DICE Embeddings

Release 0.1.3.2

Caglar Demir

Sep 15, 2025

Contents:

1 Dicee Manual **2**

2 Installation **3**
2.1 Installation from Source . . . 3

3 Download Knowledge Graphs **3**

4 Knowledge Graph Embedding Models **3**

5 How to Train **3**

6 Creating an Embedding Vector Database **5**
6.1 Learning Embeddings . . . 5
6.2 Loading Embeddings into Qdrant Vector Database . . . 6
6.3 Launching Webservice . . . 6

7 Answering Complex Queries **6**

8 Predicting Missing Links **8**

9 Downloading Pretrained Models **8**

10 How to Deploy **8**

11 Docker **8**

12 Coverage Report **8**

13 How to cite **10**

14 dicee **12**
14.1 Submodules . . . 12
14.2 Attributes . . . 170
14.3 Classes . . . 171
14.4 Functions . . . 172
14.5 Package Contents . . . 173

Python Module Index **220**

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

1 Dicee Manual

Version: dicee 0.2.0

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

Publisher and maintainer: Caglar Demir[2]

Contact: caglar.demir@upb.de

License: OSI Approved :: MIT License

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
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
```

12 Coverage Report

The coverage report is generated using coverage.py[12]:

```
Name                                                    Stmts   Miss  Cover   Missing
-------------------------------------------------------------------------------------
dicee/__init__.py                                           7      0   100%
dicee/abstracts.py                                        338    115    66%   112-113,␣
```

(continues on next page)

[11] https://files.dice-research.org/projects/DiceEmbeddings/
[12] https://coverage.readthedocs.io/en/7.6.0/

(continued from previous page)

```
↪131, 154-155, 160, 173, 197, 240-254, 290, 303-306, 309-313, 353-364, 379-387, 402,␣
↪413-417, 427-428, 434-436, 442-445, 448-453, 576-596, 602-606, 610-612, 631, 658-696
dicee/callbacks.py                                     248    103    58%   50-55,␣
↪67-73, 76, 88-93, 98-103, 106-109, 116-133, 138-142, 146-147, 247, 281-285, 291-292,
↪ 310-316, 319, 324-325, 337-343, 349-358, 363-365, 410, 421-434, 438-473, 485-491
dicee/config.py                                         97      2    98%   146-147
dicee/dataset_classes.py                               430    146    66%   16, 44,␣
↪57, 89-98, 104, 111-118, 121, 124, 127-151, 207-213, 216, 219-221, 324, 335-338,␣
↪354, 420-421, 439, 562-581, 583, 587-599, 606-615, 618, 622-636, 780-787, 790-794,␣
↪845, 866-878, 902-915, 937, 941-954, 964-967, 973, 985, 987, 989, 1012-1022
dicee/eval_static_funcs.py                             256    100    61%   104, 109,
↪ 114, 261-356, 363-414, 442, 465-468
dicee/evaluator.py                                     267     48    82%   48, 53,␣
↪58, 77, 82-83, 86, 102, 119, 130, 134, 139, 173-184, 191-202, 310, 340-358, 452,␣
↪462, 480-485
dicee/executer.py                                      134     16    88%   53-57,␣
↪166-176, 235-236, 283
dicee/knowledge_graph.py                                82     10    88%   84, 94-
↪95, 124, 128, 132-134, 137-138, 140
dicee/knowledge_graph_embeddings.py                    654    415    37%   25, 28-
↪29, 37-50, 55-88, 91-125, 129-137, 171, 173-229, 261, 265, 276-277, 301-303, 311,␣
↪339-362, 493, 497-519, 523-547, 580, 656, 665, 710-716, 748, 806-1171, 1202-1263,␣
↪1267-1295, 1326, 1332
dicee/models/__init__.py                                 9      0   100%
dicee/models/adopt.py                                  187    172     8%   50-86,␣
↪99-110, 129-185, 195-242, 266-322, 346-448, 484-517
dicee/models/base_model.py                             240     35    85%   30-35,␣
↪64, 66, 92, 99-116, 171, 204, 244, 250, 259, 262, 266, 273, 277, 279, 294, 307-308,␣
↪362, 365, 438, 450
dicee/models/clifford.py                               470    278    41%   10, 12,␣
↪16, 24-25, 52-56, 79-87, 101-103, 108-109, 140-160, 184, 191, 195-256, 273-277, 289,
↪ 292, 297, 302, 346-361, 377-444, 464-470, 483, 486, 491, 496, 525-531, 544, 547,␣
↪552, 557, 567-576, 592-593, 613-685, 696-699, 724-749, 773-806, 842-846, 859, 869,␣
↪872, 877, 882, 887, 891, 895, 904-905, 935, 942, 947, 975-979, 1007-1016, 1026-1034,
↪ 1052-1054, 1072-1074, 1090-1092
dicee/models/complex.py                                162     25    85%   86-109,␣
↪273-287
dicee/models/dualE.py                                   59     10    83%   93-102,␣
↪142-156
dicee/models/ensemble.py                                89     67    25%   7-29, 31,
↪ 34, 37, 40, 43, 46, 49, 52-54, 56-58, 64-68, 71-90, 93-94, 97-112, 131
dicee/models/function_space.py                         262    221    16%   10-23,␣
↪27-36, 39-48, 52-69, 76-87, 90-99, 102-111, 115-127, 135-157, 160-166, 169-186, 189-
↪195, 198-206, 209, 214-235, 244-247, 251-255, 259-268, 272-293, 302-308, 312-329,␣
↪333-336, 345-353, 356, 367-373, 393-407, 425-439, 444-454, 462-466, 475-479
dicee/models/literal.py                                 33      1    97%   82
dicee/models/octonion.py                               227     83    63%   21-44,␣
↪320-329, 334-345, 348-370, 374-416, 426-474
dicee/models/pykeen_models.py                           55      5    91%   77-80,␣
↪135
dicee/models/quaternion.py                             192     69    64%   7-21, 30-
↪55, 68-72, 107, 185, 328-342, 345-364, 368-389, 399-426
```

(continues on next page)

(continued from previous page)

```
dicee/models/real.py                                    61     12    80%   37-42,␣
↪70-73, 91, 107-110
dicee/models/static_funcs.py                            10      0   100%
dicee/models/transformers.py                           234    189    19%   20-39,␣
↪42, 56-71, 80-98, 101-112, 119-121, 124, 130-147, 151-176, 182-186, 189-193, 199-
↪203, 206-208, 225-252, 261-264, 267-272, 275-300, 306-311, 315-368, 372-394, 400-410
dicee/query_generator.py                               374    346     7%   17-51,␣
↪55, 61-64, 68-69, 77-91, 99-146, 154-187, 191-205, 211-268, 273-302, 306-442, 452-
↪471, 479-502, 509-513, 518, 523-529
dicee/read_preprocess_save_load_kg/__init__.py           3      0   100%
dicee/read_preprocess_save_load_kg/preprocess.py       243     40    84%   33, 39,␣
↪76, 100-125, 131, 136-149, 175, 205, 380-381
dicee/read_preprocess_save_load_kg/read_from_disk.py    36     11    69%   34, 38-
↪40, 47, 55, 58-72
dicee/read_preprocess_save_load_kg/save_load_disk.py    53     21    60%   29-30,␣
↪38, 47-68
dicee/read_preprocess_save_load_kg/util.py             236    125    47%   159, 173-
↪175, 179-180, 198-204, 207-209, 214-216, 230, 244-247, 252-260, 265-271, 276-281,␣
↪286-291, 303-324, 330-386, 390-394, 398-399, 403, 407-408, 436, 441, 448-449
dicee/sanity_checkers.py                                47     19    60%   8-12, 21-
↪31, 46, 51, 58, 69-79
dicee/static_funcs.py                                  483    194    60%   42, 52,␣
↪58-63, 85, 92-96, 109-119, 129-131, 136, 143, 167, 172, 184, 190, 198, 202, 229-233,
↪ 295, 303-309, 320-330, 341-361, 389, 413-414, 419-420, 437-438, 440-441, 443-444,␣
↪452, 470-474, 491-494, 498-503, 507-511, 515-516, 522-524, 539-553, 558-561, 566-
↪569, 578-629, 634-646, 663-680, 683-691, 695-713, 724
dicee/static_funcs_training.py                         155     66    57%   7-10,␣
↪222-319, 327-328
dicee/static_preprocess_funcs.py                        98     43    56%   17-25,␣
↪50, 57, 59, 70, 83-107, 112-115, 120-123, 128-131
dicee/trainer/__init__.py                                1      0   100%
dicee/trainer/dice_trainer.py                          151     18    88%   22, 30-
↪31, 33-35, 97, 104, 109-114, 152, 237, 280-283
dicee/trainer/model_parallelism.py                      99     87    12%   10-25,␣
↪30-116, 121-132, 136, 141-197
dicee/trainer/torch_trainer.py                          77      6    92%   31, 102,␣
↪168, 179-181
dicee/trainer/torch_trainer_ddp.py                      89     71    20%   11-14,␣
↪43, 47-67, 78-94, 113-122, 126-136, 151-158, 168-191
----------------------------------------------------------------------------------
TOTAL                                                 6948   3169    54%
```

13 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}
↪,
```

(continues on next page)

(continued from previous page)

```
  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},
  pdf =          {https://proceedings.mlr.press/v157/demir21a/demir21a.pdf},
  url =          {https://proceedings.mlr.press/v157/demir21a.html},
}
# ConEx
```

(continues on next page)

(continued from previous page)

```
@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`

14 dicee

14.1 Submodules

dicee.__main__

dicee.abstracts

Classes

AbstractTrainer	Abstract class for Trainer class for knowledge graph embedding models
BaseInteractiveKGE	Abstract/base class for using knowledge graph embedding models interactively.
InteractiveQueryDecomposition	
AbstractCallback	Abstract class for Callback class for knowledge graph embedding models
AbstractPPECallback	Abstract class for Callback class for knowledge graph embedding models
BaseInteractiveTrainKGE	Abstract/base class for training knowledge graph embedding models interactively.

Module Contents

class dicee.abstracts.**AbstractTrainer**(*args*, *callbacks*)

Abstract class for Trainer class for knowledge graph embedding models

Parameter

args

[str] ?

callbacks: list

?

attributes

callbacks

is_global_zero = True

global_rank = 0

local_rank = 0

strategy = None

on_fit_start(**args*, ***kwargs*)

A function to call callbacks before the training starts.

Parameter

args

kwargs

rtype
None

on_fit_end(**args*, ***kwargs*)

A function to call callbacks at the ned of the training.

Parameter

args

kwargs

rtype
None

on_train_epoch_start(**args*, ***kwargs*)

A function to call callbacks at the start of an epoch.

Parameter

args

kwargs

rtype
None

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

A function to call callbacks at the end of an epoch.

Parameter

args

kwargs

rtype
None

on_train_batch_end(**args*, ***kwargs*)

A function to call callbacks at the end of each mini-batch during training.

Parameter

args

kwargs

rtype
None

static save_checkpoint(*full_path: str*, *model*) → None

A static function to save a model into disk

Parameter

full_path : str

model:

rtype
None

class dicee.abstracts.**BaseInteractiveKGE**(*path: str = None*, *url: str = None*, *construct_ensemble: bool = False*, *model_name: str = None*, *apply_semantic_constraint: bool = False*)

Abstract/base class for using knowledge graph embedding models interactively.

Parameter

path_of_pretrained_model_dir
[str] ?

construct_ensemble: boolean
?

model_name: str apply_semantic_constraint : boolean

construct_ensemble = False

apply_semantic_constraint = False

configs

get_eval_report() → dict

get_bpe_token_representation(*str_entity_or_relation: List[str] | str*) → List[List[int]] | List[int]

Parameters
str_entity_or_relation (*corresponds to a str or a list of strings to be tokenized via BPE and shaped.*)

Return type
A list integer(s) or a list of lists containing integer(s)

get_padded_bpe_triple_representation(*triples: List[List[str]]*) → Tuple[List, List, List]

Parameters
triples

set_model_train_mode() → None

Setting the model into training mode

Parameter

set_model_eval_mode() → None

Setting the model into eval mode

Parameter

property name

sample_entity(*n: int*) → List[str]

sample_relation(*n: int*) → List[str]

is_seen(*entity: str = None*, *relation: str = None*) → bool

save() → None

get_entity_index(*x: str*)

get_relation_index(*x: str*)

index_triple(*head_entity: List[str]*, *relation: List[str]*, *tail_entity: List[str]*) → Tuple[torch.LongTensor, torch.LongTensor, torch.LongTensor]

Index 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.

Returns: Tuple

pytorch tensor of triple score

add_new_entity_embeddings(*entity_name: str = None*, *embeddings: torch.FloatTensor = None*)

get_entity_embeddings(*items: List[str]*)

Return embedding of an entity given its string representation

Parameter

items:
entities

get_relation_embeddings(*items: List[str]*)

Return embedding of a relation given its string representation

Parameter

items:
relations

construct_input_and_output(*head_entity: List[str]*, *relation: List[str]*, *tail_entity: List[str]*, *labels*)

Construct a data point :param head_entity: :param relation: :param tail_entity: :param labels: :return:

parameters()

class dicee.abstracts.**InteractiveQueryDecomposition**

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

class dicee.abstracts.**AbstractCallback**

Bases: abc.ABC, lightning.pytorch.callbacks.Callback

Abstract class for Callback class for knowledge graph embedding models

Parameter

on_init_start(**args*, ***kwargs*)

Parameter

trainer:

model:

rtype
None

on_init_end(**args*, ***kwargs*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype
None

on_fit_start(*trainer*, *model*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype
None

on_train_epoch_end(*trainer*, *model*)
Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype
None

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*)
Call at the end of the training.

Parameter

trainer:

model:

rtype
None

class dicee.abstracts.**AbstractPPECallback**(*num_epochs*, *path*, *epoch_to_start*, *last_percent_to_consider*)

Bases: *AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

num_epochs

path

sample_counter = 0

epoch_count = 0

alphas = None

on_fit_start(*trainer*, *model*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype
None

on_fit_end(*trainer*, *model*)

Call at the end of the training.

Parameter

trainer:

model:

rtype
None

store_ensemble(*param_ensemble*) → None

class dicee.abstracts.**BaseInteractiveTrainKGE**

Abstract/base class for training knowledge graph embedding models interactively. This class provides methods for re-training KGE models and also Literal Embedding model.

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.

train_literals(*train_file_path: str = None*, *num_epochs: int = 100*, *lit_lr: float = 0.001*, *lit_normalization_type: str = 'z-norm'*, *batch_size: int = 1024*, *sampling_ratio: float = None*, *random_seed=1*, *loader_backend: str = 'pandas'*, *freeze_entity_embeddings: bool = True*, *gate_residual: bool = True*, *device: str = None*, *suffle_data: bool = True*)

Trains the Literal Embeddings model using literal data.

Parameters

- **train_file_path** (*str*) – Path to the training data file.
- **num_epochs** (*int*) – Number of training epochs.
- **lit_lr** (*float*) – Learning rate for the literal model.
- **norm_type** (*str*) – Normalization type to use (‘z-norm’, ‘min-max’, or None).
- **batch_size** (*int*) – Batch size for training.
- **sampling_ratio** (*float*) – Ratio of training triples to use.

- **loader_backend** (*str*) – Backend for loading the dataset ('pandas' or 'rdflib').
- **freeze_entity_embeddings** (*bool*) – If True, freeze the entity embeddings during training.
- **gate_residual** (*bool*) – If True, use gate residual connections in the model.
- **device** (*str*) – Device to use for training ('cuda' or 'cpu'). If None, will use available GPU or CPU.
- **suffle_data** (*bool*) – If True, shuffle the dataset before training.

dicee.analyse_experiments

This script should be moved to dicee/scripts Example: python dicee/analyse_experiments.py –dir Experiments –features "model" "trainMRR" "testMRR"

Classes

Experiment

Functions

get_default_arguments()
analyse(args)

Module Contents

dicee.analyse_experiments.**get_default_arguments**()

class dicee.analyse_experiments.**Experiment**

model_name = []

callbacks = []

embedding_dim = []

num_params = []

num_epochs = []

batch_size = []

lr = []

byte_pair_encoding = []

aswa = []

path_dataset_folder = []

full_storage_path = []

pq = []

train_mrr = []

train_h1 = []

train_h3 = []

train_h10 = []

val_mrr = []

val_h1 = []

val_h3 = []

val_h10 = []

test_mrr = []

test_h1 = []

test_h3 = []

test_h10 = []

runtime = []

normalization = []

scoring_technique = []

save_experiment(*x*)

to_df()

dicee.analyse_experiments.**analyse**(*args*)

dicee.callbacks

Classes

AccumulateEpochLossCallback	Abstract class for Callback class for knowledge graph embedding models
PrintCallback	Abstract class for Callback class for knowledge graph embedding models
KGESaveCallback	Abstract class for Callback class for knowledge graph embedding models
PseudoLabellingCallback	Abstract class for Callback class for knowledge graph embedding models
ASWA	Adaptive stochastic weight averaging
Eval	Abstract class for Callback class for knowledge graph embedding models
KronE	Abstract class for Callback class for knowledge graph embedding models
Perturb	A callback for a three-Level Perturbation
PeriodicEvalCallback	Callback to periodically evaluate the model and optionally save checkpoints during training.
LRScheduler	Callback for managing learning rate scheduling and model snapshots.
SWA	Stochastic Weight Averaging callbacks.

Functions

estimate_q(eps)	estimate rate of convergence q from sequence esp
compute_convergence(seq, i)	

Module Contents

class dicee.callbacks.**AccumulateEpochLossCallback**(*path: str*)

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

path

on_fit_end(*trainer*, *model*) → None

Store epoch loss

Parameter

trainer:

model:

rtype

None

class dicee.callbacks.**PrintCallback**

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

start_time

on_fit_start(*trainer*, *pl_module*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_fit_end(*trainer*, *pl_module*)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

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*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype

None

class dicee.callbacks.**KGESaveCallback**(*every_x_epoch: int*, *max_epochs: int*, *path: str*)

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

every_x_epoch

max_epochs

epoch_counter = 0

path

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*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype
None

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

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype
None

on_fit_end(**args*, ***kwargs*)

Call at the end of the training.

Parameter

trainer:

model:

rtype
None

on_epoch_end(*model*, *trainer*, ***kwargs*)

class dicee.callbacks.**PseudoLabellingCallback**(*data_module*, *kg*, *batch_size*)

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

data_module

kg

num_of_epochs = 0

unlabelled_size

batch_size

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.

path

num_epochs

initial_eval_setting = None

epoch_count = 0

alphas = []

val_aswa = -1

on_fit_end(*trainer*, *model*)

Call at the end of the training.

Parameter

trainer:

model:

rtype
None

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*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype
None

class dicee.callbacks.**Eval**(*path*, *epoch_ratio: int = None*)

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

path

reports = []

epoch_ratio = None

epoch_counter = 0

on_fit_start(*trainer*, *model*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype
None

on_fit_end(*trainer*, *model*)

Call at the end of the training.

Parameter

trainer:

model:

rtype
None

on_train_epoch_end(*trainer*, *model*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype
None

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 class for Callback class for knowledge graph embedding models

Parameter

f = None

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*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

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:

level = 'input'

ratio = 0.0

method = None

scaler = None

frequency = None

on_train_batch_start(*trainer*, *model*, *batch*, *batch_idx*)

Called when the train batch begins.

class dicee.callbacks.**PeriodicEvalCallback**(*experiment_path: str*, *max_epochs: int*, *eval_every_n_epoch: int = 0*, *eval_at_epochs: list = None*, *save_model_every_n_epoch: bool = True*, *n_epochs_eval_model: str = 'val_test'*)

Bases: *dicee.abstracts.AbstractCallback*

Callback to periodically evaluate the model and optionally save checkpoints during training.

Evaluates at regular intervals (every N epochs) or at explicitly specified epochs. Stores evaluation reports and model checkpoints.

experiment_dir

max_epochs

epoch_counter = 0

save_model_every_n_epoch = True

reports

n_epochs_eval_model = 'val_test'

default_eval_model = None

eval_epochs

on_fit_end(*trainer*, *model*)

Called at the end of training. Saves final evaluation report.

on_train_epoch_end(*trainer*, *model*)

Called at the end of each training epoch. Performs evaluation and checkpointing if scheduled.

class dicee.callbacks.**LRScheduler**(*adaptive_lr_config: dict*, *total_epochs: int*, *experiment_dir: str*, *eta_max: float = 0.1*, *snapshot_dir: str = 'snapshots'*)

Bases: *dicee.abstracts.AbstractCallback*

Callback for managing learning rate scheduling and model snapshots.

Supports cosine annealing ("cca"), MMCCLR ("mmcclr"), and their deferred (warmup) variants: - "deferred_cca" - "deferred_mmcclr"

At the end of each learning rate cycle, the model can optionally be saved as a snapshot.

total_epochs

experiment_dir

snapshot_dir

batches_per_epoch = None

total_steps = None

cycle_length = None

warmup_steps = None

lr_lambda = None

scheduler = None

step_count = 0

snapshot_loss

on_train_start(*trainer*, *model*)

Initialize training parameters and LR scheduler at start of training.

on_train_batch_end(*trainer*, *model*, *outputs*, *batch*, *batch_idx*)

Step the LR scheduler and save model snapshot if needed after each batch.

on_fit_end(*trainer*, *model*)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

class dicee.callbacks.**SWA**(*swa_start_epoch*, *swa_c_epochs: int = 1*, *lr_init: float = 0.1*, *swa_lr: float = 0.05*, *max_epochs: int = None*)

Bases: *dicee.abstracts.AbstractCallback*

Stochastic Weight Averaging callbacks.

swa_start_epoch

swa_c_epochs = 1

swa_lr = 0.05

lr_init = 0.1

max_epochs = None

swa_model = None

swa_n = 0

current_epoch = -1

static moving_average(*swa_model*, *running_model*, *alpha*)

Update SWA model with moving average of current model.

on_train_epoch_start(*trainer*, *model*)

Update learning rate according to SWA schedule.

on_train_epoch_end(*trainer*, *model*)

Apply SWA averaging if conditions are met.

on_fit_end(*trainer*, *model*)

Replace main model with SWA model at the end of training.

dicee.config

Classes

Namespace	Simple object for storing attributes.

Module Contents

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 = None

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 = None
A single directory created that contains related data about embeddings.

path_single_kg = None
Path of a file corresponding to the input knowledge graph

sparql_endpoint = None
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 = None
The ratio of added random triples into training dataset

gpus = None
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

separator: str = '\\s+'
separator for extracting head, relation and tail from a triple

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 = None
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 = None
Not tested

label_smoothing_rate: float = 0.0

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 = None
Read some triples that are uniformly at random sampled. Ratio being between 0 and 1

read_only_few: int = None
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 = None

block size of LLM

continual_learning = None

Path of a pretrained model size of LLM

auto_batch_finding = False

A flag for using auto batch finding

eval_every_n_epochs: int = 0

Evaluate model every n epochs. If 0, no evaluation is applied.

save_every_n_epochs: bool = False

Save model every n epochs. If True, save model at every epoch.

eval_at_epochs: list = None

List of epoch numbers at which to evaluate the model (e.g., 1 5 10).

n_epochs_eval_model: str = 'val_test'

Evaluating link prediction performance on data splits while performing periodic evaluation.

adaptive_lr

"cca"}'

Type

Adaptive learning rate parameters, e.g., '{"scheduler_name"

swa_start_epoch: int = None

Epoch at which to start applying stochastic weight averaging.

__iter__()

dicee.dataset_classes

Classes

BPE_NegativeSamplingDataset	An abstract class representing a `Dataset`.
MultiLabelDataset	An abstract class representing a `Dataset`.
MultiClassClassificationDataset	Dataset for the 1vsALL training strategy
OnevsAllDataset	Dataset for the 1vsALL training strategy
KvsAll	Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.
AllvsAll	Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.
OnevsSample	A custom PyTorch Dataset class for knowledge graph embeddings, which includes
KvsSampleDataset	KvsSample a Dataset:
NegSampleDataset	An abstract class representing a `Dataset`.
TriplePredictionDataset	Triple Dataset
CVDataModule	Create a Dataset for cross validation
LiteralDataset	Dataset for loading and processing literal data for training Literal Embedding model.

Functions

reload_dataset(path, form_of_labelling, ...)	Reload the files from disk to construct the Pytorch dataset
construct_dataset(→ torch.utils.data.Dataset)	

Module Contents

`dicee.dataset_classes.`**`reload_dataset`**`(`*path: str*, *form_of_labelling*, *scoring_technique*, *neg_ratio*, *label_smoothing_rate*`)`

Reload the files from disk to construct the Pytorch 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

`class` `dicee.dataset_classes.`**`BPE_NegativeSamplingDataset`**`(`*train_set: torch.LongTensor*, *ordered_shaped_bpe_entities: torch.LongTensor*, *neg_ratio: int*`)`

Bases: `torch.utils.data.Dataset`

An abstract class representing a `Dataset`.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite *__getitem__()*, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite *__len__()*, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

DataLoader by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

train_set

ordered_bpe_entities

num_bpe_entities

neg_ratio

num_datapoints

__len__()

__getitem__(*idx*)

collate_fn(*batch_shaped_bpe_triples: List[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

An abstract class representing a Dataset.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite *__getitem__()*, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite *__len__()*, which is expected to return the size of the dataset by many Sampler implementations and the default options of DataLoader. Subclasses could also optionally implement __getitems__(), for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

DataLoader by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

train_set

train_indices_target

target_dim

num_datapoints

torch_ordered_shaped_bpe_entities

collate_fn = None

__len__()

__getitem__(*idx*)

class dicee.dataset_classes.**MultiClassClassificationDataset**(*subword_units: numpy.ndarray*, *block_size: int = 8*)

Bases: torch.utils.data.Dataset

Dataset for the 1vsALL training strategy

Parameters

- **train_set_idx** – Indexed triples for the training.
- **entity_idxs** – mapping.
- **relation_idxs** – mapping.
- **form** – ?
- **num_workers** – int for https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

Return type

torch.utils.data.Dataset

train_data

block_size = 8

num_of_data_points

collate_fn = None

__len__()

__getitem__(*idx*)

class dicee.dataset_classes.**OnevsAllDataset**(*train_set_idx: numpy.ndarray*, *entity_idxs*)

Bases: torch.utils.data.Dataset

Dataset for the 1vsALL training strategy

Parameters

- **train_set_idx** – Indexed triples for the training.
- **entity_idxs** – mapping.
- **relation_idxs** – mapping.
- **form** – ?
- **num_workers** – int for https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

Return type

torch.utils.data.Dataset

train_data

target_dim

collate_fn = None

__len__()

__getitem__(*idx*)

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 KvsAll training by inheriting from torch.utils.data.Dataset.

Let D denote a dataset for KvsAll training and be defined as D:= {(x,y)_i}_i ^N, where x: (h,r) is an unique tuple of an entity h in E and a relation r in R that has been seed in the input graph. 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

Note

TODO

train_set_idx
[numpy.ndarray] n by 3 array representing n triples

entity_idxs
[dictonary] string representation of an entity to its integer id

relation_idxs
[dictonary] string representation of a relation to its integer id

self : torch.utils.data.Dataset

```
>>> a = KvsAll()
>>> a
? array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

train_data = None

train_target = None

label_smoothing_rate

collate_fn = None

__len__()

__getitem__(*idx*)

class dicee.dataset_classes.**AllvsAll**(*train_set_idx: numpy.ndarray*, *entity_idxs*, *relation_idxs*, *label_smoothing_rate=0.0*)

Bases: `torch.utils.data.Dataset`

Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.

