lightning ensures this method is called only within a single process, so you can safely add your downloading logic within.

> **Warning:** DO NOT set state to the model (use `setup` instead) since this is NOT called on every device

Example:

```
def prepare_data(self):
    # good
    download_data()
    tokenize()
    etc()

    # bad
    self.split = data_split
    self.some_state = some_other_state()
```

In a distributed environment, `prepare_data` can be called in two ways (using prepare_data_per_node)

1. Once per node. This is the default and is only called on LOCAL_RANK=0.
2. Once in total. Only called on GLOBAL_RANK=0.

Example:

```
# DEFAULT
# called once per node on LOCAL_RANK=0 of that node
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = True


# call on GLOBAL_RANK=0 (great for shared file systems)
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = False
```

This is called before requesting the dataloaders:

```
model.prepare_data()
initialize_distributed()
model.setup(stage)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
model.predict_dataloader()
```

class dicee.**QueryGenerator**(*train_path*, *val_path: str*, *test_path: str*, *ent2id: Dict = None*, *rel2id: Dict = None*, *seed: int = 1*, *gen_valid: bool = False*, *gen_test: bool = True*)

list2tuple(*list_data*)

tuple2list (*x: List | Tuple*) → List | Tuple

Convert a nested tuple to a nested list.

set_global_seed (*seed: int*)

Set seed

construct_graph (*paths: List[str]*) → Tuple[Dict, Dict]

Construct graph from triples Returns dicts with incoming and outgoing edges

fill_query (*query_structure: List[str | List]*, *ent_in: Dict*, *ent_out: Dict*, *answer: int*) → bool

Private method for fill_query logic.

achieve_answer (*query: List[str | List]*, *ent_in: Dict*, *ent_out: Dict*) → set

Private method for achieve_answer logic. @TODO: Document the code

write_links (*ent_out*, *small_ent_out*)

ground_queries (*query_structure: List[str | List]*, *ent_in: Dict*, *ent_out: Dict*, *small_ent_in: Dict*, *small_ent_out: Dict*, *gen_num: int*, *query_name: str*)

Generating queries and achieving answers

unmap (*query_type*, *queries*, *tp_answers*, *fp_answers*, *fn_answers*)

unmap_query (*query_structure*, *query*, *id2ent*, *id2rel*)

generate_queries (*query_struct: List*, *gen_num: int*, *query_type: str*)

Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting queries and answers in return @ TODO: create a class for each single query struct

save_queries (*query_type: str*, *gen_num: int*, *save_path: str*)

abstract load_queries (*path*)

get_queries (*query_type: str*, *gen_num: int*)

static save_queries_and_answers (*path: str*, *data: List[Tuple[str, Tuple[collections.defaultdict]]]*) → None

Save Queries into Disk

static load_queries_and_answers (*path: str*) → List[Tuple[str, Tuple[collections.defaultdict]]]

Load Queries from Disk to Memory

dicee.**__version__ = '0.1.4'**

Python Module Index

d

Index

Non-alphabetical

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W