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

Note

AllvsAll extends KvsAll via none existing (h,r). Hence, it adds data points that are labelled without 1s,
only with 0s.

train_set_idx
[numpy.ndarray] n by 3 array representing n triples

entity_idxs
[dictonary] string representation of an entity to its integer id

relation_idxs
[dictonary] string representation of a relation to its integer id

self : torch.utils.data.Dataset

```
>>> a = AllvsAll()
>>> a
? array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

train_data = None

train_target = None

label_smoothing_rate

collate_fn = None

target_dim

__len__()

__getitem__(*idx*)

class dicee.dataset_classes.**OnevsSample**(*train_set: numpy.ndarray*, *num_entities*, *num_relations*, *neg_sample_ratio: int = None*, *label_smoothing_rate: float = 0.0*)

Bases: torch.utils.data.Dataset

A custom PyTorch Dataset class for knowledge graph embeddings, which includes both positive and negative sampling for a given dataset for multi-class classification problem..

Parameters

- **train_set** (*np.ndarray*) – A numpy array containing triples of knowledge graph data. Each triple consists of (head_entity, relation, tail_entity).
- **num_entities** (*int*) – The number of unique entities in the knowledge graph.
- **num_relations** (*int*) – The number of unique relations in the knowledge graph.
- **neg_sample_ratio** (*int, optional*) – The number of negative samples to be generated per positive sample. Must be a positive integer and less than num_entities.
- **label_smoothing_rate** (*float, optional*) – A label smoothing rate to apply to the positive and negative labels. Defaults to 0.0.

train_data

The input data converted into a PyTorch tensor.

Type
torch.Tensor

num_entities

Number of entities in the dataset.

Type
int

num_relations

Number of relations in the dataset.

Type
int

neg_sample_ratio

Ratio of negative samples to be drawn for each positive sample.

Type
int

label_smoothing_rate

The smoothing factor applied to the labels.

Type
torch.Tensor

collate_fn

A function that can be used to collate data samples into batches (set to None by default).

Type
function, optional

train_data

num_entities

num_relations

neg_sample_ratio = None

label_smoothing_rate

collate_fn = None

__len__()

Returns the number of samples in the dataset.

__getitem__(*idx*)

Retrieves a single data sample from the dataset at the given index.

Parameters
idx (*int*) – The index of the sample to retrieve.

Returns

A tuple consisting of:

- x (torch.Tensor): The head and relation part of the triple.
- y_idx (torch.Tensor): The concatenated indices of the true object (tail entity) and the indices of the negative samples.
- y_vec (torch.Tensor): A vector containing the labels for the positive and negative samples, with label smoothing applied.

Return type
tuple

class `dicee.dataset_classes.`**`KvsSampleDataset`**(*train_set_idx: numpy.ndarray*, *entity_idxs*, *relation_idxs*, *form*, *store=None*, *neg_ratio=None*, *label_smoothing_rate: float = 0.0*)

Bases: `torch.utils.data.Dataset`

KvsSample a Dataset:

D:= {(x,y)_i}_i ^N, where
. x:(h,r) is a unique h in E and a relation r in R and . y in [0,1]^{|E|} is a binary label.

orall y_i =1 s.t. (h r E_i) in KG

At each mini-batch construction, we subsample(y), hence n
|new_y| << |E| new_y contains all 1's if sum(y)< neg_sample ratio new_y contains

train_set_idx
Indexed triples for the training.

entity_idxs
mapping.

relation_idxs
mapping.

form
?

store
?

label_smoothing_rate
?

torch.utils.data.Dataset

`train_data = None`

`train_target = None`

`neg_ratio = None`

`num_entities`

`label_smoothing_rate`

`collate_fn = None`

`max_num_of_classes`

`__len__`()

`__getitem__`(*idx*)

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`

An abstract class representing a `Dataset`.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite *__getitem__()*, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite *__len__()*, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

> **Note**
>
> `DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

neg_sample_ratio

train_triples

length

num_entities

num_relations

labels

train_set = []

__len__()

__getitem__(*idx*)

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`

Triple Dataset

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

train_set_idx
Indexed triples for the training.

entity_idxs
mapping.

relation_idxs
mapping.

form
?

store
?

label_smoothing_rate

collate_fn: batch:List[torch.IntTensor] Returns ——- torch.utils.data.Dataset

label_smoothing_rate

neg_sample_ratio

train_set

length

num_entities

num_relations

__len__()

__getitem__(*idx*)

collate_fn(*batch: List[torch.Tensor]*)

class dicee.dataset_classes.**CVDataModule**(*train_set_idx: numpy.ndarray*, *num_entities*, *num_relations*, *neg_sample_ratio*, *batch_size*, *num_workers*)

Bases: `pytorch_lightning.LightningDataModule`

Create a Dataset for cross validation

Parameters

- **train_set_idx** – Indexed triples for the training.
- **num_entities** – entity to index mapping.
- **num_relations** – relation to index mapping.
- **batch_size** – int
- **form** – ?
- **num_workers** – int for https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

Return type

?

train_set_idx

num_entities

num_relations

neg_sample_ratio

batch_size

num_workers

train_dataloader() → torch.utils.data.DataLoader

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:`~pytorch_lightning.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.

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
```

> **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.dataset_classes.**LiteralDataset**(*file_path: str*, *ent_idx: dict = None*, *normalization_type: str = 'z-norm'*, *sampling_ratio: float = None*, *loader_backend: str = 'pandas'*)

Bases: `torch.utils.data.Dataset`

Dataset for loading and processing literal data for training Literal Embedding model. This dataset handles the loading, normalization, and preparation of triples for training a literal embedding model.

Extends torch.utils.data.Dataset for supporting PyTorch dataloaders.

`train_file_path`

Path to the training data file.

Type
str

`normalization`

Type of normalization to apply ('z-norm', 'min-max', or None).

Type
str

`normalization_params`

Parameters used for normalization.

Type
dict

`sampling_ratio`

Fraction of the training set to use for ablations.

Type
float

`entity_to_idx`

Mapping of entities to their indices.

Type
dict

`num_entities`

Total number of entities.

Type
int

`data_property_to_idx`

Mapping of data properties to their indices.

Type
dict

`num_data_properties`

Total number of data properties.

Type
int

`loader_backend`

Backend to use for loading data ('pandas' or 'rdflib').

Type
str

train_file_path

loader_backend = 'pandas'

normalization_type = 'z-norm'

normalization_params

sampling_ratio = None

entity_to_idx = None

num_entities

__getitem__(*index*)

__len__()

static load_and_validate_literal_data(*file_path: str = None*, *loader_backend: str = 'pandas'*) → pandas.DataFrame

Loads and validates the literal data file. :param file_path: Path to the literal data file. :type file_path: str

Returns
DataFrame containing the loaded and validated data.

Return type
pd.DataFrame

static denormalize(*preds_norm*, *attributes*, *normalization_params*) → numpy.ndarray

Denormalizes the predictions based on the normalization type.

Args: preds_norm (np.ndarray): Normalized predictions to be denormalized. attributes (list): List of attributes corresponding to the predictions. normalization_params (dict): Dictionary containing normalization parameters for each attribute.

Returns
Denormalized predictions.

Return type
np.ndarray

dicee.eval_static_funcs

Functions

evaluate_link_prediction_performance(→ Dict)	
*evaluate_link_prediction_performance_with_*	
*evaluate_link_prediction_performance_with_*	
*evaluate_link_prediction_performance_with_* ...)	
evaluate_lp_bpe_k_vs_all(model, triples[, er_vocab, ...])	
evaluate_literal_prediction(kge_model[, ...])	Evaluates the trained literal prediction model on a test file.
evaluate_ensemble_link_prediction_performa Dict)	Evaluates link prediction performance of an ensemble of KGE models.

Module Contents

`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.eval_static_funcs.`**`evaluate_literal_prediction`**(*kge_model: dicee.knowledge_graph_embeddings.KGE*, *eval_file_path: str = None*, *store_lit_preds: bool = True*, *eval_literals: bool = True*, *loader_backend: str = 'pandas'*, *return_attr_error_metrics: bool = False*)

Evaluates the trained literal prediction model on a test file.

Parameters

- **`eval_file_path`** (*`str`*) – Path to the evaluation file.
- **`store_lit_preds`** (*`bool`*) – If True, stores the predictions in a CSV file.
- **`eval_literals`** (*`bool`*) – If True, evaluates the literal predictions and prints error metrics.
- **`loader_backend`** (*`str`*) – Backend for loading the dataset ('pandas' or 'rdflib').

Returns

DataFrame containing error metrics for each attribute if return_attr_error_metrics is True.

Return type

pd.DataFrame

Raises

- **RuntimeError** – If the kGE model does not have a trained literal model.
- **AssertionError** – If the kGE model is not an instance of KGE or if the test set has no valid entities or attributes.

dicee.eval_static_funcs.**evaluate_ensemble_link_prediction_performance**(*models*, *triples*, *er_vocab: Dict[Tuple, List]*, *weights: List[float] = None*, *batch_size: int = 512*, *weighted_averaging: bool = True*) → Dict

Evaluates link prediction performance of an ensemble of KGE models. :param models: List of KGE models (snapshots) :param triples: np.ndarray or list of lists, shape (N,3), all integer indices (head, rel, tail) :param er_vocab: Dict[Tuple, List]

Mapping (head_idx, rel_idx) → list of tail_idx to filter (incl. target).

Parameters

- **weights** – Optional[List[float]] Weights for model averaging. If None, use uniform (=simple mean).
- **batch_size** – int

Returns

dict of link prediction metrics (H@1, H@3, H@10, MRR)

dicee.evaluator

Classes

Evaluator	Evaluator class to evaluate KGE models in various downstream tasks

Module Contents

class dicee.evaluator.**Evaluator**(*args*, *is_continual_training=None*)

Evaluator class to evaluate KGE models in various downstream tasks

Arguments

re_vocab = None

er_vocab = None

ee_vocab = None

func_triple_to_bpe_representation = None

is_continual_training = None

num_entities = None

num_relations = None

args

report

during_training = False

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

Classes

Execute	A class for Training, Retraining and Evaluation a model.
ContinuousExecute	A subclass of Execute Class for retraining

Module Contents

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

distributed

args

is_continual_training = False

trainer = None

trained_model = None

knowledge_graph = None

report

evaluator = None

start_time = None

is_rank_zero() → bool

cleanup()

setup_executor() → None

create_and_store_kg()

load_from_memmap()

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

Classes

KG	Knowledge Graph

Module Contents

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_reciprocal: 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*, *separator: str = None*)

Knowledge Graph

dataset_dir = None

sparql_endpoint = None

path_single_kg = None

byte_pair_encoding = False

ordered_shaped_bpe_tokens = None

add_noise_rate = None

num_entities = None

num_relations = None

path_for_deserialization = None

add_reciprocal = None

eval_model = None

read_only_few = None

sample_triples_ratio = None

path_for_serialization = None

entity_to_idx = None

relation_to_idx = None

backend = 'pandas'

training_technique = None

idx_entity_to_bpe_shaped

enc

num_tokens

num_bpe_entities = None

padding = False

dummy_id

max_length_subword_tokens = None

train_set_target = None

target_dim = None

train_target_indices = None

ordered_bpe_entities = None

separator = None

description_of_input = None

describe() → None

property entities_str: List

property relations_str: List

exists(*h: str*, *r: str*, *t: str*)

__iter__()

__len__()

func_triple_to_bpe_representation(*triple: List[str]*)

dicee.knowledge_graph_embeddings

Classes

KGE	Knowledge Graph Embedding Class for interactive usage of pre-trained models

Module Contents

class dicee.knowledge_graph_embeddings.**KGE**(*path=None*, *url=None*, *construct_ensemble=False*, *model_name=None*)

Bases: *dicee.abstracts.BaseInteractiveKGE*, *dicee.abstracts.InteractiveQueryDecomposition*, *dicee.abstracts.BaseInteractiveTrainKGE*

Knowledge Graph Embedding Class for interactive usage of pre-trained models

__str__()

to(*device: str*) → None

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=''*)

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*, *batch_size=2*, *topk=1*, *return_indices=False*) → 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*, *batch_size=2*, *topk=1*, *return_indices=False*) → 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*, *batch_size=2*, *topk=1*, *return_indices=False*) → 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: str | List[str] = None, r: str | List[str] = None, t: str | List[str] = None, topk: int = 10, within: List[str] = None, batch_size: int = 1024*)

Predict missing item in a given triple.

Returns

- If you query a single (h, r, ?) or (?, r, t) or (h, ?, t), returns List[(item, score)]
- If you query a batch of B, returns List of B such lists.

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

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*)

predict_literals(*entity: List[str] | str = None*, *attribute: List[str] | str = None*, *denormalize_preds: bool = True*) → numpy.ndarray

Predicts literal values for given entities and attributes.

Parameters

- **entity** (*Union[List[str], str]*) – Entity or list of entities to predict literals for.

- **attribute** (*Union[List[str], str]*) – Attribute or list of attributes to predict literals for.
- **denormalize_preds** (*bool*) – If True, denormalizes the predictions.

Returns
Predictions for the given entities and attributes.

Return type
numpy ndarray

dicee.models

Submodules

dicee.models.adopt

Classes

ADOPT	Base class for all optimizers.

Functions

adopt(params, grads, exp_avgs, exp_avg_sqs, state_steps)	Functional API that performs ADOPT algorithm computation.

Module Contents

class dicee.models.adopt.**ADOPT**(*params: torch.optim.optimizer.ParamsT, lr: float | torch.Tensor = 0.001, betas: Tuple[float, float] = (0.9, 0.9999), eps: float = 1e-06, clip_lambda: Callable[[int], float] | None = lambda step: ..., weight_decay: float = 0.0, decouple: bool = False, *, foreach: bool | None = None, maximize: bool = False, capturable: bool = False, differentiable: bool = False, fused: bool | None = None*)

Bases: `torch.optim.optimizer.Optimizer`

Base class for all optimizers.

> **Warning**
>
> Parameters need to be specified as collections that have a deterministic ordering that is consistent between runs. Examples of objects that don't satisfy those properties are sets and iterators over values of dictionaries.

Parameters

- **params** (*iterable*) – an iterable of `torch.Tensor` s or `dict` s. Specifies what Tensors should be optimized.
- **defaults** – (dict): a dict containing default values of optimization options (used when a parameter group doesn't specify them).

clip_lambda

__setstate__(*state*)

step(*closure=None*)

Perform a single optimization step.

Parameters

closure (`Callable, optional`) – A closure that reevaluates the model and returns the loss.

`dicee.models.adopt.`**adopt**(*params: List[torch.Tensor]*, *grads: List[torch.Tensor]*, *exp_avgs: List[torch.Tensor]*, *exp_avg_sqs: List[torch.Tensor]*, *state_steps: List[torch.Tensor]*, *foreach: bool | None = None*, *capturable: bool = False*, *differentiable: bool = False*, *fused: bool | None = None*, *grad_scale: torch.Tensor | None = None*, *found_inf: torch.Tensor | None = None*, *has_complex: bool = False*, ***, *beta1: float*, *beta2: float*, *lr: float | torch.Tensor*, *clip_lambda: Callable[[int], float] | None*, *weight_decay: float*, *decouple: bool*, *eps: float*, *maximize: bool*)

Functional API that performs ADOPT algorithm computation.

dicee.models.base_model

Classes

`BaseKGELightning`	Base class for all neural network modules.
`BaseKGE`	Base class for all neural network modules.
`IdentityClass`	Base class for all neural network modules.

Module Contents

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

training_step_outputs = []

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()

```

(continues on next page)

(continued from previous page)

```
    # 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 **:param-ref:`~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 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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

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]*)

byte pair encoded neural link predictors

Parameters

----

init_params_with_sanity_checking()

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters
x

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

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

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple*)

get_sentence_representation(*x: torch.LongTensor*)

Parameters

- **(b** (*x shape*)
- **3**
- **t)**

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

Parameters
x (*B x 2 x T*)

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

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

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 them to be nested 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) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
```

(continues on next page)

(continued from previous page)

```
        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 also have their parameters converted 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.

args = None

__call__(*x*)

static forward(*x*)

dicee.models.clifford

Classes

Keci	Base class for all neural network modules.
CKeci	Without learning dimension scaling
DeCaL	Base class for all neural network modules.

Module Contents

class dicee.models.clifford.**Keci**(*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 them to be nested 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) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
```

(continues on next page)

(continued from previous page)

```
        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 also have their parameters converted 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.

name = 'Keci'

p

q

r

requires_grad_for_interactions = True

compute_sigma_pp(*hp, rp*)

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

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_{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.

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)

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

Multiplying a base vector with its scalar coefficient

clifford_multiplication(*h0*, *hp*, *hq*, *r0*, *rp*, *rq*)

Compute our CL multiplication

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

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 with (n,d) shape

returns

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

forward_k_vs_with_explicit(*x: torch.Tensor*)

k_vs_all_score(*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*)

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

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

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

Parameter

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

returns

- **a0** (*torch.FloatTensor with (n,k, m) shape*)
- **ap** (*torch.FloatTensor with (n,k, m, p) shape*)
- **aq** (*torch.FloatTensor with (n,k, m, q) shape*)

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

Parameter

x: torch.LongTensor with (n,2) shape

target_entity_idx: torch.LongTensor with (n, k) shape k denotes the selected number of examples.

rtype
torch.FloatTensor with (n, k) shape

score(*h*, *r*, *t*)

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.**CKeci**(*args*)

Bases: *Keci*

Without learning dimension scaling

name = 'CKeci'

requires_grad_for_interactions = False

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'DeCaL'

entity_embeddings

relation_embeddings

p

q

r

re

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

Parameter

x: torch.LongTensor with (n,) shape

rtype

torch.FloatTensor with (n) shape

cl_pqr(*a: torch.tensor*) → torch.tensor

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,

$$s0 = h_0 r_0 t_0 s1 = \sum_{i=1}^{p} h_i r_i t_0 s2 = \sum_{j=p+1}^{p+q} h_j r_j t_0 s3 = \sum_{i=1}^{q} (h_0 r_i t_i + h_i r_0 t_i) s4 = \sum_{i=p+1}^{p+q} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+1}^{p+q+r} (h_0 r_i t$$

and return:

$$sigma_0 t = \sigma_0 \cdot t_0 = s0 + s1 - s2 s3, s4 and s5$$

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

$$\sigma_p p = \sum_{i=1}^{p-1} \sum_{i'=i+1}^{p} (h_i r_{i'} - h_{i'} r_i)(models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'}$$

For different base vector interactions, we have

$$\sigma_p q = \sum_{i=1}^{p} \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i)(interactionsn between e_i and e_j for 1 <= i <= p and p + 1 <= j <= p + q) \sigma_p r = \sum$$

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

Kvsall training

(1) Retrieve real-valued embedding vectors for heads and relations

(2) Construct head entity and relation embeddings according to Cl_{p,q, r}(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,) 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*)

Compute .. math:

```
\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*)

Compute

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

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*)

Compute

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

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

Classes

ConEx	Convolutional ComplEx Knowledge Graph Embeddings
AConEx	Additive Convolutional ComplEx Knowledge Graph Embeddings
ComplEx	Base class for all neural network modules.

Module Contents

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

Bases: *dicee.models.base_model.BaseKGE*

Convolutional ComplEx Knowledge Graph Embeddings

name = 'ConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

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

Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

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

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

Parameters

x

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

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional ComplEx Knowledge Graph Embeddings

name = 'AConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

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

Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

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

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

Parameters

x

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

class dicee.models.complex.**ComplEx**(*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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'ComplEx'

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

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

Parameters

- **emb_h**
- **emb_r**
- **emb_E**

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

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

dicee.models.dualE

Classes

DualE	Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)

Module Contents

class dicee.models.dualE.**DualE**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)

name = 'DualE'

entity_embeddings

relation_embeddings

num_ent = None

kvsall_score(*e_1_h, e_2_h, e_3_h, e_4_h, e_5_h, e_6_h, e_7_h, e_8_h, e_1_t, e_2_t, e_3_t, e_4_t, e_5_t, e_6_t, e_7_t, e_8_t, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8*) → torch.tensor

KvsAll scoring function

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

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

Negative Sampling forward pass:

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_k_vs_all(*x*)

KvsAll forward pass

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

T(*x: torch.tensor*) → torch.tensor

Transpose function

Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)

dicee.models.ensemble

Classes

EnsembleKGE

Module Contents

class dicee.models.ensemble.**EnsembleKGE**(*models: list = None*, *seed_model=None*, *pretrained_models: List = None*)

name

train_mode = True

args

named_children()

property example_input_array

parameters()

modules()

__iter__()

__len__()

eval()

to(*device*)

state_dict()

Return the state dict of the ensemble.

load_state_dict(*state_dict*, *strict=True*)

Load the state dict into the ensemble.

mem_of_model()

__call__(*x_batch*)

step()

get_embeddings()

__str__()

dicee.models.function_space

Classes

FMult	Learning Knowledge Neural Graphs
GFMult	Learning Knowledge Neural Graphs
FMult2	Learning Knowledge Neural Graphs
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:

Module Contents

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

Bases: *dicee.models.base_model.BaseKGE*

Learning Knowledge Neural Graphs

name = 'FMult'

entity_embeddings

relation_embeddings

k

num_sample = 50

gamma

roots

weights

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

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

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

Parameters

x

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

Bases: *dicee.models.base_model.BaseKGE*

Learning Knowledge Neural Graphs

name = 'GFMult'

entity_embeddings

relation_embeddings

k

num_sample = 250

roots

weights

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

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

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

Parameters

x

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

Bases: *dicee.models.base_model.BaseKGE*

Learning Knowledge Neural Graphs

name = 'FMult2'

n_layers = 3

k

n = 50

score_func = 'compositional'

discrete_points

entity_embeddings

relation_embeddings

build_func(*Vec*)

build_chain_funcs(*list_Vec*)

compute_func(*W*, *b*, *x*) → torch.FloatTensor

function(*list_W*, *list_b*)

trapezoid(*list_W*, *list_b*)

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

Parameters

x

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

name = 'LFMult1'

entity_embeddings

relation_embeddings

forward_triples (*idx_triple*)

Parameters

x

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.

name = 'LFMult'

entity_embeddings

relation_embeddings

degree

m

x_values

forward_triples (*idx_triple*)

Parameters

x

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.literal

Classes

LiteralEmbeddings	A model for learning and predicting numerical literals using pre-trained KGE.

Module Contents

class dicee.models.literal.**LiteralEmbeddings**(*num_of_data_properties: int*, *embedding_dims: int*, *entity_embeddings: torch.tensor*, *dropout: float = 0.3*, *gate_residual=True*, *freeze_entity_embeddings=True*)

Bases: `torch.nn.Module`

A model for learning and predicting numerical literals using pre-trained KGE.

num_of_data_properties

Number of data properties (attributes).

Type
int

embedding_dims

Dimension of the embeddings.

Type
int

entity_embeddings

Pre-trained entity embeddings.

Type
torch.tensor

dropout

Dropout rate for regularization.

Type

float

gate_residual

Whether to use gated residual connections.

Type

bool

freeze_entity_embeddings

Whether to freeze the entity embeddings during training.

Type

bool

embedding_dim

num_of_data_properties

hidden_dim

gate_residual = True

freeze_entity_embeddings = True

entity_embeddings

data_property_embeddings

fc

fc_out

dropout

gated_residual_proj

layer_norm

forward(*entity_idx*, *attr_idx*)

Parameters

- **entity_idx** (*Tensor*) – Entity indices (batch).
- **attr_idx** (*Tensor*) – Attribute (Data property) indices (batch).

Returns

scalar predictions.

Return type

Tensor

property device

dicee.models.octonion

Classes

OMult	Base class for all neural network modules.
ConvO	Base class for all neural network modules.
AConvO	Additive Convolutional Octonion Knowledge Graph Embeddings

Functions

octonion_mul(*, O_1, O_2)	
octonion_mul_norm(*, O_1, O_2)	

Module Contents

dicee.models.octonion.**octonion_mul**(*, *O_1*, *O_2*)

dicee.models.octonion.**octonion_mul_norm**(*, *O_1*, *O_2*)

class dicee.models.octonion.**OMult**(*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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'OMult'

static octonion_normalizer(*emb_rel_e0*, *emb_rel_e1*, *emb_rel_e2*, *emb_rel_e3*, *emb_rel_e4*, *emb_rel_e5*, *emb_rel_e6*, *emb_rel_e7*)

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

k_vs_all_score(*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*)

forward_k_vs_all(*x*)

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples,i.e., [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|)

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

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'ConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

static octonion_normalizer(*emb_rel_e0*, *emb_rel_e1*, *emb_rel_e2*, *emb_rel_e3*, *emb_rel_e4*, *emb_rel_e5*, *emb_rel_e6*, *emb_rel_e7*)

residual_convolution(*O_1*, *O_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Octonion Knowledge Graph Embeddings

name = 'AConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

static octonion_normalizer(*emb_rel_e0*, *emb_rel_e1*, *emb_rel_e2*, *emb_rel_e3*, *emb_rel_e4*, *emb_rel_e5*, *emb_rel_e6*, *emb_rel_e7*)

residual_convolution(*O_1*, *O_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

dicee.models.pykeen_models

Classes

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

Module Contents

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

Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Pykeen_HolE: Pykeen_HolE: Pykeen_TransD: Pykeen_TransE: Pykeen_TransF: Pykeen_TransH: Pykeen_TransR:

model_kwargs

name

model

loss_history = []

args

entity_embeddings = None

relation_embeddings = None

forward_k_vs_all(*x: torch.LongTensor*)

=> 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

=> 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*)

dicee.models.quaternion

Classes

QMult	Base class for all neural network modules.
ConvQ	Convolutional Quaternion Knowledge Graph Embeddings
AConvQ	Additive Convolutional Quaternion Knowledge Graph Embeddings

Functions

quaternion_mul_with_unit_norm(*, Q_1, Q_2)	

Module Contents

dicee.models.quaternion.**quaternion_mul_with_unit_norm**(*, *Q_1*, *Q_2*)

class dicee.models.quaternion.**QMult**(*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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'QMult'

explicit = True

quaternion_multiplication_followed_by_inner_product (*h*, *r*, *t*)

Parameters

- **h** – shape: (**batch_dims*, dim) The head representations.
- **r** – shape: (**batch_dims*, dim) The head representations.
- **t** – shape: (**batch_dims*, dim) The tail representations.

Returns

Triple scores.

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

Normalize the length of relation vectors, if the forward constraint has not been applied yet.

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

Parameters

x – The vector.

Returns

The normalized vector.

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

k_vs_all_score (*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*)

Parameters

- **bpe_head_ent_emb**
- **bpe_rel_ent_emb**
- **E**

forward_k_vs_all (*x*)

Parameters

x

forward_k_vs_sample(*x*, *target_entity_idx*)

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples,i.e., [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|)

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

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings

name = 'ConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution(*Q_1*, *Q_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Quaternion Knowledge Graph Embeddings

name = 'AConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution(*Q_1*, *Q_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

dicee.models.real

Classes

DistMult	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
TransE	Translating Embeddings for Modeling
Shallom	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
Pyke	A Physical Embedding Model for Knowledge Graphs

Module Contents

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

Bases: *dicee.models.base_model.BaseKGE*

Embedding Entities and Relations for Learning and Inference in Knowledge Bases https://arxiv.org/abs/1412.6575

name = 'DistMult'

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

Parameters

- **emb_h**
- **emb_r**
- **emb_E**

forward_k_vs_all(*x: torch.LongTensor*)

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

score(*h*, *r*, *t*)

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

Bases: *dicee.models.base_model.BaseKGE*

Translating Embeddings for Modeling Multi-relational Data https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf

name = 'TransE'

margin = 4

score(*head_ent_emb*, *rel_ent_emb*, *tail_ent_emb*)

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

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

Bases: *dicee.models.base_model.BaseKGE*

A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)

name = 'Shallom'

shallom

get_embeddings() → Tuple[numpy.ndarray, None]

forward_k_vs_all(*x*) → torch.FloatTensor

forward_triples(*x*) → torch.FloatTensor

Parameters

x

Returns

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

Bases: *dicee.models.base_model.BaseKGE*

A Physical Embedding Model for Knowledge Graphs

name = 'Pyke'

dist_func

margin = 1.0

forward_triples(*x: torch.LongTensor*)

Parameters

x

dicee.models.static_funcs

Functions

quaternion_mul(→ Tuple[torch.Tensor, torch.Tensor, ...)	Perform quaternion multiplication

Module Contents

dicee.models.static_funcs.**quaternion_mul**(*, *Q_1*, *Q_2*)
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Perform quaternion multiplication :param Q_1: :param Q_2: :return:

dicee.models.transformers

Full definition of a GPT Language Model, all of it in this single file. References: 1) the official GPT-2 TensorFlow implementation released by OpenAI: https://github.com/openai/gpt-2/blob/master/src/model.py 2) huggingface/transformers PyTorch implementation: https://github.com/huggingface/transformers/blob/main/src/transformers/models/gpt2/modeling_gpt2.py

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.

Module Contents

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

`name = 'BytE'`

`config`

`temperature = 0.5`

`topk = 2`

`transformer`

`lm_head`

`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

weight

bias

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 them to be nested in a tree structure. You can assign the submodules as regular attributes:

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

(continues on next page)

(continued from previous page)

```
class Model(nn.Module):
    def __init__(self) -> None:
        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 also have their parameters converted 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.

c_attn

c_proj

attn_dropout

resid_dropout

n_head

n_embd

dropout

flash = True

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 them to be nested 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) -> None:
```

(continues on next page)

(continued from previous page)

```
        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 also have their parameters converted 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.

c_fc

gelu

c_proj

dropout

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

ln_1

attn

ln_2

mlp

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

config

transformer

lm_head

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

Classes

ADOPT	Base class for all optimizers.
BaseKGELightning	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
IdentityClass	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
DistMult	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
TransE	Translating Embeddings for Modeling
Shallom	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
Pyke	A Physical Embedding Model for Knowledge Graphs
BaseKGE	Base class for all neural network modules.
ConEx	Convolutional ComplEx Knowledge Graph Embeddings
AConEx	Additive Convolutional ComplEx Knowledge Graph Embeddings
ComplEx	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
IdentityClass	Base class for all neural network modules.

continues on next page

Table 1 – continued from previous page

QMult	Base class for all neural network modules.
ConvQ	Convolutional Quaternion Knowledge Graph Embeddings
AConvQ	Additive Convolutional Quaternion Knowledge Graph Embeddings
BaseKGE	Base class for all neural network modules.
IdentityClass	Base class for all neural network modules.
OMult	Base class for all neural network modules.
ConvO	Base class for all neural network modules.
AConvO	Additive Convolutional Octonion Knowledge Graph Embeddings
Keci	Base class for all neural network modules.
CKeci	Without learning dimension scaling
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	Learning Knowledge Neural Graphs
GFMult	Learning Knowledge Neural Graphs
FMult2	Learning Knowledge Neural Graphs
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:
DualE	Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)

Functions

quaternion_mul(→ Tuple[torch.Tensor, torch.Tensor, ...)	Perform quaternion multiplication
quaternion_mul_with_unit_norm(*, Q_1, Q_2)	
octonion_mul(*, O_1, O_2)	
octonion_mul_norm(*, O_1, O_2)	

Package Contents

class dicee.models.**ADOPT**(*params: torch.optim.optimizer.ParamsT, lr: float | torch.Tensor = 0.001, betas: Tuple[float, float] = (0.9, 0.9999), eps: float = 1e-06, clip_lambda: Callable[[int], float] | None = lambda step: ..., weight_decay: float = 0.0, decouple: bool = False, *, foreach: bool | None = None, maximize: bool = False, capturable: bool = False, differentiable: bool = False, fused: bool | None = None*)

Bases: `torch.optim.optimizer.Optimizer`

Base class for all optimizers.

> **Warning**
>
> Parameters need to be specified as collections that have a deterministic ordering that is consistent between runs. Examples of objects that don't satisfy those properties are sets and iterators over values of dictionaries.

Parameters

- **params** (*iterable*) – an iterable of `torch.Tensor` s or `dict` s. Specifies what Tensors should be optimized.
- **defaults** – (dict): a dict containing default values of optimization options (used when a parameter group doesn't specify them).

clip_lambda

__setstate__(*state*)

step(*closure=None*)

Perform a single optimization step.

Parameters

closure (*Callable, optional*) – A closure that reevaluates the model and returns the loss.

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

training_step_outputs = []

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 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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

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]*)

byte pair encoded neural link predictors

Parameters

----

init_params_with_sanity_checking()

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters
x

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

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

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple*)

get_sentence_representation(*x: torch.LongTensor*)

Parameters

- **(b** (*x shape*)
- **3**
- **t)**

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

Parameters
x (*B x 2 x T*)

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

class dicee.models.**IdentityClass**(*args=None*)

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args = None

__call__(*x*)

static forward(*x*)

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args

embedding_dim = None

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

`loss`

`selected_optimizer = None`

`normalizer_class = None`

`normalize_head_entity_embeddings`

`normalize_relation_embeddings`

`normalize_tail_entity_embeddings`

`hidden_normalizer`

`param_init`

`input_dp_ent_real`

`input_dp_rel_real`

`hidden_dropout`

`loss_history = []`

`byte_pair_encoding`

`max_length_subword_tokens`

`block_size`

`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]*)

byte pair encoded neural link predictors

Parameters

----

init_params_with_sanity_checking()

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters

x

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

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

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple*)

get_sentence_representation(*x: torch.LongTensor*)

Parameters

- **(b** (*x shape*)
- **3**
- **t)**

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

Parameters

x (*B x 2 x T*)

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

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

Bases: *dicee.models.base_model.BaseKGE*

Embedding Entities and Relations for Learning and Inference in Knowledge Bases https://arxiv.org/abs/1412.6575

name = 'DistMult'

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

Parameters

- **emb_h**
- **emb_r**

- **emb_E**

forward_k_vs_all(*x: torch.LongTensor*)

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

score(*h*, *r*, *t*)

class dicee.models.**TransE**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Translating Embeddings for Modeling Multi-relational Data https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf

name = 'TransE'

margin = 4

score(*head_ent_emb*, *rel_ent_emb*, *tail_ent_emb*)

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

class dicee.models.**Shallom**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)

name = 'Shallom'

shallom

get_embeddings() → Tuple[numpy.ndarray, None]

forward_k_vs_all(*x*) → torch.FloatTensor

forward_triples(*x*) → torch.FloatTensor

Parameters

x

Returns

class dicee.models.**Pyke**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

A Physical Embedding Model for Knowledge Graphs

name = 'Pyke'

dist_func

margin = 1.0

forward_triples(*x: torch.LongTensor*)

Parameters

x

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

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]*)

byte pair encoded neural link predictors

Parameters

----

init_params_with_sanity_checking()

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters

x

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

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

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple*)

get_sentence_representation(*x: torch.LongTensor*)

Parameters

- **(b** (*x shape*)
- **3**
- **t)**

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

Parameters

x (*B x 2 x T*)

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

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

Bases: *dicee.models.base_model.BaseKGE*

Convolutional ComplEx Knowledge Graph Embeddings

name = 'ConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

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

Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

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

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

Parameters

x

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

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional ComplEx Knowledge Graph Embeddings

name = 'AConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

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

Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

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

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

Parameters

x

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

class dicee.models.**ComplEx**(*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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'ComplEx'

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

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

Parameters

- **emb_h**
- **emb_r**
- **emb_E**

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

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

dicee.models.**quaternion_mul**(*, *Q_1*, *Q_2*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Perform quaternion multiplication :param Q_1: :param Q_2: :return:

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

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]*)

byte pair encoded neural link predictors

Parameters

----

init_params_with_sanity_checking()

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters

x

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

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

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple*)

get_sentence_representation(*x: torch.LongTensor*)

Parameters

- **(b** (*x shape*)
- **3**
- **t)**

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

Parameters

x (*B x 2 x T*)

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

class dicee.models.**IdentityClass**(*args=None*)

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args = None

__call__(*x*)

static forward(*x*)

dicee.models.**quaternion_mul_with_unit_norm**(*, *Q_1*, *Q_2*)

class dicee.models.**QMult**(*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 them to be nested in a tree structure. You can assign the submodules as regular attributes:

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


class Model(nn.Module):
    def __init__(self) -> None:
        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 also have their parameters converted 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.

name = 'QMult'

explicit = True

quaternion_multiplication_followed_by_inner_product(*h*, *r*, *t*)

Parameters

- **h** – shape: (**batch_dims*, dim) The head representations.
- **r** – shape: (**batch_dims*, dim) The head representations.
- **t** – shape: (**batch_dims*, dim) The tail representations.

Returns

Triple scores.

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

Normalize the length of relation vectors, if the forward constraint has not been applied yet.

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

Parameters

x – The vector.

Returns

The normalized vector.

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

k_vs_all_score(*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*)

Parameters

- **bpe_head_ent_emb**
- **bpe_rel_ent_emb**
- **E**

forward_k_vs_all(*x*)

Parameters

x

forward_k_vs_sample(*x*, *target_entity_idx*)

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples,i.e., [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|)

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

Bases: `dicee.models.base_model.BaseKGE`

Convolutional Quaternion Knowledge Graph Embeddings

name = 'ConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution(*Q_1*, *Q_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Quaternion Knowledge Graph Embeddings

name = 'AConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution(*Q_1*, *Q_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

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]*)

byte pair encoded neural link predictors

Parameters

----

init_params_with_sanity_checking()

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters

x

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

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

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple*)

get_sentence_representation(*x: torch.LongTensor*)

Parameters

- **(b** (*x shape*)
- **3**
- **t)**

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

Parameters

x (*B x 2 x T*)

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

class dicee.models.**IdentityClass**(*args=None*)

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args = None

__call__(*x*)

static forward(*x*)

dicee.models.**octonion_mul**(**, O_1, O_2*)

dicee.models.**octonion_mul_norm**(**, O_1, O_2*)

class dicee.models.**OMult**(*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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'OMult'

static octonion_normalizer(*emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4, emb_rel_e5, emb_rel_e6, emb_rel_e7*)

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

k_vs_all_score(*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

forward_k_vs_all(*x*)

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples,i.e., [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|)

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

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 them to be nested 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):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self) -> None:
        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 also have their parameters converted 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.

name = 'ConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

static octonion_normalizer(*emb_rel_e0*, *emb_rel_e1*, *emb_rel_e2*, *emb_rel_e3*, *emb_rel_e4*, *emb_rel_e5*, *emb_rel_e6*, *emb_rel_e7*)

residual_convolution(*O_1*, *O_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Octonion Knowledge Graph Embeddings

name = 'AConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

static octonion_normalizer(*emb_rel_e0*, *emb_rel_e1*, *emb_rel_e2*, *emb_rel_e3*, *emb_rel_e4*, *emb_rel_e5*, *emb_rel_e6*, *emb_rel_e7*)

residual_convolution(*O_1*, *O_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

class dicee.models.**Keci**(*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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'Keci'

p

q

r

requires_grad_for_interactions = True

compute_sigma_pp(*hp*, *rp*)

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

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_{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.

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)

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

Multiplying a base vector with its scalar coefficient

clifford_multiplication(*h0*, *hp*, *hq*, *r0*, *rp*, *rq*)

Compute our CL multiplication

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

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 with (n,d) shape

returns

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

forward_k_vs_with_explicit(*x: torch.Tensor*)

k_vs_all_score(*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*)

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

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

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

Parameter

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

returns

- **a0** (*torch.FloatTensor with (n,k, m) shape*)
- **ap** (*torch.FloatTensor with (n,k, m, p) shape*)
- **aq** (*torch.FloatTensor with (n,k, m, q) shape*)

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

Parameter

x: torch.LongTensor with (n,2) shape

target_entity_idx: torch.LongTensor with (n, k) shape k denotes the selected number of examples.

rtype
torch.FloatTensor with (n, k) shape

score(*h*, *r*, *t*)

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

Parameter

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

rtype
torch.FloatTensor with (n) shape

class dicee.models.**CKeci**(*args*)

Bases: *Keci*

Without learning dimension scaling

name = 'CKeci'

requires_grad_for_interactions = False

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 them to be nested 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) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
```

(continues on next page)

(continued from previous page)

```
        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 also have their parameters converted 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.

name = 'DeCaL'

entity_embeddings

relation_embeddings

p

q

r

re

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

Parameter

x: torch.LongTensor with (n,) shape

rtype

torch.FloatTensor with (n) shape

cl_pqr(*a: torch.tensor*) → torch.tensor

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,

$$s0 = h_0 r_0 t_0 s1 = \sum_{i=1}^{p} h_i r_i t_0 s2 = \sum_{j=p+1}^{p+q} h_j r_j t_0 s3 = \sum_{i=1}^{q} (h_0 r_i t_i + h_i r_0 t_i) s4 = \sum_{i=p+1}^{p+q} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+1}^{p+q+r} (h_0 r_i t$$

and return:

$$sigma_0 t = \sigma_0 \cdot t_0 = s0 + s1 - s2s3, s4ands5$$

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

$$\sigma_p p = \sum_{i=1}^{p-1} \sum_{i'=i+1}^{p} (h_i r_{i'} - h_{i'} r_i) (modelstheinteractionsbetweene_i ande'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'}$$

For different base vector interactions, we have

$$\sigma_p q = \sum_{i=1}^{p} \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) (interactionsnbetweene_i ande_j for 1 <= i <= pandp + 1 <= j <= p + q) \sigma_p r = \sum$$

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

Kvsall training

(1) Retrieve real-valued embedding vectors for heads and relations

(2) Construct head entity and relation embeddings according to Cl_{p,q, r}(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,) 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*)

Compute .. math:

```
\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

$$\sigma^*_{q,q} = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (x_j y_{j'} - x_{j'} y_j) Eq.16$$

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*)

$$\sigma^*_{r,r} = \sum_{k=p+q+1}^{p+q+r-1} \sum_{k'=k+1}^{p} (x_k y_{k'} - x_{k'} y_k)$$

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

Compute

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

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*)

Compute

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

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

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]*)
byte pair encoded neural link predictors

Parameters
: **-------**

init_params_with_sanity_checking()

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters
: **x**

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

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

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple*)

get_sentence_representation(*x: torch.LongTensor*)

Parameters

- **(b** (*x shape*)
- **3**
- **t)**

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

Parameters
: **x** (*B x 2 x T*)

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

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

A class for using knowledge graph embedding models implemented in Pykeen

Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Pykeen_HolE: Pykeen_HolE: Pykeen_TransD: Pykeen_TransE: Pykeen_TransF: Pykeen_TransH: Pykeen_TransR:

model_kwargs

name

model

loss_history = []

args

entity_embeddings = None

relation_embeddings = None

forward_k_vs_all (*x: torch.LongTensor*)

=> 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

=> 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*)

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 them to be nested 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):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self) -> None:
        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 also have their parameters converted 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.

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

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]*)

byte pair encoded neural link predictors

Parameters

----

init_params_with_sanity_checking()

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters

x

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

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

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple*)

get_sentence_representation(*x: torch.LongTensor*)

Parameters

- **(b** (*x shape*)
- **3**
- **t)**

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

Parameters

x (*B x 2 x T*)

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

class dicee.models.**FMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Learning Knowledge Neural Graphs

name = 'FMult'

entity_embeddings

relation_embeddings

k

num_sample = 50

gamma

roots

weights

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

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

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

Parameters

x

class dicee.models.**GFMult**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Learning Knowledge Neural Graphs

name = 'GFMult'

entity_embeddings

relation_embeddings

k

num_sample = 250

roots

weights

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

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

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

Parameters

x

class dicee.models.**FMult2**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Learning Knowledge Neural Graphs

name = 'FMult2'

n_layers = 3

k

n = 50

score_func = 'compositional'

discrete_points

entity_embeddings

relation_embeddings

build_func(*Vec*)

build_chain_funcs(*list_Vec*)

compute_func(*W*, *b*, *x*) → torch.FloatTensor

function(*list_W*, *list_b*)

trapezoid(*list_W*, *list_b*)

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

Parameters

x

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

name = 'LFMult1'

entity_embeddings

relation_embeddings

forward_triples(*idx_triple*)

Parameters

x

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.

name = 'LFMult'

entity_embeddings

relation_embeddings

degree

m

x_values

forward_triples(*idx_triple*)

Parameters

x

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.models.**DualE**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)

name = 'DualE'

entity_embeddings

relation_embeddings

num_ent = None

kvsall_score(*e_1_h*, *e_2_h*, *e_3_h*, *e_4_h*, *e_5_h*, *e_6_h*, *e_7_h*, *e_8_h*, *e_1_t*, *e_2_t*, *e_3_t*, *e_4_t*, *e_5_t*, *e_6_t*, *e_7_t*, *e_8_t*, *r_1*, *r_2*, *r_3*, *r_4*, *r_5*, *r_6*, *r_7*, *r_8*) → torch.tensor

KvsAll scoring function

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

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

Negative Sampling forward pass:

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_k_vs_all(*x*)

KvsAll forward pass

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

T(*x: torch.tensor*) → torch.tensor

Transpose function

Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)

dicee.query_generator

Classes

QueryGenerator

Module Contents

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*)

train_path

val_path

test_path

gen_valid = False

gen_test = True

seed = 1

max_ans_num = 1000000.0

mode

ent2id = None

rel2id: Dict = None

ent_in: Dict

ent_out: Dict

query_name_to_struct

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.read_preprocess_save_load_kg

Submodules

dicee.read_preprocess_save_load_kg.preprocess

Classes

PreprocessKG	Preprocess the data in memory

Module Contents

class dicee.read_preprocess_save_load_kg.preprocess.**PreprocessKG**(*kg*)

Preprocess the data in memory

kg

start() → None

Preprocess train, valid and test datasets stored in knowledge graph instance

Parameter

rtype
None

preprocess_with_byte_pair_encoding()

preprocess_with_byte_pair_encoding_with_padding() → None

preprocess_with_pandas() → None

Preprocess train, valid and test datasets stored in knowledge graph instance with pandas

(1) Add recipriocal or noisy triples

(2) Construct vocabulary

(3) Index datasets

Parameter

rtype
None

preprocess_with_polars() → None

sequential_vocabulary_construction() → None

(1) Read input data into memory

(2) Remove triples with a condition

(3) **Serialize vocabularies in a pandas dataframe where**
=> the index is integer and => a single column is string (e.g. URI)

dicee.read_preprocess_save_load_kg.read_from_disk

Classes

ReadFromDisk	Read the data from disk into memory

Module Contents

class dicee.read_preprocess_save_load_kg.read_from_disk.**ReadFromDisk**(*kg*)

Read the data from disk into memory

kg

start() → None

Read a knowledge graph from disk into memory

Data will be available at the train_set, test_set, valid_set attributes.

Parameter

None

rtype

None

add_noisy_triples_into_training()

dicee.read_preprocess_save_load_kg.save_load_disk

Classes

LoadSaveToDisk

Module Contents

class dicee.read_preprocess_save_load_kg.save_load_disk.**LoadSaveToDisk**(*kg*)

kg

save()

load()

dicee.read_preprocess_save_load_kg.util

Functions

polars_dataframe_indexer(→ polars.DataFrame)	Replaces 'subject', 'relation', and 'object' columns in the input Polars DataFrame with their corresponding index values
pandas_dataframe_indexer(→ pandas.DataFrame)	Replaces 'subject', 'relation', and 'object' columns in the input Pandas DataFrame with their corresponding index values
apply_reciprical_or_noise(add_reciprical, eval_model)	
timeit(func)	
read_with_polars(→ polars.DataFrame)	Load and Preprocess via Polars
read_with_pandas(data_path[, read_only_few, ...])	
read_from_disk(→ Tuple[polars.DataFrame, pandas.DataFrame])	
read_from_triple_store([endpoint])	Read triples from triple store into pandas dataframe
get_er_vocab(data[, file_path])	
get_re_vocab(data[, file_path])	
get_ee_vocab(data[, file_path])	
create_constraints(triples[, file_path])	
load_with_pandas(→ None)	Deserialize data
save_numpy_ndarray(*, data, file_path)	
load_numpy_ndarray(*, file_path)	
save_pickle(*, data[, file_path])	
load_pickle(*[, file_path])	
create_recipriocal_triples(x)	Add inverse triples into dask dataframe
dataset_sanity_checking(→ None)	

Module Contents

`dicee.read_preprocess_save_load_kg.util.`**`polars_dataframe_indexer`**`(`
df_polars: polars.DataFrame, *idx_entity: polars.DataFrame*, *idx_relation: polars.DataFrame*`)`
→ polars.DataFrame

Replaces 'subject', 'relation', and 'object' columns in the input Polars DataFrame with their corresponding index values from the entity and relation index DataFrames.

This function processes the DataFrame in three main steps: 1. Replace the 'relation' values with the corresponding index from *idx_relation*. 2. Replace the 'subject' values with the corresponding index from *idx_entity*. 3. Replace the 'object' values with the corresponding index from *idx_entity*.

Parameters:

df_polars
[polars.DataFrame] The input Polars DataFrame containing columns: 'subject', 'relation', and 'object'.

idx_entity
[polars.DataFrame] A Polars DataFrame that contains the mapping between entity names and their corresponding indices. Must have columns: 'entity' and 'index'.

idx_relation
[polars.DataFrame] A Polars DataFrame that contains the mapping between relation names and their corresponding indices. Must have columns: 'relation' and 'index'.

Returns:

polars.DataFrame
A DataFrame with the 'subject', 'relation', and 'object' columns replaced by their corresponding indices.

Example Usage:

```
>>> df_polars = pl.DataFrame({
        "subject": ["Alice", "Bob", "Charlie"],
        "relation": ["knows", "works_with", "lives_in"],
        "object": ["Dave", "Eve", "Frank"]
    })
>>> idx_entity = pl.DataFrame({
        "entity": ["Alice", "Bob", "Charlie", "Dave", "Eve", "Frank"],
        "index": [0, 1, 2, 3, 4, 5]
    })
>>> idx_relation = pl.DataFrame({
        "relation": ["knows", "works_with", "lives_in"],
        "index": [0, 1, 2]
    })
>>> polars_dataframe_indexer(df_polars, idx_entity, idx_relation)
```

Steps:

1. Join the input DataFrame *df_polars* on the 'relation' column with *idx_relation* to replace the relations with their indices.
2. Join on 'subject' to replace it with the corresponding entity index using a left join on *idx_entity*.
3. Join on 'object' to replace it with the corresponding entity index using a left join on *idx_entity*.
4. Select only the 'subject', 'relation', and 'object' columns to return the final result.

`dicee.read_preprocess_save_load_kg.util.`**`pandas_dataframe_indexer`**`(`*df_pandas: pandas.DataFrame*, *idx_entity: pandas.DataFrame*, *idx_relation: pandas.DataFrame*`)` → pandas.DataFrame

Replaces 'subject', 'relation', and 'object' columns in the input Pandas DataFrame with their corresponding index values from the entity and relation index DataFrames.

Parameters:

df_pandas
: [pd.DataFrame] The input Pandas DataFrame containing columns: 'subject', 'relation', and 'object'.

idx_entity
: [pd.DataFrame] A Pandas DataFrame that contains the mapping between entity names and their corresponding indices. Must have columns: 'entity' and 'index'.

idx_relation
: [pd.DataFrame] A Pandas DataFrame that contains the mapping between relation names and their corresponding indices. Must have columns: 'relation' and 'index'.

Returns:

pd.DataFrame
: A DataFrame with the 'subject', 'relation', and 'object' columns replaced by their corresponding indices.

`dicee.read_preprocess_save_load_kg.util.`**`apply_reciprical_or_noise`**(*add_reciprical: bool*, *eval_model: str*, *df: object = None*, *info: str = None*)

(1) Add reciprocal triples (2) Add noisy triples

`dicee.read_preprocess_save_load_kg.util.`**`timeit`**(*func*)

`dicee.read_preprocess_save_load_kg.util.`**`read_with_polars`**(*data_path*, *read_only_few: int = None*, *sample_triples_ratio: float = None*, *separator: str = None*) → polars.DataFrame

Load and Preprocess via Polars

`dicee.read_preprocess_save_load_kg.util.`**`read_with_pandas`**(*data_path*, *read_only_few: int = None*, *sample_triples_ratio: float = None*, *separator: str = 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*, *separator: str = None*) → Tuple[polars.DataFrame, pandas.DataFrame]

`dicee.read_preprocess_save_load_kg.util.`**`read_from_triple_store`**(*endpoint: str = None*)

Read triples from triple store into pandas dataframe

`dicee.read_preprocess_save_load_kg.util.`**`get_er_vocab`**(*data*, *file_path: str = None*)

`dicee.read_preprocess_save_load_kg.util.`**`get_re_vocab`**(*data*, *file_path: str = None*)

`dicee.read_preprocess_save_load_kg.util.`**`get_ee_vocab`**(*data*, *file_path: str = None*)

`dicee.read_preprocess_save_load_kg.util.`**`create_constraints`**(*triples*, *file_path: str = None*)

(1) Extract domains and ranges of relations

(2) Store a mapping from relations to entities that are outside of the domain and range. Crete constrainted entities based on the range of relations :param triples: :return: Tuple[dict, dict]

`dicee.read_preprocess_save_load_kg.util.`**`load_with_pandas`**(*self*) → None

Deserialize data

`dicee.read_preprocess_save_load_kg.util.`**`save_numpy_ndarray`**(**, *data: numpy.ndarray*, *file_path: str*) dicee.read_preprocess_save_load_kg.util.**load_numpy_ndarray**(*, *file_path: str*)

dicee.read_preprocess_save_load_kg.util.**save_pickle**(*, *data: object*, *file_path=str*)

dicee.read_preprocess_save_load_kg.util.**load_pickle**(*, *file_path=str*)

dicee.read_preprocess_save_load_kg.util.**create_recipriocal_triples**(*x*)

Add inverse triples into dask dataframe :param x: :return:

dicee.read_preprocess_save_load_kg.util.**dataset_sanity_checking**(*train_set: numpy.ndarray*, *num_entities: int*, *num_relations: int*) → None

Parameters

- **train_set**
- **num_entities**
- **num_relations**

Returns

Classes

PreprocessKG	Preprocess the data in memory
LoadSaveToDisk	
ReadFromDisk	Read the data from disk into memory

Package Contents

class dicee.read_preprocess_save_load_kg.**PreprocessKG**(*kg*)

Preprocess the data in memory

kg

start() → None

Preprocess train, valid and test datasets stored in knowledge graph instance

Parameter

rtype
None

preprocess_with_byte_pair_encoding()

preprocess_with_byte_pair_encoding_with_padding() → None

preprocess_with_pandas() → None

Preprocess train, valid and test datasets stored in knowledge graph instance with pandas

(1) Add recipriocal or noisy triples

(2) Construct vocabulary

(3) Index datasets

Parameter

rtype
None

preprocess_with_polars() → None

sequential_vocabulary_construction() → None

(1) Read input data into memory

(2) Remove triples with a condition

(3) **Serialize vocabularies in a pandas dataframe where**
=> the index is integer and => a single column is string (e.g. URI)

class dicee.read_preprocess_save_load_kg.**LoadSaveToDisk**(*kg*)

kg

save()

load()

class dicee.read_preprocess_save_load_kg.**ReadFromDisk**(*kg*)

Read the data from disk into memory

kg

start() → None

Read a knowledge graph from disk into memory

Data will be available at the train_set, test_set, valid_set attributes.

Parameter

None

rtype
None

add_noisy_triples_into_training()

dicee.sanity_checkers

Functions

is_sparql_endpoint_alive([sparql_endpoint])	
validate_knowledge_graph(args)	Validating the source of knowledge graph
sanity_checking_with_arguments(args)	
sanity_check_callback_args(args)	Perform sanity checks on callback-related arguments.

Module Contents

`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.sanity_checkers.`**`sanity_check_callback_args`**(*args*)

Perform sanity checks on callback-related arguments.

dicee.scripts

Submodules

dicee.scripts.index_serve

$ docker pull qdrant/qdrant && docker run -p 6333:6333 -p 6334:6334 -v $(pwd)/qdrant_storage:/qdrant/storage:z qdrant/qdrant $ dicee_vector_db –index –serve –path CountryEmbeddings –collection "countries_vdb"

Attributes

app	
neural_searcher	

Classes

NeuralSearcher	
StringListRequest	!!! abstract "Usage Documentation"

Functions

get_default_arguments()	
index(args)	
root()	
search_embeddings(q)	
retrieve_embeddings(q)	
search_embeddings_batch(request)	
serve(args)	
main()	

Module Contents

dicee.scripts.index_serve.**get_default_arguments**()

dicee.scripts.index_serve.**index**(*args*)

dicee.scripts.index_serve.**app**

dicee.scripts.index_serve.**neural_searcher = None**

class dicee.scripts.index_serve.**NeuralSearcher**(*args*)

collection_name

entity_to_idx = None

qdrant_client

topk = 5

retrieve_embedding(*entity: str = None*, *entities: List[str] = None*) → List

search(*entity: str*)

async dicee.scripts.index_serve.**root**()

async dicee.scripts.index_serve.**search_embeddings**(*q: str*)

async dicee.scripts.index_serve.**retrieve_embeddings**(*q: str*)

class dicee.scripts.index_serve.**StringListRequest**(/, ***data: Any*)

Bases: `pydantic.BaseModel`

!!! abstract "Usage Documentation"
[Models](../concepts/models.md)

A base class for creating Pydantic models.

__class_vars__

The names of the class variables defined on the model.

__private_attributes__

Metadata about the private attributes of the model.

__signature__

The synthesized *__init__* [*Signature*][inspect.Signature] of the model.

__pydantic_complete__

Whether model building is completed, or if there are still undefined fields.

__pydantic_core_schema__

The core schema of the model.

__pydantic_custom_init__

Whether the model has a custom *__init__* function.

__pydantic_decorators__

Metadata containing the decorators defined on the model. This replaces *Model.__validators__* and *Model.__root_validators__* from Pydantic V1.

__pydantic_generic_metadata__

Metadata for generic models; contains data used for a similar purpose to __args__, __origin__, __parameters__ in typing-module generics. May eventually be replaced by these.

__pydantic_parent_namespace__

Parent namespace of the model, used for automatic rebuilding of models.

__pydantic_post_init__

The name of the post-init method for the model, if defined.

__pydantic_root_model__

Whether the model is a [*RootModel*][pydantic.root_model.RootModel].

__pydantic_serializer__

The *pydantic-core SchemaSerializer* used to dump instances of the model.

__pydantic_validator__

The *pydantic-core SchemaValidator* used to validate instances of the model.

__pydantic_fields__

A dictionary of field names and their corresponding [*FieldInfo*][pydantic.fields.FieldInfo] objects.

__pydantic_computed_fields__

A dictionary of computed field names and their corresponding [*ComputedFieldInfo*][pydantic.fields.ComputedFieldInfo] objects.

__pydantic_extra__

A dictionary containing extra values, if [*extra*][pydantic.config.ConfigDict.extra] is set to *'allow'*.

__pydantic_fields_set__

The names of fields explicitly set during instantiation.

__pydantic_private__

Values of private attributes set on the model instance.

queries: List[str]

reducer: str | None = None

async dicee.scripts.index_serve.**search_embeddings_batch**(*request: StringListRequest*)

dicee.scripts.index_serve.**serve**(*args*)

dicee.scripts.index_serve.**main**()

dicee.scripts.run

Functions

get_default_arguments([description])	Extends pytorch_lightning Trainer's arguments with ours
main()	

Module Contents

dicee.scripts.run.**get_default_arguments**(*description=None*)

Extends pytorch_lightning Trainer's arguments with ours

dicee.scripts.run.**main**()

dicee.static_funcs

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])	
load_term_mapping([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

continues on next page

Table 2 – continued from previous page

save_checkpoint_model(→ None)	Store Pytorch model into disk
store(→ None)	
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_kge, ...)	
deploy_head_entity_prediction(pre_trained_kge, ...)	
deploy_relation_prediction(pre_trained_kge, ...)	
vocab_to_parquet(vocab_to_idx, name, ...)	
create_experiment_folder([folder_name])	
continual_training_setup_executor(→ None)	
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)	
download_pretrained_model(→ str)	
write_csv_from_model_parallel(path)	Create
from_pretrained_model_write_embeddings_int(None)	

Module Contents

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.**load_term_mapping**(*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**(*trained_model*, *model_name: str = 'model'*, *full_storage_path: str = None*, *save_embeddings_as_csv=False*) → None

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

`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.`**`write_csv_from_model_parallel`**(*path: str*)

Create

`dicee.static_funcs.`**`from_pretrained_model_write_embeddings_into_csv`**(*path: str*) → None

dicee.static_funcs_training

Functions

`make_iterable_verbose`(→ Iterable)	
`evaluate_lp`([model, triple_idx, num_entities, ...])	
`evaluate_bpe_lp`(model, triple_idx, ...[, info])	
`efficient_zero_grad`(model)	

Module Contents

`dicee.static_funcs_training.`**`make_iterable_verbose`**(*iterable_object*, *verbose*, *desc='Default'*, *position=None*, *leave=True*) → Iterable dicee.static_funcs_training.**evaluate_lp**(*model=None*, *triple_idx=None*, *num_entities=None*, *er_vocab: Dict[Tuple, List] = None*, *re_vocab: Dict[Tuple, List] = None*, *info='Eval Starts'*, *batch_size=128*, *chunk_size=1000*)

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

Attributes

enable_log

Functions

timeit(func)	
preprocesses_input_args(args)	Sanity Checking in input arguments
create_constraints(→ Tuple[dict, dict, dict, dict])	
get_er_vocab(data)	
get_re_vocab(data)	
get_ee_vocab(data)	
mapping_from_first_two_cols_to_third(train_se	

Module Contents

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*)

dicee.trainer

Submodules

dicee.trainer.dice_trainer

Classes

DICE_Trainer	DICE_Trainer implement

Functions

load_term_mapping([file_path])
initialize_trainer(...)
get_callbacks(args)

Module Contents

dicee.trainer.dice_trainer.**load_term_mapping**(*file_path=str*)

dicee.trainer.dice_trainer.**initialize_trainer**(*args*, *callbacks*)
→ *dicee.trainer.torch_trainer.TorchTrainer* | *dicee.trainer.model_parallelism.TensorParallel* | *dicee.trainer.torch_trainer_ddp.*

dicee.trainer.dice_trainer.**get_callbacks**(*args*)

class dicee.trainer.dice_trainer.**DICE_Trainer**(*args*, *is_continual_training: bool*, *storage_path*, *evaluator=None*)

DICE_Trainer implement

1- Pytorch Lightning trainer (https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html) 2- Multi-GPU Trainer(https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html) 3- CPU Trainer

args

is_continual_training:bool

storage_path:str

evaluator:

report:dict

report

args

trainer = None

is_continual_training

storage_path

evaluator = None

form_of_labelling = None

continual_start(*knowledge_graph*)

(1) Initialize training.

(2) Load model

(3) Load trainer (3) Fit model

Parameter

returns

- *model*
- **form_of_labelling** (*str*)

initialize_trainer(*callbacks: List*)
→ lightning.Trainer | *dicee.trainer.model_parallelism.TensorParallel* | *dicee.trainer.torch_trainer.TorchTrainer* | *dicee.t*

Initialize Trainer from input arguments

initialize_or_load_model()

init_dataloader(*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

init_dataset() → torch.utils.data.Dataset

start(*knowledge_graph: dicee.knowledge_graph.KG | numpy.memmap*)
→ Tuple[*dicee.models.base_model.BaseKGE*, str]

Start the training

(1) Initialize Trainer

(2) Initialize or load a pretrained KGE model

in DDP setup, we need to load the memory map of already read/index KG.

k_fold_cross_validation(*dataset*) → Tuple[*dicee.models.base_model.BaseKGE*, str]

Perform K-fold Cross-Validation

1. Obtain K train and test splits.
2. **For each split,**
   2.1 initialize trainer and model 2.2. Train model with configuration provided in args. 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.
3. Report the mean and average MRR .

Parameters

- **self**
- **dataset**

Returns
model

dicee.trainer.model_parallelism

Classes

TensorParallel	Abstract class for Trainer class for knowledge graph embedding models

Functions

extract_input_outputs(z[, device])	
find_good_batch_size(train_loader, tp_ensemble_model)	
forward_backward_update_loss(→ float)	

Module Contents

dicee.trainer.model_parallelism.**extract_input_outputs**(*z: list*, *device=None*)

dicee.trainer.model_parallelism.**find_good_batch_size**(*train_loader*, *tp_ensemble_model*)

dicee.trainer.model_parallelism.**forward_backward_update_loss**(*z: Tuple*, *ensemble_model*) → float

class dicee.trainer.model_parallelism.**TensorParallel**(*args*, *callbacks*)

Bases: *dicee.abstracts.AbstractTrainer*

Abstract class for Trainer class for knowledge graph embedding models

Parameter

args
[str] ?

callbacks: list
?

fit(**args*, ***kwargs*)
Train model

dicee.trainer.torch_trainer

Classes

TorchTrainer	TorchTrainer for using single GPU or multi CPUs on a single node

Module Contents

class dicee.trainer.torch_trainer.**TorchTrainer**(*args*, *callbacks*)

Bases: *dicee.abstracts.AbstractTrainer*

TorchTrainer for using single GPU or multi CPUs on a single node

Arguments

callbacks: list of Abstract callback instances

loss_function = None

optimizer = None

model = None

train_dataloaders = None

training_step = None

process

fit(**args*, *train_dataloaders*, ***kwargs*) → None

Training starts

Arguments

kwargs:Tuple
empty dictionary

Return type
batch loss (float)

forward_backward_update(*x_batch: torch.Tensor*, *y_batch: torch.Tensor*) → torch.Tensor

Compute forward, loss, backward, and parameter update

Arguments

Return type
batch loss (float)

extract_input_outputs_set_device(*batch: list*) → Tuple

Construct inputs and outputs from a batch of inputs with outputs From a batch of inputs and put

Arguments

Return type
(tuple) mini-batch on select device

dicee.trainer.torch_trainer_ddp

Classes

TorchDDPTrainer	A Trainer based on torch.nn.parallel.DistributedDataParallel
NodeTrainer	

Functions

make_iterable_verbose(→ Iterable)	

Module Contents

dicee.trainer.torch_trainer_ddp.**make_iterable_verbose**(*iterable_object*, *verbose*, *desc='Default'*, *position=None*, *leave=True*) → Iterable

class dicee.trainer.torch_trainer_ddp.**TorchDDPTrainer**(*args*, *callbacks*)

Bases: *dicee.abstracts.AbstractTrainer*

A Trainer based on torch.nn.parallel.DistributedDataParallel

Arguments

entity_idxs
mapping.

relation_idxs
mapping.

form
?

store
?

label_smoothing_rate
Using hard targets (0,1) drives weights to infinity. An outlier produces enormous gradients.

Return type
torch.utils.data.Dataset

fit(**args*, ***kwargs*)

Train model

class dicee.trainer.torch_trainer_ddp.**NodeTrainer**(*trainer*, *model: torch.nn.Module*, *train_dataset_loader: torch.utils.data.DataLoader*, *callbacks*, *num_epochs: int*)

trainer

local_rank

global_rank

optimizer

train_dataset_loader

loss_func

callbacks

model

num_epochs

loss_history = []

ctx

scaler

extract_input_outputs(*z: list*)

train()

Training loop for DDP

Classes

DICE_Trainer	DICE_Trainer implement

Package Contents

class dicee.trainer.**DICE_Trainer**(*args*, *is_continual_training: bool*, *storage_path*, *evaluator=None*)

DICE_Trainer implement

1- Pytorch Lightning trainer (https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html) 2- Multi-GPU Trainer(https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html) 3- CPU Trainer

args

is_continual_training:bool

storage_path:str

evaluator:

report:dict

report

args

trainer = None

is_continual_training

storage_path

evaluator = None

form_of_labelling = None

continual_start(*knowledge_graph*)

(1) Initialize training.

(2) Load model

(3) Load trainer (3) Fit model

Parameter

returns

- *model*
- **form_of_labelling** (*str*)

initialize_trainer(*callbacks: List*)
→ lightning.Trainer | *dicee.trainer.model_parallelism.TensorParallel* | *dicee.trainer.torch_trainer.TorchTrainer* | *dicee.t*

Initialize Trainer from input arguments

initialize_or_load_model()

init_dataloader(*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

init_dataset() → torch.utils.data.Dataset

start(*knowledge_graph: dicee.knowledge_graph.KG* | *numpy.memmap*)
→ Tuple[*dicee.models.base_model.BaseKGE*, str]

Start the training

(1) Initialize Trainer

(2) Initialize or load a pretrained KGE model

in DDP setup, we need to load the memory map of already read/index KG.

k_fold_cross_validation(*dataset*) → Tuple[*dicee.models.base_model.BaseKGE*, str]

Perform K-fold Cross-Validation

1. Obtain K train and test splits.
2. **For each split,**
   2.1 initialize trainer and model 2.2. Train model with configuration provided in args. 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.
3. Report the mean and average MRR .

Parameters

- **self**
- **dataset**

Returns
model

14.2 Attributes

```
__version__
```

14.3 Classes

Pyke	A Physical Embedding Model for Knowledge Graphs
DistMult	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
CKeci	Without learning dimension scaling
Keci	Base class for all neural network modules.
TransE	Translating Embeddings for Modeling
DeCaL	Base class for all neural network modules.
DualE	Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)
ComplEx	Base class for all neural network modules.
AConEx	Additive Convolutional ComplEx Knowledge Graph Embeddings
AConvO	Additive Convolutional Octonion Knowledge Graph Embeddings
AConvQ	Additive Convolutional Quaternion Knowledge Graph Embeddings
ConvQ	Convolutional Quaternion Knowledge Graph Embeddings
ConvO	Base class for all neural network modules.
ConEx	Convolutional ComplEx Knowledge Graph Embeddings
QMult	Base class for all neural network modules.
OMult	Base class for all neural network modules.
Shallom	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
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.
EnsembleKGE	
DICE_Trainer	DICE_Trainer implement
KGE	Knowledge Graph Embedding Class for interactive usage of pre-trained models
Execute	A class for Training, Retraining and Evaluation a model.
BPE_NegativeSamplingDataset	An abstract class representing a `Dataset`.
MultiLabelDataset	An abstract class representing a `Dataset`.
MultiClassClassificationDataset	Dataset for the 1vsALL training strategy
OnevsAllDataset	Dataset for the 1vsALL training strategy
KvsAll	Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.
AllvsAll	Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.
OnevsSample	A custom PyTorch Dataset class for knowledge graph embeddings, which includes
KvsSampleDataset	KvsSample a Dataset:
NegSampleDataset	An abstract class representing a `Dataset`.
TriplePredictionDataset	Triple Dataset

continues on next page

Table 3 – continued from previous page

CVDataModule	Create a Dataset for cross validation
LiteralDataset	Dataset for loading and processing literal data for training Literal Embedding model.
QueryGenerator	

14.4 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])	
load_term_mapping([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)	
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_kge, ...)	

continues on next page

Table 4 – continued from previous page

deploy_head_entity_prediction(pre_trained_kge, ...)	
deploy_relation_prediction(pre_trained_kge, ...)	
vocab_to_parquet(vocab_to_idx, name, ...)	
create_experiment_folder([folder_name])	
continual_training_setup_executor(→ None)	
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)	
download_pretrained_model(→ str)	
write_csv_from_model_parallel(path)	Create
from_pretrained_model_write_embeddings_int(None)	
mapping_from_first_two_cols_to_third(train_se	
timeit(func)	
load_term_mapping([file_path])	
reload_dataset(path, form_of_labelling, ...)	Reload the files from disk to construct the Pytorch dataset
construct_dataset(→ torch.utils.data.Dataset)	

14.5 Package Contents

class dicee.**Pyke**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

A Physical Embedding Model for Knowledge Graphs

name = 'Pyke'

dist_func

margin = 1.0

forward_triples(*x: torch.LongTensor*)

Parameters

x

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

Bases: *dicee.models.base_model.BaseKGE*

Embedding Entities and Relations for Learning and Inference in Knowledge Bases https://arxiv.org/abs/1412.6575

name = 'DistMult'

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

Parameters

- **emb_h**
- **emb_r**
- **emb_E**

forward_k_vs_all(*x: torch.LongTensor*)

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

score(*h*, *r*, *t*)

class dicee.**CKeci**(*args*)

Bases: *Keci*

Without learning dimension scaling

name = 'CKeci'

requires_grad_for_interactions = False

class dicee.**Keci**(*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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'Keci'

p

q

r

requires_grad_for_interactions = True

compute_sigma_pp(*hp*, *rp*)

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

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_{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.

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)

`apply_coefficients`(*hp*, *hq*, *rp*, *rq*)

Multiplying a base vector with its scalar coefficient

`clifford_multiplication`(*h0*, *hp*, *hq*, *r0*, *rp*, *rq*)

Compute our CL multiplication

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

`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 with (n,d) shape

returns

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

`forward_k_vs_with_explicit`(*x: torch.Tensor*)

`k_vs_all_score`(*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*)

`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

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

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

Parameter

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

returns

- **a0** (*torch.FloatTensor with (n,k, m) shape*)
- **ap** (*torch.FloatTensor with (n,k, m, p) shape*)
- **aq** (*torch.FloatTensor with (n,k, m, q) shape*)

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

Parameter

x: torch.LongTensor with (n,2) shape

target_entity_idx: torch.LongTensor with (n, k) shape k denotes the selected number of examples.

rtype
torch.FloatTensor with (n, k) shape

score(*h*, *r*, *t*)

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

Parameter

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

rtype
torch.FloatTensor with (n) shape

class dicee.**TransE**(*args*)

Bases: `dicee.models.base_model.BaseKGE`

Translating Embeddings for Modeling Multi-relational Data https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf

name = 'TransE'

margin = 4

score(*head_ent_emb*, *rel_ent_emb*, *tail_ent_emb*)

forward_k_vs_all(*x: torch.Tensor*) → 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 them to be nested in a tree structure. You can assign the submodules as regular attributes:

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


class Model(nn.Module):
    def __init__(self) -> None:
        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 also have their parameters converted 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.

name = 'DeCaL'

entity_embeddings

relation_embeddings

p

q

r

re

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

Parameter

x: torch.LongTensor with (n,) shape

rtype

torch.FloatTensor with (n) shape

cl_pqr(*a: torch.tensor*) → torch.tensor

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,

$$s0 = h_0 r_0 t_0 s1 = \sum_{i=1}^{p} h_i r_i t_0 s2 = \sum_{j=p+1}^{p+q} h_j r_j t_0 s3 = \sum_{i=1}^{q} (h_0 r_i t_i + h_i r_0 t_i) s4 = \sum_{i=p+1}^{p+q} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+1}^{p+q+r} (h_0 r_i t$$

and return:

$$sigma_0 t = \sigma_0 \cdot t_0 = s0 + s1 - s2 s3, s4 and s5$$

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

$$\sigma_p p = \sum_{i=1}^{p-1} \sum_{i'=i+1}^{p} (h_i r_{i'} - h_{i'} r_i)(models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'}$$

For different base vector interactions, we have

$$\sigma_p q = \sum_{i=1}^{p} \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i)(interactionsn between e_i and e_j for 1 <= i <= p and p+1 <= j <= p+q) \sigma_p r = \sum$$

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

Kvsall training

(1) Retrieve real-valued embedding vectors for heads and relations

(2) Construct head entity and relation embeddings according to Cl_{p,q, r}(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,) 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*)

Compute .. math:

```
\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*)

Compute

$$\sum_{i=1}^{p}\sum_{j=p+1}^{p+q}(h_ir_j - h_jr_i)e_ie_j$$

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*)

Compute

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

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.**DualE**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)

name = 'DualE'

entity_embeddings

relation_embeddings

num_ent = None

kvsall_score(*e_1_h, e_2_h, e_3_h, e_4_h, e_5_h, e_6_h, e_7_h, e_8_h, e_1_t, e_2_t, e_3_t, e_4_t, e_5_t, e_6_t, e_7_t, e_8_t, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8*) → torch.tensor

KvsAll scoring function

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

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

Negative Sampling forward pass:

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_k_vs_all(*x*)

KvsAll forward pass

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

T(*x: torch.tensor*) → torch.tensor

Transpose function

Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)

class dicee.**ComplEx**(*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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'ComplEx'

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

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

Parameters

- **emb_h**
- **emb_r**
- **emb_E**

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

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

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional ComplEx Knowledge Graph Embeddings

name = 'AConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

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

Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

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

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

Parameters

x

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

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Octonion Knowledge Graph Embeddings

name = 'AConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

static octonion_normalizer(*emb_rel_e0*, *emb_rel_e1*, *emb_rel_e2*, *emb_rel_e3*, *emb_rel_e4*, *emb_rel_e5*, *emb_rel_e6*, *emb_rel_e7*)

residual_convolution(*O_1*, *O_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Quaternion Knowledge Graph Embeddings

name = 'AConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution(*Q_1*, *Q_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

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

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings

name = 'ConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution(*Q_1, Q_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

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

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'ConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

static octonion_normalizer(*emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4, emb_rel_e5, emb_rel_e6, emb_rel_e7*)

residual_convolution(*O_1, O_2*)

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

Parameters

x

forward_k_vs_all(*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute 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|)

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

Bases: *dicee.models.base_model.BaseKGE*

Convolutional ComplEx Knowledge Graph Embeddings

name = 'ConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

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

Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

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

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

Parameters

x

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

class dicee.**QMult**(*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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'QMult'

explicit = True

quaternion_multiplication_followed_by_inner_product(*h*, *r*, *t*)

Parameters

- **h** – shape: (**batch_dims*, dim) The head representations.
- **r** – shape: (**batch_dims*, dim) The head representations.
- **t** – shape: (**batch_dims*, dim) The tail representations.

Returns

Triple scores.

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

Normalize the length of relation vectors, if the forward constraint has not been applied yet.

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

Parameters

x – The vector.

Returns

The normalized vector.

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

k_vs_all_score(*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*)

Parameters

- **bpe_head_ent_emb**
- **bpe_rel_ent_emb**
- **E**

forward_k_vs_all(*x*)

Parameters

x

forward_k_vs_sample(*x*, *target_entity_idx*)

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples,i.e., [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|)

class dicee.**OMult**(*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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'OMult'

static octonion_normalizer(*emb_rel_e0*, *emb_rel_e1*, *emb_rel_e2*, *emb_rel_e3*, *emb_rel_e4*, *emb_rel_e5*, *emb_rel_e6*, *emb_rel_e7*)

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

k_vs_all_score(*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*)

forward_k_vs_all(*x*)

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples,i.e., [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|)

class dicee.**Shallom**(*args*)

Bases: *dicee.models.base_model.BaseKGE*

A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)

name = 'Shallom'

shallom

get_embeddings() → Tuple[numpy.ndarray, None]

forward_k_vs_all(*x*) → torch.FloatTensor

forward_triples(*x*) → torch.FloatTensor

Parameters

x

Returns

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.

name = 'LFMult'

entity_embeddings

relation_embeddings

degree

m

x_values

forward_triples(*idx_triple*)

Parameters

x

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

Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Pykeen_HolE: Pykeen_HolE: Pykeen_TransD: Pykeen_TransE: Pykeen_TransF: Pykeen_TransH: Pykeen_TransR:

model_kwargs

name

model

loss_history = []

args

entity_embeddings = None

relation_embeddings = None

forward_k_vs_all(*x: torch.LongTensor*)

=> 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

=> 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*)

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

name = 'BytE'

config

temperature = 0.5

topk = 2

transformer

lm_head

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 them to be nested 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) -> None:
        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 also have their parameters converted 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.

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

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]*)
byte pair encoded neural link predictors

Parameters
----

init_params_with_sanity_checking()

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters
x

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

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

get_triple_representation(*idx_hrt*)

get_head_relation_representation(*indexed_triple*)

get_sentence_representation(*x: torch.LongTensor*)

Parameters

- **(b** (*x shape*)
- **3**
- **t)**

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

Parameters
x (*B x 2 x T*)

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

class dicee.**EnsembleKGE**(*models: list = None*, *seed_model=None*, *pretrained_models: List = None*)

name

train_mode = True

args

named_children()

property example_input_array

parameters()

modules()

__iter__()

__len__()

eval()

to(*device*)

state_dict()

Return the state dict of the ensemble.

load_state_dict(*state_dict*, *strict=True*)

Load the state dict into the ensemble.

mem_of_model()

__call__(*x_batch*)

step()

get_embeddings()

__str__()

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.**load_term_mapping**(*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**(*trained_model, model_name: str = 'model', full_storage_path: str = None, save_embeddings_as_csv=False*) → None

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

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

dicee.**write_csv_from_model_parallel**(*path: str*)

Create

dicee.**from_pretrained_model_write_embeddings_into_csv**(*path: str*) → None

class dicee.**DICE_Trainer**(*args*, *is_continual_training: bool*, *storage_path*, *evaluator=None*)

DICE_Trainer implement

1- Pytorch Lightning trainer (https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html) 2- Multi-GPU Trainer(https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html) 3- CPU Trainer

args

is_continual_training:bool

storage_path:str

evaluator:

report:dict

report

args

trainer = None

is_continual_training

storage_path

evaluator = None

form_of_labelling = None

continual_start(*knowledge_graph*)

(1) Initialize training.

(2) Load model

(3) Load trainer (3) Fit model

Parameter

returns

- *model*
- **form_of_labelling** (*str*)

initialize_trainer(*callbacks: List*)
→ lightning.Trainer | *dicee.trainer.model_parallelism.TensorParallel* | *dicee.trainer.torch_trainer.TorchTrainer* | *dicee.t*

Initialize Trainer from input arguments

initialize_or_load_model()

init_dataloader(*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

init_dataset() → torch.utils.data.Dataset

start(*knowledge_graph: dicee.knowledge_graph.KG | numpy.memmap*)
→ Tuple[*dicee.models.base_model.BaseKGE*, str]

Start the training

(1) Initialize Trainer

(2) Initialize or load a pretrained KGE model

in DDP setup, we need to load the memory map of already read/index KG.

k_fold_cross_validation(*dataset*) → Tuple[*dicee.models.base_model.BaseKGE*, str]

Perform K-fold Cross-Validation

1. Obtain K train and test splits.
2. **For each split,**
   2.1 initialize trainer and model 2.2. Train model with configuration provided in args. 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.
3. Report the mean and average MRR .

Parameters

- **self**
- **dataset**

Returns
model

class dicee.**KGE**(*path=None*, *url=None*, *construct_ensemble=False*, *model_name=None*)

Bases: `dicee.abstracts.BaseInteractiveKGE`, `dicee.abstracts.InteractiveQueryDecomposition`, `dicee.abstracts.BaseInteractiveTrainKGE`

Knowledge Graph Embedding Class for interactive usage of pre-trained models

__str__()

to(*device: str*) → None

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=''*)

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*, *batch_size=2*, *topk=1*, *return_indices=False*) → 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*, *batch_size=2*, *topk=1*, *return_indices=False*) → 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*, *batch_size=2*, *topk=1*, *return_indices=False*) → 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: str | List[str] = None, r: str | List[str] = None, t: str | List[str] = None, topk: int = 10, within: List[str] = None, batch_size: int = 1024*)

Predict missing item in a given triple.

Returns

- If you query a single (h, r, ?) or (?, r, t) or (h, ?, t), returns List[(item, score)]
- If you query a batch of B, returns List of B such lists.

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

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*)

predict_literals (*entity: List[str] | str = None*, *attribute: List[str] | str = None*, *denormalize_preds: bool = True*) → numpy.ndarray

Predicts literal values for given entities and attributes.

Parameters

- **entity** (*Union[List[str], str]*) – Entity or list of entities to predict literals for.

- **attribute** (*Union[List[str], str]*) – Attribute or list of attributes to predict literals for.
- **denormalize_preds** (*bool*) – If True, denormalizes the predictions.

Returns
Predictions for the given entities and attributes.

Return type
numpy ndarray

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

distributed

args

is_continual_training = False

trainer = None

trained_model = None

knowledge_graph = None

report

evaluator = None

start_time = None

is_rank_zero() → bool

cleanup()

setup_executor() → None

create_and_store_kg()

load_from_memmap()

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_term_mapping**(*file_path=str*)

dicee.**reload_dataset**(*path: str*, *form_of_labelling*, *scoring_technique*, *neg_ratio*, *label_smoothing_rate*)

Reload the files from disk to construct the Pytorch 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

class dicee.**BPE_NegativeSamplingDataset**(*train_set: torch.LongTensor*, *ordered_shaped_bpe_entities: torch.LongTensor*, *neg_ratio: int*)

Bases: `torch.utils.data.Dataset`

An abstract class representing a `Dataset`.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

`DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

train_set

ordered_bpe_entities

num_bpe_entities

neg_ratio

num_datapoints

__len__()

__getitem__(*idx*)

collate_fn(*batch_shaped_bpe_triples: List[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`

An abstract class representing a `Dataset`.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite *__getitem__()*, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite *__len__()*, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

`DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

train_set

train_indices_target

target_dim

num_datapoints

torch_ordered_shaped_bpe_entities

collate_fn = None

__len__()

__getitem__(*idx*)

class dicee.**MultiClassClassificationDataset**(*subword_units: numpy.ndarray*, *block_size: int = 8*)

Bases: `torch.utils.data.Dataset`

Dataset for the 1vsALL training strategy

Parameters

- **train_set_idx** – Indexed triples for the training.
- **entity_idxs** – mapping.
- **relation_idxs** – mapping.
- **form** – ?
- **num_workers** – int for https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

Return type

torch.utils.data.Dataset

train_data

block_size = 8

num_of_data_points

collate_fn = None

__len__()

__getitem__(*idx*)

class dicee.**OnevsAllDataset**(*train_set_idx: numpy.ndarray*, *entity_idxs*)

Bases: `torch.utils.data.Dataset`

Dataset for the 1vsALL training strategy

Parameters

- **train_set_idx** – Indexed triples for the training.
- **entity_idxs** – mapping.
- **relation_idxs** – mapping.
- **form** – ?
- **num_workers** – int for https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

Return type

torch.utils.data.Dataset

train_data

target_dim

collate_fn = None

__len__()

__getitem__(*idx*)

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 KvsAll training by inheriting from torch.utils.data.Dataset.

Let D denote a dataset for KvsAll training and be defined as D:= {(x,y)_i}_i ^N, where x: (h,r) is an unique tuple of an entity h in E and a relation r in R that has been seed in the input graph. 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

> **Note**
>
> TODO

train_set_idx
[numpy.ndarray] n by 3 array representing n triples

entity_idxs
[dictonary] string representation of an entity to its integer id

relation_idxs
[dictonary] string representation of a relation to its integer id

self : torch.utils.data.Dataset

```
>>> a = KvsAll()
>>> a
? array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

train_data = None

train_target = None

label_smoothing_rate

collate_fn = None

__len__()

__getitem__(*idx*)

class dicee.**AllvsAll**(*train_set_idx: numpy.ndarray*, *entity_idxs*, *relation_idxs*, *label_smoothing_rate=0.0*)

Bases: `torch.utils.data.Dataset`

Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.

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

> **Note**

AllvsAll extends KvsAll via none existing (h,r). Hence, it adds data points that are labelled without 1s,
only with 0s.

train_set_idx
[numpy.ndarray] n by 3 array representing n triples

entity_idxs
[dictonary] string representation of an entity to its integer id

relation_idxs
[dictonary] string representation of a relation to its integer id

self : torch.utils.data.Dataset

```
>>> a = AllvsAll()
>>> a
? array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

train_data = None

train_target = None

label_smoothing_rate

collate_fn = None

target_dim

__len__()

__getitem__(*idx*)

class dicee.**OnevsSample**(*train_set: numpy.ndarray*, *num_entities*, *num_relations*, *neg_sample_ratio: int = None*, *label_smoothing_rate: float = 0.0*)

Bases: torch.utils.data.Dataset

A custom PyTorch Dataset class for knowledge graph embeddings, which includes both positive and negative sampling for a given dataset for multi-class classification problem..

Parameters

- **train_set** (*np.ndarray*) – A numpy array containing triples of knowledge graph data. Each triple consists of (head_entity, relation, tail_entity).
- **num_entities** (*int*) – The number of unique entities in the knowledge graph.
- **num_relations** (*int*) – The number of unique relations in the knowledge graph.
- **neg_sample_ratio** (*int, optional*) – The number of negative samples to be generated per positive sample. Must be a positive integer and less than num_entities.
- **label_smoothing_rate** (*float, optional*) – A label smoothing rate to apply to the positive and negative labels. Defaults to 0.0.

train_data

The input data converted into a PyTorch tensor.

Type
torch.Tensor

num_entities

Number of entities in the dataset.

Type
int

num_relations

Number of relations in the dataset.

Type
int

neg_sample_ratio

Ratio of negative samples to be drawn for each positive sample.

Type
int

label_smoothing_rate

The smoothing factor applied to the labels.

Type
torch.Tensor

collate_fn

A function that can be used to collate data samples into batches (set to None by default).

Type
function, optional

train_data

num_entities

num_relations

neg_sample_ratio = None

label_smoothing_rate

collate_fn = None

__len__()

Returns the number of samples in the dataset.

__getitem__(*idx*)

Retrieves a single data sample from the dataset at the given index.

Parameters
idx (*int*) – The index of the sample to retrieve.

Returns

A tuple consisting of:

- x (torch.Tensor): The head and relation part of the triple.
- y_idx (torch.Tensor): The concatenated indices of the true object (tail entity) and the indices of the negative samples.
- y_vec (torch.Tensor): A vector containing the labels for the positive and negative samples, with label smoothing applied.

Return type
tuple

class dicee.**KvsSampleDataset**(*train_set_idx: numpy.ndarray*, *entity_idxs*, *relation_idxs*, *form*, *store=None*, *neg_ratio=None*, *label_smoothing_rate: float = 0.0*)

Bases: `torch.utils.data.Dataset`

KvsSample a Dataset:

D:= {(x,y)_i}_i ^N, where
. x:(h,r) is a unique h in E and a relation r in R and . y in [0,1]^{|E|} is a binary label.

orall y_i =1 s.t. (h r E_i) in KG

At each mini-batch construction, we subsample(y), hence n
|new_y| << |E| new_y contains all 1's if sum(y)< neg_sample ratio new_y contains

train_set_idx
Indexed triples for the training.

entity_idxs
mapping.

relation_idxs
mapping.

form
?

store
?

label_smoothing_rate
?

torch.utils.data.Dataset

train_data = None

train_target = None

neg_ratio = None

num_entities

label_smoothing_rate

collate_fn = None

max_num_of_classes

__len__()

__getitem__(*idx*)

class dicee.**NegSampleDataset**(*train_set: numpy.ndarray*, *num_entities: int*, *num_relations: int*, *neg_sample_ratio: int = 1*)

Bases: `torch.utils.data.Dataset`

An abstract class representing a `Dataset`.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

`DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

neg_sample_ratio

train_triples

length

num_entities

num_relations

labels

train_set = []

__len__()

__getitem__(*idx*)

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`

Triple Dataset

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

train_set_idx
Indexed triples for the training.

entity_idxs
mapping.

relation_idxs
mapping.

form
?

store
?

label_smoothing_rate

collate_fn: batch:List[torch.IntTensor] Returns ——- torch.utils.data.Dataset

label_smoothing_rate

neg_sample_ratio

train_set

length

num_entities

num_relations

__len__()

__getitem__(*idx*)

collate_fn(*batch: List[torch.Tensor]*)

class dicee.**CVDataModule**(*train_set_idx: numpy.ndarray*, *num_entities*, *num_relations*, *neg_sample_ratio*, *batch_size*, *num_workers*)

Bases: `pytorch_lightning.LightningDataModule`

Create a Dataset for cross validation

Parameters

- **train_set_idx** – Indexed triples for the training.
- **num_entities** – entity to index mapping.
- **num_relations** – relation to index mapping.
- **batch_size** – int
- **form** – ?
- **num_workers** – int for https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

Return type

?

train_set_idx

num_entities

num_relations

neg_sample_ratio

batch_size

num_workers

train_dataloader() → torch.utils.data.DataLoader

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:`~pytorch_lightning.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.

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
```

> **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.**LiteralDataset**(*file_path: str*, *ent_idx: dict = None*, *normalization_type: str = 'z-norm'*, *sampling_ratio: float = None*, *loader_backend: str = 'pandas'*)

Bases: `torch.utils.data.Dataset`

Dataset for loading and processing literal data for training Literal Embedding model. This dataset handles the loading, normalization, and preparation of triples for training a literal embedding model.

Extends torch.utils.data.Dataset for supporting PyTorch dataloaders.

train_file_path

Path to the training data file.

Type
str

normalization

Type of normalization to apply ('z-norm', 'min-max', or None).

Type
str

normalization_params

Parameters used for normalization.

Type
dict

sampling_ratio

Fraction of the training set to use for ablations.

Type
float

entity_to_idx

Mapping of entities to their indices.

Type
dict

num_entities

Total number of entities.

Type
int

data_property_to_idx

Mapping of data properties to their indices.

Type
dict

num_data_properties

Total number of data properties.

Type
int

loader_backend

Backend to use for loading data ('pandas' or 'rdflib').

Type
str

train_file_path

loader_backend = 'pandas'

normalization_type = 'z-norm'

normalization_params

sampling_ratio = None

entity_to_idx = None

num_entities

__getitem__(*index*)

__len__()

static load_and_validate_literal_data(*file_path: str = None*, *loader_backend: str = 'pandas'*) → pandas.DataFrame

Loads and validates the literal data file. :param file_path: Path to the literal data file. :type file_path: str

Returns
DataFrame containing the loaded and validated data.

Return type
pd.DataFrame

static denormalize(*preds_norm*, *attributes*, *normalization_params*) → numpy.ndarray

Denormalizes the predictions based on the normalization type.

Args: preds_norm (np.ndarray): Normalized predictions to be denormalized. attributes (list): List of attributes corresponding to the predictions. normalization_params (dict): Dictionary containing normalization parameters for each attribute.

Returns
Denormalized predictions.

Return type
np.ndarray

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*)

train_path

val_path

test_path

gen_valid = False

gen_test = True

seed = 1

max_ans_num = 1000000.0

mode

ent2id = None

rel2id: Dict = None

ent_in: Dict

ent_out: Dict

query_name_to_struct

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__ = '0.1.5'**

Python Module Index

d

dicee, 12
dicee.__main__, 12
dicee.abstracts, 12
dicee.analyse_experiments, 19
dicee.callbacks, 20
dicee.config, 29
dicee.dataset_classes, 32
dicee.eval_static_funcs, 46
dicee.evaluator, 48
dicee.executer, 49
dicee.knowledge_graph, 51
dicee.knowledge_graph_embeddings, 53
dicee.models, 57
dicee.models.adopt, 57
dicee.models.base_model, 58
dicee.models.clifford, 67
dicee.models.complex, 74
dicee.models.dualE, 77
dicee.models.ensemble, 78
dicee.models.function_space, 79
dicee.models.literal, 82
dicee.models.octonion, 84
dicee.models.pykeen_models, 87
dicee.models.quaternion, 88
dicee.models.real, 91
dicee.models.static_funcs, 92
dicee.models.transformers, 93
dicee.query_generator, 147
dicee.read_preprocess_save_load_kg, 148
dicee.read_preprocess_save_load_kg.preprocess, 148
dicee.read_preprocess_save_load_kg.read_from_disk, 149
dicee.read_preprocess_save_load_kg.save_load_disk, 150
dicee.read_preprocess_save_load_kg.util, 150
dicee.sanity_checkers, 155
dicee.scripts, 156
dicee.scripts.index_serve, 156
dicee.scripts.run, 159
dicee.static_funcs, 159
dicee.static_funcs_training, 162
dicee.static_preprocess_funcs, 163
dicee.trainer, 164
dicee.trainer.dice_trainer, 164
dicee.trainer.model_parallelism, 166
dicee.trainer.torch_trainer, 166
dicee.trainer.torch_trainer_ddp, 168

Index

Non-alphabetical

__call__() (*dicee.EnsembleKGE method*), 197
__call__() (*dicee.models.base_model.IdentityClass method*), 67
__call__() (*dicee.models.ensemble.EnsembleKGE method*), 78
__call__() (*dicee.models.IdentityClass method*), 110, 121, 127
__class_vars__ (*dicee.scripts.index_serve.StringListRequest attribute*), 157
__getitem__() (*dicee.AllvsAll method*), 209
__getitem__() (*dicee.BPE_NegativeSamplingDataset method*), 206
__getitem__() (*dicee.dataset_classes.AllvsAll method*), 37
__getitem__() (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 34
__getitem__() (*dicee.dataset_classes.KvsAll method*), 36
__getitem__() (*dicee.dataset_classes.KvsSampleDataset method*), 39
__getitem__() (*dicee.dataset_classes.LiteralDataset method*), 46
__getitem__() (*dicee.dataset_classes.MultiClassClassificationDataset method*), 35
__getitem__() (*dicee.dataset_classes.MultiLabelDataset method*), 34
__getitem__() (*dicee.dataset_classes.NegSampleDataset method*), 40
__getitem__() (*dicee.dataset_classes.OnevsAllDataset method*), 35
__getitem__() (*dicee.dataset_classes.OnevsSample method*), 38
__getitem__() (*dicee.dataset_classes.TriplePredictionDataset method*), 41
__getitem__() (*dicee.KvsAll method*), 208
__getitem__() (*dicee.KvsSampleDataset method*), 211
__getitem__() (*dicee.LiteralDataset method*), 218
__getitem__() (*dicee.MultiClassClassificationDataset method*), 207
__getitem__() (*dicee.MultiLabelDataset method*), 206
__getitem__() (*dicee.NegSampleDataset method*), 212
__getitem__() (*dicee.OnevsAllDataset method*), 207
__getitem__() (*dicee.OnevsSample method*), 210
__getitem__() (*dicee.TriplePredictionDataset method*), 213
__iter__() (*dicee.config.Namespace method*), 32
__iter__() (*dicee.EnsembleKGE method*), 197
__iter__() (*dicee.knowledge_graph.KG method*), 53
__iter__() (*dicee.models.ensemble.EnsembleKGE method*), 78
__len__() (*dicee.AllvsAll method*), 209
__len__() (*dicee.BPE_NegativeSamplingDataset method*), 206
__len__() (*dicee.dataset_classes.AllvsAll method*), 37
__len__() (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 34
__len__() (*dicee.dataset_classes.KvsAll method*), 36
__len__() (*dicee.dataset_classes.KvsSampleDataset method*), 39
__len__() (*dicee.dataset_classes.LiteralDataset method*), 46
__len__() (*dicee.dataset_classes.MultiClassClassificationDataset method*), 35
__len__() (*dicee.dataset_classes.MultiLabelDataset method*), 34
__len__() (*dicee.dataset_classes.NegSampleDataset method*), 40
__len__() (*dicee.dataset_classes.OnevsAllDataset method*), 35
__len__() (*dicee.dataset_classes.OnevsSample method*), 38
__len__() (*dicee.dataset_classes.TriplePredictionDataset method*), 41
__len__() (*dicee.EnsembleKGE method*), 197
__len__() (*dicee.knowledge_graph.KG method*), 53
__len__() (*dicee.KvsAll method*), 208
__len__() (*dicee.KvsSampleDataset method*), 211
__len__() (*dicee.LiteralDataset method*), 218
__len__() (*dicee.models.ensemble.EnsembleKGE method*), 78
__len__() (*dicee.MultiClassClassificationDataset method*), 207
__len__() (*dicee.MultiLabelDataset method*), 206
__len__() (*dicee.NegSampleDataset method*), 212
__len__() (*dicee.OnevsAllDataset method*), 207
__len__() (*dicee.OnevsSample method*), 210
__len__() (*dicee.TriplePredictionDataset method*), 213
__private_attributes__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_complete__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_computed_fields__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_core_schema__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_custom_init__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_decorators__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_extra__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_fields__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158

__pydantic_fields_set__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_generic_metadata__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_parent_namespace__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_post_init__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_private__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_root_model__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_serializer__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__pydantic_validator__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__setstate__() (*dicee.models.ADOPT method*), 101
__setstate__() (*dicee.models.adopt.ADOPT method*), 57
__signature__ (*dicee.scripts.index_serve.StringListRequest attribute*), 158
__str__() (*dicee.EnsembleKGE method*), 197
__str__() (*dicee.KGE method*), 200
__str__() (*dicee.knowledge_graph_embeddings.KGE method*), 53
__str__() (*dicee.models.ensemble.EnsembleKGE method*), 78
__version__ (*in module dicee*), 219

A

AbstractCallback (*class in dicee.abstracts*), 16
AbstractPPECallback (*class in dicee.abstracts*), 17
AbstractTrainer (*class in dicee.abstracts*), 12
AccumulateEpochLossCallback (*class in dicee.callbacks*), 21
achieve_answer() (*dicee.query_generator.QueryGenerator method*), 148
achieve_answer() (*dicee.QueryGenerator method*), 219
AConEx (*class in dicee*), 183
AConEx (*class in dicee.models*), 117
AConEx (*class in dicee.models.complex*), 75
AConvO (*class in dicee*), 183
AConvO (*class in dicee.models*), 129
AConvO (*class in dicee.models.octonion*), 86
AConvQ (*class in dicee*), 184
AConvQ (*class in dicee.models*), 123
AConvQ (*class in dicee.models.quaternion*), 90
adaptive_lr (*dicee.config.Namespace attribute*), 32
adaptive_swa (*dicee.config.Namespace attribute*), 32
add_new_entity_embeddings() (*dicee.abstracts.BaseInteractiveKGE method*), 15
add_noise_rate (*dicee.config.Namespace attribute*), 30
add_noise_rate (*dicee.knowledge_graph.KG attribute*), 52
add_noisy_triples() (*in module dicee*), 198
add_noisy_triples() (*in module dicee.static_funcs*), 161
add_noisy_triples_into_training() (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method*), 150
add_noisy_triples_into_training() (*dicee.read_preprocess_save_load_kg.ReadFromDisk method*), 155
add_reciprocal (*dicee.knowledge_graph.KG attribute*), 52
ADOPT (*class in dicee.models*), 100
ADOPT (*class in dicee.models.adopt*), 57
adopt() (*in module dicee.models.adopt*), 58
AllvsAll (*class in dicee*), 208
AllvsAll (*class in dicee.dataset_classes*), 36
alphas (*dicee.abstracts.AbstractPPECallback attribute*), 18
alphas (*dicee.callbacks.ASWA attribute*), 24
analyse() (*in module dicee.analyse_experiments*), 20
answer_multi_hop_query() (*dicee.KGE method*), 202
answer_multi_hop_query() (*dicee.knowledge_graph_embeddings.KGE method*), 55
app (*in module dicee.scripts.index_serve*), 157
apply_coefficients() (*dicee.DeCaL method*), 179
apply_coefficients() (*dicee.Keci method*), 176
apply_coefficients() (*dicee.models.clifford.DeCaL method*), 72
apply_coefficients() (*dicee.models.clifford.Keci method*), 69
apply_coefficients() (*dicee.models.DeCaL method*), 135
apply_coefficients() (*dicee.models.Keci method*), 131
apply_reciprical_or_noise() (*in module dicee.read_preprocess_save_load_kg.util*), 153
apply_semantic_constraint (*dicee.abstracts.BaseInteractiveKGE attribute*), 14
apply_unit_norm (*dicee.BaseKGE attribute*), 195
apply_unit_norm (*dicee.models.base_model.BaseKGE attribute*), 65
apply_unit_norm (*dicee.models.BaseKGE attribute*), 107, 111, 114, 119, 125, 138, 141
args (*dicee.BaseKGE attribute*), 195
args (*dicee.DICE_Trainer attribute*), 199

args (*dicee.EnsembleKGE attribute*), 197
args (*dicee.evaluator.Evaluator attribute*), 48
args (*dicee.Execute attribute*), 204
args (*dicee.executer.Execute attribute*), 50
args (*dicee.models.base_model.BaseKGE attribute*), 64
args (*dicee.models.base_model.IdentityClass attribute*), 67
args (*dicee.models.BaseKGE attribute*), 107, 110, 114, 119, 125, 138, 141
args (*dicee.models.ensemble.EnsembleKGE attribute*), 78
args (*dicee.models.IdentityClass attribute*), 110, 121, 127
args (*dicee.models.pykeen_models.PykeenKGE attribute*), 87
args (*dicee.models.PykeenKGE attribute*), 140
args (*dicee.PykeenKGE attribute*), 191
args (*dicee.trainer.DICE_Trainer attribute*), 169
args (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 164
ASWA (*class in dicee.callbacks*), 24
aswa (*dicee.analyse_experiments.Experiment attribute*), 19
attn (*dicee.models.transformers.Block attribute*), 98
attn_dropout (*dicee.models.transformers.CausalSelfAttention attribute*), 96
attributes (*dicee.abstracts.AbstractTrainer attribute*), 13
auto_batch_finding (*dicee.config.Namespace attribute*), 32

B

backend (*dicee.config.Namespace attribute*), 30
backend (*dicee.knowledge_graph.KG attribute*), 52
BaseInteractiveKGE (*class in dicee.abstracts*), 14
BaseInteractiveTrainKGE (*class in dicee.abstracts*), 18
BaseKGE (*class in dicee*), 194
BaseKGE (*class in dicee.models*), 106, 110, 113, 118, 124, 137, 140
BaseKGE (*class in dicee.models.base_model*), 64
BaseKGELightning (*class in dicee.models*), 101
BaseKGELightning (*class in dicee.models.base_model*), 58
batch_kronecker_product() (*dicee.callbacks.KronE static method*), 26
batch_size (*dicee.analyse_experiments.Experiment attribute*), 19
batch_size (*dicee.callbacks.PseudoLabellingCallback attribute*), 24
batch_size (*dicee.config.Namespace attribute*), 30
batch_size (*dicee.CVDataModule attribute*), 213
batch_size (*dicee.dataset_classes.CVDataModule attribute*), 41
batches_per_epoch (*dicee.callbacks.LRScheduler attribute*), 28
bias (*dicee.models.transformers.GPTConfig attribute*), 98
bias (*dicee.models.transformers.LayerNorm attribute*), 95
Block (*class in dicee.models.transformers*), 97
block_size (*dicee.BaseKGE attribute*), 196
block_size (*dicee.config.Namespace attribute*), 32
block_size (*dicee.dataset_classes.MultiClassClassificationDataset attribute*), 35
block_size (*dicee.models.base_model.BaseKGE attribute*), 65
block_size (*dicee.models.BaseKGE attribute*), 108, 111, 115, 120, 126, 138, 142
block_size (*dicee.models.transformers.GPTConfig attribute*), 98
block_size (*dicee.MultiClassClassificationDataset attribute*), 207
bn_conv1 (*dicee.AConvQ attribute*), 184
bn_conv1 (*dicee.ConvQ attribute*), 185
bn_conv1 (*dicee.models.AConvQ attribute*), 124
bn_conv1 (*dicee.models.ConvQ attribute*), 123
bn_conv1 (*dicee.models.quaternion.AConvQ attribute*), 90
bn_conv1 (*dicee.models.quaternion.ConvQ attribute*), 90
bn_conv2 (*dicee.AConvQ attribute*), 184
bn_conv2 (*dicee.ConvQ attribute*), 185
bn_conv2 (*dicee.models.AConvQ attribute*), 124
bn_conv2 (*dicee.models.ConvQ attribute*), 123
bn_conv2 (*dicee.models.quaternion.AConvQ attribute*), 90
bn_conv2 (*dicee.models.quaternion.ConvQ attribute*), 90
bn_conv2d (*dicee.AConEx attribute*), 183
bn_conv2d (*dicee.AConvO attribute*), 184
bn_conv2d (*dicee.ConEx attribute*), 187
bn_conv2d (*dicee.ConvO attribute*), 186
bn_conv2d (*dicee.models.AConEx attribute*), 117
bn_conv2d (*dicee.models.AConvO attribute*), 130
bn_conv2d (*dicee.models.complex.AConEx attribute*), 75

bn_conv2d (*dicee.models.complex.ConEx attribute*), 75
bn_conv2d (*dicee.models.ConEx attribute*), 116
bn_conv2d (*dicee.models.ConvO attribute*), 129
bn_conv2d (*dicee.models.octonion.AConvO attribute*), 86
bn_conv2d (*dicee.models.octonion.ConvO attribute*), 86
BPE_NegativeSamplingDataset (*class in dicee*), 205
BPE_NegativeSamplingDataset (*class in dicee.dataset_classes*), 33
build_chain_funcs() (*dicee.models.FMult2 method*), 144
build_chain_funcs() (*dicee.models.function_space.FMult2 method*), 80
build_func() (*dicee.models.FMult2 method*), 144
build_func() (*dicee.models.function_space.FMult2 method*), 80
BytE (*class in dicee*), 192
BytE (*class in dicee.models.transformers*), 93
byte_pair_encoding (*dicee.analyse_experiments.Experiment attribute*), 19
byte_pair_encoding (*dicee.BaseKGE attribute*), 195
byte_pair_encoding (*dicee.config.Namespace attribute*), 31
byte_pair_encoding (*dicee.knowledge_graph.KG attribute*), 52
byte_pair_encoding (*dicee.models.base_model.BaseKGE attribute*), 65
byte_pair_encoding (*dicee.models.BaseKGE attribute*), 108, 111, 115, 120, 126, 138, 142

C

c_attn (*dicee.models.transformers.CausalSelfAttention attribute*), 96
c_fc (*dicee.models.transformers.MLP attribute*), 97
c_proj (*dicee.models.transformers.CausalSelfAttention attribute*), 96
c_proj (*dicee.models.transformers.MLP attribute*), 97
callbacks (*dicee.abstracts.AbstractTrainer attribute*), 13
callbacks (*dicee.analyse_experiments.Experiment attribute*), 19
callbacks (*dicee.config.Namespace attribute*), 30
callbacks (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 169
CausalSelfAttention (*class in dicee.models.transformers*), 95
chain_func() (*dicee.models.FMult method*), 143
chain_func() (*dicee.models.function_space.FMult method*), 79
chain_func() (*dicee.models.function_space.GFMult method*), 80
chain_func() (*dicee.models.GFMult method*), 144
CKeci (*class in dicee*), 174
CKeci (*class in dicee.models*), 133
CKeci (*class in dicee.models.clifford*), 70
cl_pqr() (*dicee.DeCaL method*), 178
cl_pqr() (*dicee.models.clifford.DeCaL method*), 72
cl_pqr() (*dicee.models.DeCaL method*), 134
cleanup() (*dicee.Execute method*), 204
cleanup() (*dicee.executer.Execute method*), 50
clifford_multiplication() (*dicee.Keci method*), 176
clifford_multiplication() (*dicee.models.clifford.Keci method*), 69
clifford_multiplication() (*dicee.models.Keci method*), 132
clip_lambda (*dicee.models.ADOPT attribute*), 101
clip_lambda (*dicee.models.adopt.ADOPT attribute*), 57
collate_fn (*dicee.AllvsAll attribute*), 209
collate_fn (*dicee.dataset_classes.AllvsAll attribute*), 37
collate_fn (*dicee.dataset_classes.KvsAll attribute*), 36
collate_fn (*dicee.dataset_classes.KvsSampleDataset attribute*), 39
collate_fn (*dicee.dataset_classes.MultiClassClassificationDataset attribute*), 35
collate_fn (*dicee.dataset_classes.MultiLabelDataset attribute*), 34
collate_fn (*dicee.dataset_classes.OnevsAllDataset attribute*), 35
collate_fn (*dicee.dataset_classes.OnevsSample attribute*), 38
collate_fn (*dicee.KvsAll attribute*), 208
collate_fn (*dicee.KvsSampleDataset attribute*), 211
collate_fn (*dicee.MultiClassClassificationDataset attribute*), 207
collate_fn (*dicee.MultiLabelDataset attribute*), 206
collate_fn (*dicee.OnevsAllDataset attribute*), 207
collate_fn (*dicee.OnevsSample attribute*), 210
collate_fn() (*dicee.BPE_NegativeSamplingDataset method*), 206
collate_fn() (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 34
collate_fn() (*dicee.dataset_classes.TriplePredictionDataset method*), 41
collate_fn() (*dicee.TriplePredictionDataset method*), 213
collection_name (*dicee.scripts.index_serve.NeuralSearcher attribute*), 157
comp_func() (*dicee.LFMult method*), 191

comp_func() (*dicee.models.function_space.LFMult method*), 82
comp_func() (*dicee.models.LFMult method*), 146
ComplEx (*class in dicee*), 182
ComplEx (*class in dicee.models*), 117
ComplEx (*class in dicee.models.complex*), 75
compute_convergence() (*in module dicee.callbacks*), 24
compute_func() (*dicee.models.FMult method*), 143
compute_func() (*dicee.models.FMult2 method*), 144
compute_func() (*dicee.models.function_space.FMult method*), 79
compute_func() (*dicee.models.function_space.FMult2 method*), 80
compute_func() (*dicee.models.function_space.GFMult method*), 80
compute_func() (*dicee.models.GFMult method*), 144
compute_mrr() (*dicee.callbacks.ASWA static method*), 25
compute_sigma_pp() (*dicee.DeCaL method*), 179
compute_sigma_pp() (*dicee.Keci method*), 175
compute_sigma_pp() (*dicee.models.clifford.DeCaL method*), 73
compute_sigma_pp() (*dicee.models.clifford.Keci method*), 68
compute_sigma_pp() (*dicee.models.DeCaL method*), 135
compute_sigma_pp() (*dicee.models.Keci method*), 131
compute_sigma_pq() (*dicee.DeCaL method*), 180
compute_sigma_pq() (*dicee.Keci method*), 175
compute_sigma_pq() (*dicee.models.clifford.DeCaL method*), 74
compute_sigma_pq() (*dicee.models.clifford.Keci method*), 69
compute_sigma_pq() (*dicee.models.DeCaL method*), 136
compute_sigma_pq() (*dicee.models.Keci method*), 131
compute_sigma_pr() (*dicee.DeCaL method*), 181
compute_sigma_pr() (*dicee.models.clifford.DeCaL method*), 74
compute_sigma_pr() (*dicee.models.DeCaL method*), 136
compute_sigma_qq() (*dicee.DeCaL method*), 180
compute_sigma_qq() (*dicee.Keci method*), 175
compute_sigma_qq() (*dicee.models.clifford.DeCaL method*), 73
compute_sigma_qq() (*dicee.models.clifford.Keci method*), 68
compute_sigma_qq() (*dicee.models.DeCaL method*), 136
compute_sigma_qq() (*dicee.models.Keci method*), 131
compute_sigma_qr() (*dicee.DeCaL method*), 181
compute_sigma_qr() (*dicee.models.clifford.DeCaL method*), 74
compute_sigma_qr() (*dicee.models.DeCaL method*), 137
compute_sigma_rr() (*dicee.DeCaL method*), 180
compute_sigma_rr() (*dicee.models.clifford.DeCaL method*), 73
compute_sigma_rr() (*dicee.models.DeCaL method*), 136
compute_sigmas_multivect() (*dicee.DeCaL method*), 179
compute_sigmas_multivect() (*dicee.models.clifford.DeCaL method*), 72
compute_sigmas_multivect() (*dicee.models.DeCaL method*), 135
compute_sigmas_single() (*dicee.DeCaL method*), 178
compute_sigmas_single() (*dicee.models.clifford.DeCaL method*), 72
compute_sigmas_single() (*dicee.models.DeCaL method*), 134
ConEx (*class in dicee*), 186
ConEx (*class in dicee.models*), 116
ConEx (*class in dicee.models.complex*), 74
config (*dicee.BytE attribute*), 193
config (*dicee.models.transformers.BytE attribute*), 94
config (*dicee.models.transformers.GPT attribute*), 99
configs (*dicee.abstracts.BaseInteractiveKGE attribute*), 14
configure_optimizers() (*dicee.models.base_model.BaseKGELightning method*), 62
configure_optimizers() (*dicee.models.BaseKGELightning method*), 105
configure_optimizers() (*dicee.models.transformers.GPT method*), 99
construct_batch_selected_cl_multivector() (*dicee.Keci method*), 176
construct_batch_selected_cl_multivector() (*dicee.models.clifford.Keci method*), 70
construct_batch_selected_cl_multivector() (*dicee.models.Keci method*), 132
construct_cl_multivector() (*dicee.DeCaL method*), 179
construct_cl_multivector() (*dicee.Keci method*), 176
construct_cl_multivector() (*dicee.models.clifford.DeCaL method*), 72
construct_cl_multivector() (*dicee.models.clifford.Keci method*), 69
construct_cl_multivector() (*dicee.models.DeCaL method*), 135
construct_cl_multivector() (*dicee.models.Keci method*), 132
construct_dataset() (*in module dicee*), 205
construct_dataset() (*in module dicee.dataset_classes*), 33
construct_ensemble (*dicee.abstracts.BaseInteractiveKGE attribute*), 14

construct_graph() (*dicee.query_generator.QueryGenerator method*), 148
construct_graph() (*dicee.QueryGenerator method*), 219
construct_input_and_output() (*dicee.abstracts.BaseInteractiveKGE method*), 16
construct_multi_coeff() (*dicee.LFMult method*), 190
construct_multi_coeff() (*dicee.models.function_space.LFMult method*), 81
construct_multi_coeff() (*dicee.models.LFMult method*), 145
continual_learning (*dicee.config.Namespace attribute*), 32
continual_start() (*dicee.DICE_Trainer method*), 199
continual_start() (*dicee.executer.ContinuousExecute method*), 51
continual_start() (*dicee.trainer.DICE_Trainer method*), 169
continual_start() (*dicee.trainer.dice_trainer.DICE_Trainer method*), 165
continual_training_setup_executor() (*in module dicee*), 198
continual_training_setup_executor() (*in module dicee.static_funcs*), 162
ContinuousExecute (*class in dicee.executer*), 51
conv2d (*dicee.AConEx attribute*), 183
conv2d (*dicee.AConvO attribute*), 184
conv2d (*dicee.AConvQ attribute*), 184
conv2d (*dicee.ConEx attribute*), 186
conv2d (*dicee.ConvO attribute*), 186
conv2d (*dicee.ConvQ attribute*), 185
conv2d (*dicee.models.AConEx attribute*), 117
conv2d (*dicee.models.AConvO attribute*), 129
conv2d (*dicee.models.AConvQ attribute*), 124
conv2d (*dicee.models.complex.AConEx attribute*), 75
conv2d (*dicee.models.complex.ConEx attribute*), 74
conv2d (*dicee.models.ConEx attribute*), 116
conv2d (*dicee.models.ConvO attribute*), 129
conv2d (*dicee.models.ConvQ attribute*), 123
conv2d (*dicee.models.octonion.AConvO attribute*), 86
conv2d (*dicee.models.octonion.ConvO attribute*), 86
conv2d (*dicee.models.quaternion.AConvQ attribute*), 90
conv2d (*dicee.models.quaternion.ConvQ attribute*), 90
ConvO (*class in dicee*), 185
ConvO (*class in dicee.models*), 128
ConvO (*class in dicee.models.octonion*), 85
ConvQ (*class in dicee*), 185
ConvQ (*class in dicee.models*), 123
ConvQ (*class in dicee.models.quaternion*), 90
create_and_store_kg() (*dicee.Execute method*), 204
create_and_store_kg() (*dicee.executer.Execute method*), 50
create_constraints() (*in module dicee.read_preprocess_save_load_kg.util*), 153
create_constraints() (*in module dicee.static_preprocess_funcs*), 163
create_experiment_folder() (*in module dicee*), 198
create_experiment_folder() (*in module dicee.static_funcs*), 162
create_random_data() (*dicee.callbacks.PseudoLabellingCallback method*), 24
create_recipriocal_triples() (*in module dicee*), 197
create_recipriocal_triples() (*in module dicee.read_preprocess_save_load_kg.util*), 154
create_recipriocal_triples() (*in module dicee.static_funcs*), 160
create_vector_database() (*dicee.KGE method*), 200
create_vector_database() (*dicee.knowledge_graph_embeddings.KGE method*), 53
crop_block_size() (*dicee.models.transformers.GPT method*), 99
ctx (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 169
current_epoch (*dicee.callbacks.SWA attribute*), 29
CVDataModule (*class in dicee*), 213
CVDataModule (*class in dicee.dataset_classes*), 41
cycle_length (*dicee.callbacks.LRScheduler attribute*), 28

D

data_module (*dicee.callbacks.PseudoLabellingCallback attribute*), 24
data_property_embeddings (*dicee.models.literal.LiteralEmbeddings attribute*), 83
data_property_to_idx (*dicee.dataset_classes.LiteralDataset attribute*), 45
data_property_to_idx (*dicee.LiteralDataset attribute*), 217
dataset_dir (*dicee.config.Namespace attribute*), 29
dataset_dir (*dicee.knowledge_graph.KG attribute*), 52
dataset_sanity_checking() (*in module dicee.read_preprocess_save_load_kg.util*), 154
DeCaL (*class in dicee*), 177
DeCaL (*class in dicee.models*), 133

DeCaL (*class in dicee.models.clifford*), 70
decide() (*dicee.callbacks.ASWA method*), 25
default_eval_model (*dicee.callbacks.PeriodicEvalCallback attribute*), 28
degree (*dicee.LFMult attribute*), 190
degree (*dicee.models.function_space.LFMult attribute*), 81
degree (*dicee.models.LFMult attribute*), 145
denormalize() (*dicee.dataset_classes.LiteralDataset static method*), 46
denormalize() (*dicee.LiteralDataset static method*), 218
deploy() (*dicee.KGE method*), 203
deploy() (*dicee.knowledge_graph_embeddings.KGE method*), 56
deploy_head_entity_prediction() (*in module dicee*), 198
deploy_head_entity_prediction() (*in module dicee.static_funcs*), 161
deploy_relation_prediction() (*in module dicee*), 198
deploy_relation_prediction() (*in module dicee.static_funcs*), 162
deploy_tail_entity_prediction() (*in module dicee*), 198
deploy_tail_entity_prediction() (*in module dicee.static_funcs*), 161
deploy_triple_prediction() (*in module dicee*), 198
deploy_triple_prediction() (*in module dicee.static_funcs*), 161
describe() (*dicee.knowledge_graph.KG method*), 53
description_of_input (*dicee.knowledge_graph.KG attribute*), 53
device (*dicee.models.literal.LiteralEmbeddings property*), 83
DICE_Trainer (*class in dicee*), 199
DICE_Trainer (*class in dicee.trainer*), 169
DICE_Trainer (*class in dicee.trainer.dice_trainer*), 164
dicee
    module, 12
dicee.__main__
    module, 12
dicee.abstracts
    module, 12
dicee.analyse_experiments
    module, 19
dicee.callbacks
    module, 20
dicee.config
    module, 29
dicee.dataset_classes
    module, 32
dicee.eval_static_funcs
    module, 46
dicee.evaluator
    module, 48
dicee.executer
    module, 49
dicee.knowledge_graph
    module, 51
dicee.knowledge_graph_embeddings
    module, 53
dicee.models
    module, 57
dicee.models.adopt
    module, 57
dicee.models.base_model
    module, 58
dicee.models.clifford
    module, 67
dicee.models.complex
    module, 74
dicee.models.dualE
    module, 77
dicee.models.ensemble
    module, 78
dicee.models.function_space
    module, 79
dicee.models.literal
    module, 82
dicee.models.octonion
    module, 84

dicee.models.pykeen_models
module, 87
dicee.models.quaternion
module, 88
dicee.models.real
module, 91
dicee.models.static_funcs
module, 92
dicee.models.transformers
module, 93
dicee.query_generator
module, 147
dicee.read_preprocess_save_load_kg
module, 148
dicee.read_preprocess_save_load_kg.preprocess
module, 148
dicee.read_preprocess_save_load_kg.read_from_disk
module, 149
dicee.read_preprocess_save_load_kg.save_load_disk
module, 150
dicee.read_preprocess_save_load_kg.util
module, 150
dicee.sanity_checkers
module, 155
dicee.scripts
module, 156
dicee.scripts.index_serve
module, 156
dicee.scripts.run
module, 159
dicee.static_funcs
module, 159
dicee.static_funcs_training
module, 162
dicee.static_preprocess_funcs
module, 163
dicee.trainer
module, 164
dicee.trainer.dice_trainer
module, 164
dicee.trainer.model_parallelism
module, 166
dicee.trainer.torch_trainer
module, 166
dicee.trainer.torch_trainer_ddp
module, 168
discrete_points (*dicee.models.FMult2 attribute*), 144
discrete_points (*dicee.models.function_space.FMult2 attribute*), 80
dist_func (*dicee.models.Pyke attribute*), 113
dist_func (*dicee.models.real.Pyke attribute*), 92
dist_func (*dicee.Pyke attribute*), 173
DistMult (*class in dicee*), 173
DistMult (*class in dicee.models*), 112
DistMult (*class in dicee.models.real*), 91
distributed (*dicee.Execute attribute*), 204
distributed (*dicee.executer.Execute attribute*), 50
download_file() (*in module dicee*), 198
download_file() (*in module dicee.static_funcs*), 162
download_files_from_url() (*in module dicee*), 198
download_files_from_url() (*in module dicee.static_funcs*), 162
download_pretrained_model() (*in module dicee*), 199
download_pretrained_model() (*in module dicee.static_funcs*), 162
dropout (*dicee.models.literal.LiteralEmbeddings attribute*), 82, 83
dropout (*dicee.models.transformers.CausalSelfAttention attribute*), 96
dropout (*dicee.models.transformers.GPTConfig attribute*), 98
dropout (*dicee.models.transformers.MLP attribute*), 97
DualE (*class in dicee*), 181
DualE (*class in dicee.models*), 146

DualE (*class in dicee.models.dualE*), 77
dummy_eval() (*dicee.evaluator.Evaluator method*), 49
dummy_id (*dicee.knowledge_graph.KG attribute*), 52
during_training (*dicee.evaluator.Evaluator attribute*), 48

E

ee_vocab (*dicee.evaluator.Evaluator attribute*), 48
efficient_zero_grad() (*in module dicee.static_funcs_training*), 163
embedding_dim (*dicee.analyse_experiments.Experiment attribute*), 19
embedding_dim (*dicee.BaseKGE attribute*), 195
embedding_dim (*dicee.config.Namespace attribute*), 30
embedding_dim (*dicee.models.base_model.BaseKGE attribute*), 64
embedding_dim (*dicee.models.BaseKGE attribute*), 107, 110, 114, 119, 125, 138, 141
embedding_dim (*dicee.models.literal.LiteralEmbeddings attribute*), 83
embedding_dims (*dicee.models.literal.LiteralEmbeddings attribute*), 82
enable_log (*in module dicee.static_preprocess_funcs*), 163
enc (*dicee.knowledge_graph.KG attribute*), 52
end() (*dicee.Execute method*), 205
end() (*dicee.executer.Execute method*), 50
EnsembleKGE (*class in dicee*), 196
EnsembleKGE (*class in dicee.models.ensemble*), 78
ent2id (*dicee.query_generator.QueryGenerator attribute*), 147
ent2id (*dicee.QueryGenerator attribute*), 218
ent_in (*dicee.query_generator.QueryGenerator attribute*), 147
ent_in (*dicee.QueryGenerator attribute*), 219
ent_out (*dicee.query_generator.QueryGenerator attribute*), 147
ent_out (*dicee.QueryGenerator attribute*), 219
entities_str (*dicee.knowledge_graph.KG property*), 53
entity_embeddings (*dicee.AConvQ attribute*), 184
entity_embeddings (*dicee.ConvQ attribute*), 185
entity_embeddings (*dicee.DeCaL attribute*), 178
entity_embeddings (*dicee.DualE attribute*), 181
entity_embeddings (*dicee.LFMult attribute*), 190
entity_embeddings (*dicee.models.AConvQ attribute*), 124
entity_embeddings (*dicee.models.clifford.DeCaL attribute*), 71
entity_embeddings (*dicee.models.ConvQ attribute*), 123
entity_embeddings (*dicee.models.DeCaL attribute*), 134
entity_embeddings (*dicee.models.DualE attribute*), 146
entity_embeddings (*dicee.models.dualE.DualE attribute*), 77
entity_embeddings (*dicee.models.FMult attribute*), 143
entity_embeddings (*dicee.models.FMult2 attribute*), 144
entity_embeddings (*dicee.models.function_space.FMult attribute*), 79
entity_embeddings (*dicee.models.function_space.FMult2 attribute*), 80
entity_embeddings (*dicee.models.function_space.GFMult attribute*), 79
entity_embeddings (*dicee.models.function_space.LFMult attribute*), 81
entity_embeddings (*dicee.models.function_space.LFMult1 attribute*), 80
entity_embeddings (*dicee.models.GFMult attribute*), 143
entity_embeddings (*dicee.models.LFMult attribute*), 145
entity_embeddings (*dicee.models.LFMult1 attribute*), 144
entity_embeddings (*dicee.models.literal.LiteralEmbeddings attribute*), 82, 83
entity_embeddings (*dicee.models.pykeen_models.PykeenKGE attribute*), 87
entity_embeddings (*dicee.models.PykeenKGE attribute*), 140
entity_embeddings (*dicee.models.quaternion.AConvQ attribute*), 90
entity_embeddings (*dicee.models.quaternion.ConvQ attribute*), 90
entity_embeddings (*dicee.PykeenKGE attribute*), 191
entity_to_idx (*dicee.dataset_classes.LiteralDataset attribute*), 45, 46
entity_to_idx (*dicee.knowledge_graph.KG attribute*), 52
entity_to_idx (*dicee.LiteralDataset attribute*), 217, 218
entity_to_idx (*dicee.scripts.index_serve.NeuralSearcher attribute*), 157
epoch_count (*dicee.abstracts.AbstractPPECallback attribute*), 17
epoch_count (*dicee.callbacks.ASWA attribute*), 24
epoch_counter (*dicee.callbacks.Eval attribute*), 25
epoch_counter (*dicee.callbacks.KGESaveCallback attribute*), 23
epoch_counter (*dicee.callbacks.PeriodicEvalCallback attribute*), 27
epoch_ratio (*dicee.callbacks.Eval attribute*), 25
er_vocab (*dicee.evaluator.Evaluator attribute*), 48
estimate_mfu() (*dicee.models.transformers.GPT method*), 99

estimate_q() (*in module dicee.callbacks*), 24
Eval (*class in dicee.callbacks*), 25
eval() (*dicee.EnsembleKGE method*), 197
eval() (*dicee.evaluator.Evaluator method*), 49
eval() (*dicee.models.ensemble.EnsembleKGE method*), 78
eval_at_epochs (*dicee.config.Namespace attribute*), 32
eval_epochs (*dicee.callbacks.PeriodicEvalCallback attribute*), 28
eval_every_n_epochs (*dicee.config.Namespace attribute*), 32
eval_lp_performance() (*dicee.KGE method*), 200
eval_lp_performance() (*dicee.knowledge_graph_embeddings.KGE method*), 53
eval_model (*dicee.config.Namespace attribute*), 31
eval_model (*dicee.knowledge_graph.KG attribute*), 52
eval_rank_of_head_and_tail_byte_pair_encoded_entity() (*dicee.evaluator.Evaluator method*), 49
eval_rank_of_head_and_tail_entity() (*dicee.evaluator.Evaluator method*), 49
eval_with_bpe_vs_all() (*dicee.evaluator.Evaluator method*), 49
eval_with_byte() (*dicee.evaluator.Evaluator method*), 49
eval_with_data() (*dicee.evaluator.Evaluator method*), 49
eval_with_vs_all() (*dicee.evaluator.Evaluator method*), 49
evaluate() (*in module dicee*), 198
evaluate() (*in module dicee.static_funcs*), 162
evaluate_bpe_lp() (*in module dicee.static_funcs_training*), 163
evaluate_ensemble_link_prediction_performance() (*in module dicee.eval_static_funcs*), 48
evaluate_link_prediction_performance() (*in module dicee.eval_static_funcs*), 47
evaluate_link_prediction_performance_with_bpe() (*in module dicee.eval_static_funcs*), 47
evaluate_link_prediction_performance_with_bpe_reciprocals() (*in module dicee.eval_static_funcs*), 47
evaluate_link_prediction_performance_with_reciprocals() (*in module dicee.eval_static_funcs*), 47
evaluate_literal_prediction() (*in module dicee.eval_static_funcs*), 47
evaluate_lp() (*dicee.evaluator.Evaluator method*), 49
evaluate_lp() (*in module dicee.static_funcs_training*), 162
evaluate_lp_bpe_k_vs_all() (*dicee.evaluator.Evaluator method*), 49
evaluate_lp_bpe_k_vs_all() (*in module dicee.eval_static_funcs*), 47
evaluate_lp_k_vs_all() (*dicee.evaluator.Evaluator method*), 49
evaluate_lp_with_byte() (*dicee.evaluator.Evaluator method*), 49
Evaluator (*class in dicee.evaluator*), 48
evaluator (*dicee.DICE_Trainer attribute*), 199
evaluator (*dicee.Execute attribute*), 204
evaluator (*dicee.executer.Execute attribute*), 50
evaluator (*dicee.trainer.DICE_Trainer attribute*), 169
evaluator (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 165
every_x_epoch (*dicee.callbacks.KGESaveCallback attribute*), 23
example_input_array (*dicee.EnsembleKGE property*), 197
example_input_array (*dicee.models.ensemble.EnsembleKGE property*), 78
Execute (*class in dicee*), 204
Execute (*class in dicee.executer*), 50
exists() (*dicee.knowledge_graph.KG method*), 53
Experiment (*class in dicee.analyse_experiments*), 19
experiment_dir (*dicee.callbacks.LRScheduler attribute*), 28
experiment_dir (*dicee.callbacks.PeriodicEvalCallback attribute*), 27
explicit (*dicee.models.QMult attribute*), 122
explicit (*dicee.models.quaternion.QMult attribute*), 89
explicit (*dicee.QMult attribute*), 188
exponential_function() (*in module dicee*), 198
exponential_function() (*in module dicee.static_funcs*), 162
extract_input_outputs() (*dicee.trainer.torch_trainer_ddp.NodeTrainer method*), 169
extract_input_outputs() (*in module dicee.trainer.model_parallelism*), 166
extract_input_outputs_set_device() (*dicee.trainer.torch_trainer.TorchTrainer method*), 167

F

f (*dicee.callbacks.KronE attribute*), 26
fc (*dicee.models.literal.LiteralEmbeddings attribute*), 83
fc1 (*dicee.AConEx attribute*), 183
fc1 (*dicee.AConvO attribute*), 184
fc1 (*dicee.AConvQ attribute*), 184
fc1 (*dicee.ConEx attribute*), 186
fc1 (*dicee.ConvO attribute*), 186
fc1 (*dicee.ConvQ attribute*), 185
fc1 (*dicee.models.AConEx attribute*), 117

fc1 (*dicee.models.AConvO attribute*), 130
fc1 (*dicee.models.AConvQ attribute*), 124
fc1 (*dicee.models.complex.AConEx attribute*), 75
fc1 (*dicee.models.complex.ConEx attribute*), 75
fc1 (*dicee.models.ConEx attribute*), 116
fc1 (*dicee.models.ConvO attribute*), 129
fc1 (*dicee.models.ConvQ attribute*), 123
fc1 (*dicee.models.octonion.AConvO attribute*), 86
fc1 (*dicee.models.octonion.ConvO attribute*), 86
fc1 (*dicee.models.quaternion.AConvQ attribute*), 90
fc1 (*dicee.models.quaternion.ConvQ attribute*), 90
fc_num_input (*dicee.AConEx attribute*), 183
fc_num_input (*dicee.AConvO attribute*), 184
fc_num_input (*dicee.AConvQ attribute*), 184
fc_num_input (*dicee.ConEx attribute*), 186
fc_num_input (*dicee.ConvO attribute*), 186
fc_num_input (*dicee.ConvQ attribute*), 185
fc_num_input (*dicee.models.AConEx attribute*), 117
fc_num_input (*dicee.models.AConvO attribute*), 130
fc_num_input (*dicee.models.AConvQ attribute*), 124
fc_num_input (*dicee.models.complex.AConEx attribute*), 75
fc_num_input (*dicee.models.complex.ConEx attribute*), 75
fc_num_input (*dicee.models.ConEx attribute*), 116
fc_num_input (*dicee.models.ConvO attribute*), 129
fc_num_input (*dicee.models.ConvQ attribute*), 123
fc_num_input (*dicee.models.octonion.AConvO attribute*), 86
fc_num_input (*dicee.models.octonion.ConvO attribute*), 86
fc_num_input (*dicee.models.quaternion.AConvQ attribute*), 90
fc_num_input (*dicee.models.quaternion.ConvQ attribute*), 90
fc_out (*dicee.models.literal.LiteralEmbeddings attribute*), 83
feature_map_dropout (*dicee.AConEx attribute*), 183
feature_map_dropout (*dicee.AConvO attribute*), 184
feature_map_dropout (*dicee.AConvQ attribute*), 184
feature_map_dropout (*dicee.ConEx attribute*), 187
feature_map_dropout (*dicee.ConvO attribute*), 186
feature_map_dropout (*dicee.ConvQ attribute*), 185
feature_map_dropout (*dicee.models.AConEx attribute*), 117
feature_map_dropout (*dicee.models.AConvO attribute*), 130
feature_map_dropout (*dicee.models.AConvQ attribute*), 124
feature_map_dropout (*dicee.models.complex.AConEx attribute*), 75
feature_map_dropout (*dicee.models.complex.ConEx attribute*), 75
feature_map_dropout (*dicee.models.ConEx attribute*), 116
feature_map_dropout (*dicee.models.ConvO attribute*), 129
feature_map_dropout (*dicee.models.ConvQ attribute*), 123
feature_map_dropout (*dicee.models.octonion.AConvO attribute*), 86
feature_map_dropout (*dicee.models.octonion.ConvO attribute*), 86
feature_map_dropout (*dicee.models.quaternion.AConvQ attribute*), 90
feature_map_dropout (*dicee.models.quaternion.ConvQ attribute*), 90
feature_map_dropout_rate (*dicee.BaseKGE attribute*), 195
feature_map_dropout_rate (*dicee.config.Namespace attribute*), 31
feature_map_dropout_rate (*dicee.models.base_model.BaseKGE attribute*), 65
feature_map_dropout_rate (*dicee.models.BaseKGE attribute*), 107, 111, 114, 119, 125, 138, 141
fill_query() (*dicee.query_generator.QueryGenerator method*), 148
fill_query() (*dicee.QueryGenerator method*), 219
find_good_batch_size() (*in module dicee.trainer.model_parallelism*), 166
find_missing_triples() (*dicee.KGE method*), 203
find_missing_triples() (*dicee.knowledge_graph_embeddings.KGE method*), 56
fit() (*dicee.trainer.model_parallelism.TensorParallel method*), 166
fit() (*dicee.trainer.torch_trainer_ddp.TorchDDPTrainer method*), 168
fit() (*dicee.trainer.torch_trainer.TorchTrainer method*), 167
flash (*dicee.models.transformers.CausalSelfAttention attribute*), 96
FMult (*class in dicee.models*), 143
FMult (*class in dicee.models.function_space*), 79
FMult2 (*class in dicee.models*), 144
FMult2 (*class in dicee.models.function_space*), 80
form_of_labelling (*dicee.DICE_Trainer attribute*), 199
form_of_labelling (*dicee.trainer.DICE_Trainer attribute*), 169
form_of_labelling (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 165

forward() (*dicee.BaseKGE method*), 196
forward() (*dicee.BytE method*), 193
forward() (*dicee.models.base_model.BaseKGE method*), 66
forward() (*dicee.models.base_model.IdentityClass static method*), 67
forward() (*dicee.models.BaseKGE method*), 108, 112, 115, 120, 126, 139, 142
forward() (*dicee.models.IdentityClass static method*), 110, 121, 127
forward() (*dicee.models.literal.LiteralEmbeddings method*), 83
forward() (*dicee.models.transformers.Block method*), 98
forward() (*dicee.models.transformers.BytE method*), 94
forward() (*dicee.models.transformers.CausalSelfAttention method*), 96
forward() (*dicee.models.transformers.GPT method*), 99
forward() (*dicee.models.transformers.LayerNorm method*), 95
forward() (*dicee.models.transformers.MLP method*), 97
forward_backward_update() (*dicee.trainer.torch_trainer.TorchTrainer method*), 167
forward_backward_update_loss() (*in module dicee.trainer.model_parallelism*), 166
forward_byte_pair_encoded_k_vs_all() (*dicee.BaseKGE method*), 196
forward_byte_pair_encoded_k_vs_all() (*dicee.models.base_model.BaseKGE method*), 65
forward_byte_pair_encoded_k_vs_all() (*dicee.models.BaseKGE method*), 108, 111, 115, 120, 126, 138, 142
forward_byte_pair_encoded_triple() (*dicee.BaseKGE method*), 196
forward_byte_pair_encoded_triple() (*dicee.models.base_model.BaseKGE method*), 65
forward_byte_pair_encoded_triple() (*dicee.models.BaseKGE method*), 108, 111, 115, 120, 126, 139, 142
forward_k_vs_all() (*dicee.AConEx method*), 183
forward_k_vs_all() (*dicee.AConvO method*), 184
forward_k_vs_all() (*dicee.AConvQ method*), 184
forward_k_vs_all() (*dicee.BaseKGE method*), 196
forward_k_vs_all() (*dicee.ComplEx method*), 183
forward_k_vs_all() (*dicee.ConEx method*), 187
forward_k_vs_all() (*dicee.ConvO method*), 186
forward_k_vs_all() (*dicee.ConvQ method*), 185
forward_k_vs_all() (*dicee.DeCaL method*), 179
forward_k_vs_all() (*dicee.DistMult method*), 174
forward_k_vs_all() (*dicee.DualE method*), 182
forward_k_vs_all() (*dicee.Keci method*), 176
forward_k_vs_all() (*dicee.models.AConEx method*), 117
forward_k_vs_all() (*dicee.models.AConvO method*), 130
forward_k_vs_all() (*dicee.models.AConvQ method*), 124
forward_k_vs_all() (*dicee.models.base_model.BaseKGE method*), 66
forward_k_vs_all() (*dicee.models.BaseKGE method*), 109, 112, 116, 120, 126, 139, 142
forward_k_vs_all() (*dicee.models.clifford.DeCaL method*), 72
forward_k_vs_all() (*dicee.models.clifford.Keci method*), 69
forward_k_vs_all() (*dicee.models.ComplEx method*), 118
forward_k_vs_all() (*dicee.models.complex.AConEx method*), 75
forward_k_vs_all() (*dicee.models.complex.ComplEx method*), 76
forward_k_vs_all() (*dicee.models.complex.ConEx method*), 75
forward_k_vs_all() (*dicee.models.ConEx method*), 116
forward_k_vs_all() (*dicee.models.ConvO method*), 129
forward_k_vs_all() (*dicee.models.ConvQ method*), 123
forward_k_vs_all() (*dicee.models.DeCaL method*), 135
forward_k_vs_all() (*dicee.models.DistMult method*), 113
forward_k_vs_all() (*dicee.models.DualE method*), 147
forward_k_vs_all() (*dicee.models.dualE.DualE method*), 77
forward_k_vs_all() (*dicee.models.Keci method*), 132
forward_k_vs_all() (*dicee.models.octonion.AConvO method*), 86
forward_k_vs_all() (*dicee.models.octonion.ConvO method*), 86
forward_k_vs_all() (*dicee.models.octonion.OMult method*), 85
forward_k_vs_all() (*dicee.models.OMult method*), 128
forward_k_vs_all() (*dicee.models.pykeen_models.PykeenKGE method*), 87
forward_k_vs_all() (*dicee.models.PykeenKGE method*), 140
forward_k_vs_all() (*dicee.models.QMult method*), 123
forward_k_vs_all() (*dicee.models.quaternion.AConvQ method*), 91
forward_k_vs_all() (*dicee.models.quaternion.ConvQ method*), 90
forward_k_vs_all() (*dicee.models.quaternion.QMult method*), 89
forward_k_vs_all() (*dicee.models.real.DistMult method*), 91
forward_k_vs_all() (*dicee.models.real.Shallom method*), 92
forward_k_vs_all() (*dicee.models.real.TransE method*), 92
forward_k_vs_all() (*dicee.models.Shallom method*), 113
forward_k_vs_all() (*dicee.models.TransE method*), 113
forward_k_vs_all() (*dicee.OMult method*), 189

forward_k_vs_all() (*dicee.PykeenKGE method*), 191
forward_k_vs_all() (*dicee.QMult method*), 188
forward_k_vs_all() (*dicee.Shallom method*), 190
forward_k_vs_all() (*dicee.TransE method*), 177
forward_k_vs_sample() (*dicee.AConEx method*), 183
forward_k_vs_sample() (*dicee.BaseKGE method*), 196
forward_k_vs_sample() (*dicee.ComplEx method*), 183
forward_k_vs_sample() (*dicee.ConEx method*), 187
forward_k_vs_sample() (*dicee.DistMult method*), 174
forward_k_vs_sample() (*dicee.Keci method*), 177
forward_k_vs_sample() (*dicee.models.AConEx method*), 117
forward_k_vs_sample() (*dicee.models.base_model.BaseKGE method*), 66
forward_k_vs_sample() (*dicee.models.BaseKGE method*), 109, 112, 116, 120, 126, 139, 142
forward_k_vs_sample() (*dicee.models.clifford.Keci method*), 70
forward_k_vs_sample() (*dicee.models.ComplEx method*), 118
forward_k_vs_sample() (*dicee.models.complex.AConEx method*), 75
forward_k_vs_sample() (*dicee.models.complex.ComplEx method*), 76
forward_k_vs_sample() (*dicee.models.complex.ConEx method*), 75
forward_k_vs_sample() (*dicee.models.ConEx method*), 116
forward_k_vs_sample() (*dicee.models.DistMult method*), 113
forward_k_vs_sample() (*dicee.models.Keci method*), 133
forward_k_vs_sample() (*dicee.models.pykeen_models.PykeenKGE method*), 88
forward_k_vs_sample() (*dicee.models.PykeenKGE method*), 140
forward_k_vs_sample() (*dicee.models.QMult method*), 123
forward_k_vs_sample() (*dicee.models.quaternion.QMult method*), 89
forward_k_vs_sample() (*dicee.models.real.DistMult method*), 91
forward_k_vs_sample() (*dicee.PykeenKGE method*), 192
forward_k_vs_sample() (*dicee.QMult method*), 188
forward_k_vs_with_explicit() (*dicee.Keci method*), 176
forward_k_vs_with_explicit() (*dicee.models.clifford.Keci method*), 69
forward_k_vs_with_explicit() (*dicee.models.Keci method*), 132
forward_triples() (*dicee.AConEx method*), 183
forward_triples() (*dicee.AConvO method*), 184
forward_triples() (*dicee.AConvQ method*), 184
forward_triples() (*dicee.BaseKGE method*), 196
forward_triples() (*dicee.ConEx method*), 187
forward_triples() (*dicee.ConvO method*), 186
forward_triples() (*dicee.ConvQ method*), 185
forward_triples() (*dicee.DeCaL method*), 178
forward_triples() (*dicee.DualE method*), 181
forward_triples() (*dicee.Keci method*), 177
forward_triples() (*dicee.LFMult method*), 190
forward_triples() (*dicee.models.AConEx method*), 117
forward_triples() (*dicee.models.AConvO method*), 130
forward_triples() (*dicee.models.AConvQ method*), 124
forward_triples() (*dicee.models.base_model.BaseKGE method*), 66
forward_triples() (*dicee.models.BaseKGE method*), 108, 112, 115, 120, 126, 139, 142
forward_triples() (*dicee.models.clifford.DeCaL method*), 71
forward_triples() (*dicee.models.clifford.Keci method*), 70
forward_triples() (*dicee.models.complex.AConEx method*), 75
forward_triples() (*dicee.models.complex.ConEx method*), 75
forward_triples() (*dicee.models.ConEx method*), 116
forward_triples() (*dicee.models.ConvO method*), 129
forward_triples() (*dicee.models.ConvQ method*), 123
forward_triples() (*dicee.models.DeCaL method*), 134
forward_triples() (*dicee.models.DualE method*), 146
forward_triples() (*dicee.models.dualE.DualE method*), 77
forward_triples() (*dicee.models.FMult method*), 143
forward_triples() (*dicee.models.FMult2 method*), 144
forward_triples() (*dicee.models.function_space.FMult method*), 79
forward_triples() (*dicee.models.function_space.FMult2 method*), 80
forward_triples() (*dicee.models.function_space.GFMult method*), 80
forward_triples() (*dicee.models.function_space.LFMult method*), 81
forward_triples() (*dicee.models.function_space.LFMult1 method*), 80
forward_triples() (*dicee.models.GFMult method*), 144
forward_triples() (*dicee.models.Keci method*), 133
forward_triples() (*dicee.models.LFMult method*), 145
forward_triples() (*dicee.models.LFMult1 method*), 144

forward_triples() (*dicee.models.octonion.AConvO method*), 86
forward_triples() (*dicee.models.octonion.ConvO method*), 86
forward_triples() (*dicee.models.Pyke method*), 113
forward_triples() (*dicee.models.pykeen_models.PykeenKGE method*), 87
forward_triples() (*dicee.models.PykeenKGE method*), 140
forward_triples() (*dicee.models.quaternion.AConvQ method*), 91
forward_triples() (*dicee.models.quaternion.ConvQ method*), 90
forward_triples() (*dicee.models.real.Pyke method*), 92
forward_triples() (*dicee.models.real.Shallom method*), 92
forward_triples() (*dicee.models.Shallom method*), 113
forward_triples() (*dicee.Pyke method*), 173
forward_triples() (*dicee.PykeenKGE method*), 192
forward_triples() (*dicee.Shallom method*), 190
freeze_entity_embeddings (*dicee.models.literal.LiteralEmbeddings attribute*), 83
frequency (*dicee.callbacks.Perturb attribute*), 27
from_pretrained() (*dicee.models.transformers.GPT class method*), 99
from_pretrained_model_write_embeddings_into_csv() (*in module dicee*), 199
from_pretrained_model_write_embeddings_into_csv() (*in module dicee.static_funcs*), 162
full_storage_path (*dicee.analyse_experiments.Experiment attribute*), 19
func_triple_to_bpe_representation (*dicee.evaluator.Evaluator attribute*), 48
func_triple_to_bpe_representation() (*dicee.knowledge_graph.KG method*), 53
function() (*dicee.models.FMult2 method*), 144
function() (*dicee.models.function_space.FMult2 method*), 80

G

gamma (*dicee.models.FMult attribute*), 143
gamma (*dicee.models.function_space.FMult attribute*), 79
gate_residual (*dicee.models.literal.LiteralEmbeddings attribute*), 83
gated_residual_proj (*dicee.models.literal.LiteralEmbeddings attribute*), 83
gelu (*dicee.models.transformers.MLP attribute*), 97
gen_test (*dicee.query_generator.QueryGenerator attribute*), 147
gen_test (*dicee.QueryGenerator attribute*), 218
gen_valid (*dicee.query_generator.QueryGenerator attribute*), 147
gen_valid (*dicee.QueryGenerator attribute*), 218
generate() (*dicee.BytE method*), 193
generate() (*dicee.KGE method*), 200
generate() (*dicee.knowledge_graph_embeddings.KGE method*), 53
generate() (*dicee.models.transformers.BytE method*), 94
generate_queries() (*dicee.query_generator.QueryGenerator method*), 148
generate_queries() (*dicee.QueryGenerator method*), 219
get_aswa_state_dict() (*dicee.callbacks.ASWA method*), 25
get_bpe_head_and_relation_representation() (*dicee.BaseKGE method*), 196
get_bpe_head_and_relation_representation() (*dicee.models.base_model.BaseKGE method*), 66
get_bpe_head_and_relation_representation() (*dicee.models.BaseKGE method*), 109, 112, 116, 120, 126, 139, 143
get_bpe_token_representation() (*dicee.abstracts.BaseInteractiveKGE method*), 14
get_callbacks() (*in module dicee.trainer.dice_trainer*), 164
get_default_arguments() (*in module dicee.analyse_experiments*), 19
get_default_arguments() (*in module dicee.scripts.index_serve*), 157
get_default_arguments() (*in module dicee.scripts.run*), 159
get_ee_vocab() (*in module dicee*), 197
get_ee_vocab() (*in module dicee.read_preprocess_save_load_kg.util*), 153
get_ee_vocab() (*in module dicee.static_funcs*), 160
get_ee_vocab() (*in module dicee.static_preprocess_funcs*), 163
get_embeddings() (*dicee.BaseKGE method*), 196
get_embeddings() (*dicee.EnsembleKGE method*), 197
get_embeddings() (*dicee.models.base_model.BaseKGE method*), 66
get_embeddings() (*dicee.models.BaseKGE method*), 109, 112, 116, 120, 126, 139, 143
get_embeddings() (*dicee.models.ensemble.EnsembleKGE method*), 78
get_embeddings() (*dicee.models.real.Shallom method*), 92
get_embeddings() (*dicee.models.Shallom method*), 113
get_embeddings() (*dicee.Shallom method*), 190
get_entity_embeddings() (*dicee.abstracts.BaseInteractiveKGE method*), 15
get_entity_index() (*dicee.abstracts.BaseInteractiveKGE method*), 15
get_er_vocab() (*in module dicee*), 197
get_er_vocab() (*in module dicee.read_preprocess_save_load_kg.util*), 153
get_er_vocab() (*in module dicee.static_funcs*), 160
get_er_vocab() (*in module dicee.static_preprocess_funcs*), 163

get_eval_report() (*dicee.abstracts.BaseInteractiveKGE method*), 14
get_head_relation_representation() (*dicee.BaseKGE method*), 196
get_head_relation_representation() (*dicee.models.base_model.BaseKGE method*), 66
get_head_relation_representation() (*dicee.models.BaseKGE method*), 109, 112, 116, 120, 126, 139, 142
get_kronecker_triple_representation() (*dicee.callbacks.KronE method*), 26
get_num_params() (*dicee.models.transformers.GPT method*), 99
get_padded_bpe_triple_representation() (*dicee.abstracts.BaseInteractiveKGE method*), 14
get_queries() (*dicee.query_generator.QueryGenerator method*), 148
get_queries() (*dicee.QueryGenerator method*), 219
get_re_vocab() (*in module dicee*), 197
get_re_vocab() (*in module dicee.read_preprocess_save_load_kg.util*), 153
get_re_vocab() (*in module dicee.static_funcs*), 160
get_re_vocab() (*in module dicee.static_preprocess_funcs*), 163
get_relation_embeddings() (*dicee.abstracts.BaseInteractiveKGE method*), 15
get_relation_index() (*dicee.abstracts.BaseInteractiveKGE method*), 15
get_sentence_representation() (*dicee.BaseKGE method*), 196
get_sentence_representation() (*dicee.models.base_model.BaseKGE method*), 66
get_sentence_representation() (*dicee.models.BaseKGE method*), 109, 112, 116, 120, 126, 139, 142
get_transductive_entity_embeddings() (*dicee.KGE method*), 200
get_transductive_entity_embeddings() (*dicee.knowledge_graph_embeddings.KGE method*), 53
get_triple_representation() (*dicee.BaseKGE method*), 196
get_triple_representation() (*dicee.models.base_model.BaseKGE method*), 66
get_triple_representation() (*dicee.models.BaseKGE method*), 109, 112, 116, 120, 126, 139, 142
GFMult (*class in dicee.models*), 143
GFMult (*class in dicee.models.function_space*), 79
global_rank (*dicee.abstracts.AbstractTrainer attribute*), 13
global_rank (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 168
GPT (*class in dicee.models.transformers*), 98
GPTConfig (*class in dicee.models.transformers*), 98
gpus (*dicee.config.Namespace attribute*), 30
gradient_accumulation_steps (*dicee.config.Namespace attribute*), 31
ground_queries() (*dicee.query_generator.QueryGenerator method*), 148
ground_queries() (*dicee.QueryGenerator method*), 219

H

hidden_dim (*dicee.models.literal.LiteralEmbeddings attribute*), 83
hidden_dropout (*dicee.BaseKGE attribute*), 195
hidden_dropout (*dicee.models.base_model.BaseKGE attribute*), 65
hidden_dropout (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 142
hidden_dropout_rate (*dicee.BaseKGE attribute*), 195
hidden_dropout_rate (*dicee.config.Namespace attribute*), 31
hidden_dropout_rate (*dicee.models.base_model.BaseKGE attribute*), 65
hidden_dropout_rate (*dicee.models.BaseKGE attribute*), 107, 111, 114, 119, 125, 138, 141
hidden_normalizer (*dicee.BaseKGE attribute*), 195
hidden_normalizer (*dicee.models.base_model.BaseKGE attribute*), 65
hidden_normalizer (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 142

I

IdentityClass (*class in dicee.models*), 109, 121, 127
IdentityClass (*class in dicee.models.base_model*), 66
idx_entity_to_bpe_shaped (*dicee.knowledge_graph.KG attribute*), 52
index() (*in module dicee.scripts.index_serve*), 157
index_triple() (*dicee.abstracts.BaseInteractiveKGE method*), 15
init_dataloader() (*dicee.DICE_Trainer method*), 200
init_dataloader() (*dicee.trainer.DICE_Trainer method*), 170
init_dataloader() (*dicee.trainer.dice_trainer.DICE_Trainer method*), 165
init_dataset() (*dicee.DICE_Trainer method*), 200
init_dataset() (*dicee.trainer.DICE_Trainer method*), 170
init_dataset() (*dicee.trainer.dice_trainer.DICE_Trainer method*), 165
init_param (*dicee.config.Namespace attribute*), 31
init_params_with_sanity_checking() (*dicee.BaseKGE method*), 196
init_params_with_sanity_checking() (*dicee.models.base_model.BaseKGE method*), 65
init_params_with_sanity_checking() (*dicee.models.BaseKGE method*), 108, 112, 115, 120, 126, 139, 142
initial_eval_setting (*dicee.callbacks.ASWA attribute*), 24
initialize_or_load_model() (*dicee.DICE_Trainer method*), 200
initialize_or_load_model() (*dicee.trainer.DICE_Trainer method*), 170
initialize_or_load_model() (*dicee.trainer.dice_trainer.DICE_Trainer method*), 165

initialize_trainer() (dicee.DICE_Trainer method), 199
initialize_trainer() (dicee.trainer.DICE_Trainer method), 170
initialize_trainer() (dicee.trainer.dice_trainer.DICE_Trainer method), 165
initialize_trainer() (in module dicee.trainer.dice_trainer), 164
input_dp_ent_real (dicee.BaseKGE attribute), 195
input_dp_ent_real (dicee.models.base_model.BaseKGE attribute), 65
input_dp_ent_real (dicee.models.BaseKGE attribute), 108, 111, 115, 119, 125, 138, 142
input_dp_rel_real (dicee.BaseKGE attribute), 195
input_dp_rel_real (dicee.models.base_model.BaseKGE attribute), 65
input_dp_rel_real (dicee.models.BaseKGE attribute), 108, 111, 115, 119, 125, 138, 142
input_dropout_rate (dicee.BaseKGE attribute), 195
input_dropout_rate (dicee.config.Namespace attribute), 31
input_dropout_rate (dicee.models.base_model.BaseKGE attribute), 65
input_dropout_rate (dicee.models.BaseKGE attribute), 107, 111, 114, 119, 125, 138, 141
InteractiveQueryDecomposition (class in dicee.abstracts), 16
intialize_model() (in module dicee), 198
intialize_model() (in module dicee.static_funcs), 161
is_continual_training (dicee.DICE_Trainer attribute), 199
is_continual_training (dicee.evaluator.Evaluator attribute), 48
is_continual_training (dicee.Execute attribute), 204
is_continual_training (dicee.executer.Execute attribute), 50
is_continual_training (dicee.trainer.DICE_Trainer attribute), 169
is_continual_training (dicee.trainer.dice_trainer.DICE_Trainer attribute), 164
is_global_zero (dicee.abstracts.AbstractTrainer attribute), 13
is_rank_zero() (dicee.Execute method), 204
is_rank_zero() (dicee.executer.Execute method), 50
is_seen() (dicee.abstracts.BaseInteractiveKGE method), 15
is_sparql_endpoint_alive() (in module dicee.sanity_checkers), 156

K

k (dicee.models.FMult attribute), 143
k (dicee.models.FMult2 attribute), 144
k (dicee.models.function_space.FMult attribute), 79
k (dicee.models.function_space.FMult2 attribute), 80
k (dicee.models.function_space.GFMult attribute), 79
k (dicee.models.GFMult attribute), 143
k_fold_cross_validation() (dicee.DICE_Trainer method), 200
k_fold_cross_validation() (dicee.trainer.DICE_Trainer method), 170
k_fold_cross_validation() (dicee.trainer.dice_trainer.DICE_Trainer method), 165
k_vs_all_score() (dicee.ComplEx static method), 183
k_vs_all_score() (dicee.DistMult method), 174
k_vs_all_score() (dicee.Keci method), 176
k_vs_all_score() (dicee.models.clifford.Keci method), 69
k_vs_all_score() (dicee.models.ComplEx static method), 118
k_vs_all_score() (dicee.models.complex.ComplEx static method), 76
k_vs_all_score() (dicee.models.DistMult method), 112
k_vs_all_score() (dicee.models.Keci method), 132
k_vs_all_score() (dicee.models.octonion.OMult method), 85
k_vs_all_score() (dicee.models.OMult method), 128
k_vs_all_score() (dicee.models.QMult method), 123
k_vs_all_score() (dicee.models.quaternion.QMult method), 89
k_vs_all_score() (dicee.models.real.DistMult method), 91
k_vs_all_score() (dicee.OMult method), 189
k_vs_all_score() (dicee.QMult method), 188
Keci (class in dicee), 174
Keci (class in dicee.models), 130
Keci (class in dicee.models.clifford), 67
kernel_size (dicee.BaseKGE attribute), 195
kernel_size (dicee.config.Namespace attribute), 31
kernel_size (dicee.models.base_model.BaseKGE attribute), 65
kernel_size (dicee.models.BaseKGE attribute), 107, 111, 114, 119, 125, 138, 141
KG (class in dicee.knowledge_graph), 51
kg (dicee.callbacks.PseudoLabellingCallback attribute), 24
kg (dicee.read_preprocess_save_load_kg.LoadSaveToDisk attribute), 155
kg (dicee.read_preprocess_save_load_kg.PreprocessKG attribute), 154
kg (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG attribute), 149
kg (dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk attribute), 149

kg (*dicee.read_preprocess_save_load_kg.ReadFromDisk attribute*), 155
kg (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk attribute*), 150
KGE (*class in dicee*), 200
KGE (*class in dicee.knowledge_graph_embeddings*), 53
KGESaveCallback (*class in dicee.callbacks*), 22
knowledge_graph (*dicee.Execute attribute*), 204
knowledge_graph (*dicee.executer.Execute attribute*), 50
KronE (*class in dicee.callbacks*), 26
KvsAll (*class in dicee*), 207
KvsAll (*class in dicee.dataset_classes*), 35
kvsall_score() (*dicee.DualE method*), 181
kvsall_score() (*dicee.models.DualE method*), 146
kvsall_score() (*dicee.models.dualE.DualE method*), 77
KvsSampleDataset (*class in dicee*), 211
KvsSampleDataset (*class in dicee.dataset_classes*), 39

L

label_smoothing_rate (*dicee.AllvsAll attribute*), 209
label_smoothing_rate (*dicee.config.Namespace attribute*), 31
label_smoothing_rate (*dicee.dataset_classes.AllvsAll attribute*), 37
label_smoothing_rate (*dicee.dataset_classes.KvsAll attribute*), 36
label_smoothing_rate (*dicee.dataset_classes.KvsSampleDataset attribute*), 39
label_smoothing_rate (*dicee.dataset_classes.OnevsSample attribute*), 38
label_smoothing_rate (*dicee.dataset_classes.TriplePredictionDataset attribute*), 41
label_smoothing_rate (*dicee.KvsAll attribute*), 208
label_smoothing_rate (*dicee.KvsSampleDataset attribute*), 211
label_smoothing_rate (*dicee.OnevsSample attribute*), 210
label_smoothing_rate (*dicee.TriplePredictionDataset attribute*), 213
labels (*dicee.dataset_classes.NegSampleDataset attribute*), 40
labels (*dicee.NegSampleDataset attribute*), 212
layer_norm (*dicee.models.literal.LiteralEmbeddings attribute*), 83
LayerNorm (*class in dicee.models.transformers*), 95
learning_rate (*dicee.BaseKGE attribute*), 195
learning_rate (*dicee.models.base_model.BaseKGE attribute*), 65
learning_rate (*dicee.models.BaseKGE attribute*), 107, 111, 114, 119, 125, 138, 141
length (*dicee.dataset_classes.NegSampleDataset attribute*), 40
length (*dicee.dataset_classes.TriplePredictionDataset attribute*), 41
length (*dicee.NegSampleDataset attribute*), 212
length (*dicee.TriplePredictionDataset attribute*), 213
level (*dicee.callbacks.Perturb attribute*), 27
LFMult (*class in dicee*), 190
LFMult (*class in dicee.models*), 145
LFMult (*class in dicee.models.function_space*), 81
LFMult1 (*class in dicee.models*), 144
LFMult1 (*class in dicee.models.function_space*), 80
linear() (*dicee.LFMult method*), 190
linear() (*dicee.models.function_space.LFMult method*), 81
linear() (*dicee.models.LFMult method*), 145
list2tuple() (*dicee.query_generator.QueryGenerator method*), 148
list2tuple() (*dicee.QueryGenerator method*), 219
LiteralDataset (*class in dicee*), 216
LiteralDataset (*class in dicee.dataset_classes*), 44
LiteralEmbeddings (*class in dicee.models.literal*), 82
lm_head (*dicee.BytE attribute*), 193
lm_head (*dicee.models.transformers.BytE attribute*), 94
lm_head (*dicee.models.transformers.GPT attribute*), 99
ln_1 (*dicee.models.transformers.Block attribute*), 98
ln_2 (*dicee.models.transformers.Block attribute*), 98
load() (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk method*), 155
load() (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk method*), 150
load_and_validate_literal_data() (*dicee.dataset_classes.LiteralDataset static method*), 46
load_and_validate_literal_data() (*dicee.LiteralDataset static method*), 218
load_from_memmap() (*dicee.Execute method*), 204
load_from_memmap() (*dicee.executer.Execute method*), 50
load_json() (*in module dicee*), 198
load_json() (*in module dicee.static_funcs*), 161
load_model() (*in module dicee*), 197

load_model() (*in module dicee.static_funcs*), 161
load_model_ensemble() (*in module dicee*), 197
load_model_ensemble() (*in module dicee.static_funcs*), 161
load_numpy() (*in module dicee*), 198
load_numpy() (*in module dicee.static_funcs*), 162
load_numpy_ndarray() (*in module dicee.read_preprocess_save_load_kg.util*), 153
load_pickle() (*in module dicee*), 197
load_pickle() (*in module dicee.read_preprocess_save_load_kg.util*), 154
load_pickle() (*in module dicee.static_funcs*), 161
load_queries() (*dicee.query_generator.QueryGenerator method*), 148
load_queries() (*dicee.QueryGenerator method*), 219
load_queries_and_answers() (*dicee.query_generator.QueryGenerator static method*), 148
load_queries_and_answers() (*dicee.QueryGenerator static method*), 219
load_state_dict() (*dicee.EnsembleKGE method*), 197
load_state_dict() (*dicee.models.ensemble.EnsembleKGE method*), 78
load_term_mapping() (*in module dicee*), 197, 205
load_term_mapping() (*in module dicee.static_funcs*), 161
load_term_mapping() (*in module dicee.trainer.dice_trainer*), 164
load_with_pandas() (*in module dicee.read_preprocess_save_load_kg.util*), 153
loader_backend (*dicee.dataset_classes.LiteralDataset attribute*), 45, 46
loader_backend (*dicee.LiteralDataset attribute*), 217, 218
LoadSaveToDisk (*class in dicee.read_preprocess_save_load_kg*), 155
LoadSaveToDisk (*class in dicee.read_preprocess_save_load_kg.save_load_disk*), 150
local_rank (*dicee.abstracts.AbstractTrainer attribute*), 13
local_rank (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 168
loss (*dicee.BaseKGE attribute*), 195
loss (*dicee.models.base_model.BaseKGE attribute*), 65
loss (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 141
loss_func (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 169
loss_function (*dicee.trainer.torch_trainer.TorchTrainer attribute*), 167
loss_function() (*dicee.BytE method*), 193
loss_function() (*dicee.models.base_model.BaseKGELightning method*), 60
loss_function() (*dicee.models.BaseKGELightning method*), 103
loss_function() (*dicee.models.transformers.BytE method*), 94
loss_history (*dicee.BaseKGE attribute*), 195
loss_history (*dicee.models.base_model.BaseKGE attribute*), 65
loss_history (*dicee.models.BaseKGE attribute*), 108, 111, 115, 120, 126, 138, 142
loss_history (*dicee.models.pykeen_models.PykeenKGE attribute*), 87
loss_history (*dicee.models.PykeenKGE attribute*), 140
loss_history (*dicee.PykeenKGE attribute*), 191
loss_history (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 169
lr (*dicee.analyse_experiments.Experiment attribute*), 19
lr (*dicee.config.Namespace attribute*), 30
lr_init (*dicee.callbacks.SWA attribute*), 29
lr_lambda (*dicee.callbacks.LRScheduler attribute*), 28
LRScheduler (*class in dicee.callbacks*), 28

M

m (*dicee.LFMult attribute*), 190
m (*dicee.models.function_space.LFMult attribute*), 81
m (*dicee.models.LFMult attribute*), 145
main() (*in module dicee.scripts.index_serve*), 159
main() (*in module dicee.scripts.run*), 159
make_iterable_verbose() (*in module dicee.static_funcs_training*), 162
make_iterable_verbose() (*in module dicee.trainer.torch_trainer_ddp*), 168
mapping_from_first_two_cols_to_third() (*in module dicee*), 205
mapping_from_first_two_cols_to_third() (*in module dicee.static_preprocess_funcs*), 164
margin (*dicee.models.Pyke attribute*), 113
margin (*dicee.models.real.Pyke attribute*), 92
margin (*dicee.models.real.TransE attribute*), 92
margin (*dicee.models.TransE attribute*), 113
margin (*dicee.Pyke attribute*), 173
margin (*dicee.TransE attribute*), 177
max_ans_num (*dicee.query_generator.QueryGenerator attribute*), 147
max_ans_num (*dicee.QueryGenerator attribute*), 218
max_epochs (*dicee.callbacks.KGESaveCallback attribute*), 23
max_epochs (*dicee.callbacks.PeriodicEvalCallback attribute*), 27

max_epochs (*dicee.callbacks.SWA attribute*), 29
max_length_subword_tokens (*dicee.BaseKGE attribute*), 195
max_length_subword_tokens (*dicee.knowledge_graph.KG attribute*), 52
max_length_subword_tokens (*dicee.models.base_model.BaseKGE attribute*), 65
max_length_subword_tokens (*dicee.models.BaseKGE attribute*), 108, 111, 115, 120, 126, 138, 142
max_num_of_classes (*dicee.dataset_classes.KvsSampleDataset attribute*), 39
max_num_of_classes (*dicee.KvsSampleDataset attribute*), 211
mem_of_model() (*dicee.EnsembleKGE method*), 197
mem_of_model() (*dicee.models.base_model.BaseKGELightning method*), 59
mem_of_model() (*dicee.models.BaseKGELightning method*), 102
mem_of_model() (*dicee.models.ensemble.EnsembleKGE method*), 78
method (*dicee.callbacks.Perturb attribute*), 27
MLP (*class in dicee.models.transformers*), 96
mlp (*dicee.models.transformers.Block attribute*), 98
mode (*dicee.query_generator.QueryGenerator attribute*), 147
mode (*dicee.QueryGenerator attribute*), 218
model (*dicee.config.Namespace attribute*), 30
model (*dicee.models.pykeen_models.PykeenKGE attribute*), 87
model (*dicee.models.PykeenKGE attribute*), 140
model (*dicee.PykeenKGE attribute*), 191
model (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 169
model (*dicee.trainer.torch_trainer.TorchTrainer attribute*), 167
model_kwargs (*dicee.models.pykeen_models.PykeenKGE attribute*), 87
model_kwargs (*dicee.models.PykeenKGE attribute*), 139
model_kwargs (*dicee.PykeenKGE attribute*), 191
model_name (*dicee.analyse_experiments.Experiment attribute*), 19
module
    dicee, 12
    dicee.__main__, 12
    dicee.abstracts, 12
    dicee.analyse_experiments, 19
    dicee.callbacks, 20
    dicee.config, 29
    dicee.dataset_classes, 32
    dicee.eval_static_funcs, 46
    dicee.evaluator, 48
    dicee.executer, 49
    dicee.knowledge_graph, 51
    dicee.knowledge_graph_embeddings, 53
    dicee.models, 57
    dicee.models.adopt, 57
    dicee.models.base_model, 58
    dicee.models.clifford, 67
    dicee.models.complex, 74
    dicee.models.dualE, 77
    dicee.models.ensemble, 78
    dicee.models.function_space, 79
    dicee.models.literal, 82
    dicee.models.octonion, 84
    dicee.models.pykeen_models, 87
    dicee.models.quaternion, 88
    dicee.models.real, 91
    dicee.models.static_funcs, 92
    dicee.models.transformers, 93
    dicee.query_generator, 147
    dicee.read_preprocess_save_load_kg, 148
    dicee.read_preprocess_save_load_kg.preprocess, 148
    dicee.read_preprocess_save_load_kg.read_from_disk, 149
    dicee.read_preprocess_save_load_kg.save_load_disk, 150
    dicee.read_preprocess_save_load_kg.util, 150
    dicee.sanity_checkers, 155
    dicee.scripts, 156
    dicee.scripts.index_serve, 156
    dicee.scripts.run, 159
    dicee.static_funcs, 159
    dicee.static_funcs_training, 162
    dicee.static_preprocess_funcs, 163
    dicee.trainer, 164

dicee.trainer.dice_trainer, 164
dicee.trainer.model_parallelism, 166
dicee.trainer.torch_trainer, 166
dicee.trainer.torch_trainer_ddp, 168
modules() (*dicee.EnsembleKGE method*), 197
modules() (*dicee.models.ensemble.EnsembleKGE method*), 78
moving_average() (*dicee.callbacks.SWA static method*), 29
MultiClassClassificationDataset (*class in dicee*), 206
MultiClassClassificationDataset (*class in dicee.dataset_classes*), 34
MultiLabelDataset (*class in dicee*), 206
MultiLabelDataset (*class in dicee.dataset_classes*), 34

N

n (*dicee.models.FMult2 attribute*), 144
n (*dicee.models.function_space.FMult2 attribute*), 80
n_embd (*dicee.models.transformers.CausalSelfAttention attribute*), 96
n_embd (*dicee.models.transformers.GPTConfig attribute*), 98
n_epochs_eval_model (*dicee.callbacks.PeriodicEvalCallback attribute*), 27
n_epochs_eval_model (*dicee.config.Namespace attribute*), 32
n_head (*dicee.models.transformers.CausalSelfAttention attribute*), 96
n_head (*dicee.models.transformers.GPTConfig attribute*), 98
n_layer (*dicee.models.transformers.GPTConfig attribute*), 98
n_layers (*dicee.models.FMult2 attribute*), 144
n_layers (*dicee.models.function_space.FMult2 attribute*), 80
name (*dicee.abstracts.BaseInteractiveKGE property*), 15
name (*dicee.AConEx attribute*), 183
name (*dicee.AConvO attribute*), 183
name (*dicee.AConvQ attribute*), 184
name (*dicee.BytE attribute*), 192
name (*dicee.CKeci attribute*), 174
name (*dicee.ComplEx attribute*), 183
name (*dicee.ConEx attribute*), 186
name (*dicee.ConvO attribute*), 186
name (*dicee.ConvQ attribute*), 185
name (*dicee.DeCaL attribute*), 178
name (*dicee.DistMult attribute*), 174
name (*dicee.DualE attribute*), 181
name (*dicee.EnsembleKGE attribute*), 196
name (*dicee.Keci attribute*), 175
name (*dicee.LFMult attribute*), 190
name (*dicee.models.AConEx attribute*), 117
name (*dicee.models.AConvO attribute*), 129
name (*dicee.models.AConvQ attribute*), 124
name (*dicee.models.CKeci attribute*), 133
name (*dicee.models.clifford.CKeci attribute*), 70
name (*dicee.models.clifford.DeCaL attribute*), 71
name (*dicee.models.clifford.Keci attribute*), 68
name (*dicee.models.ComplEx attribute*), 118
name (*dicee.models.complex.AConEx attribute*), 75
name (*dicee.models.complex.ComplEx attribute*), 76
name (*dicee.models.complex.ConEx attribute*), 74
name (*dicee.models.ConEx attribute*), 116
name (*dicee.models.ConvO attribute*), 129
name (*dicee.models.ConvQ attribute*), 123
name (*dicee.models.DeCaL attribute*), 134
name (*dicee.models.DistMult attribute*), 112
name (*dicee.models.DualE attribute*), 146
name (*dicee.models.dualE.DualE attribute*), 77
name (*dicee.models.ensemble.EnsembleKGE attribute*), 78
name (*dicee.models.FMult attribute*), 143
name (*dicee.models.FMult2 attribute*), 144
name (*dicee.models.function_space.FMult attribute*), 79
name (*dicee.models.function_space.FMult2 attribute*), 80
name (*dicee.models.function_space.GFMult attribute*), 79
name (*dicee.models.function_space.LFMult attribute*), 81
name (*dicee.models.function_space.LFMult1 attribute*), 80
name (*dicee.models.GFMult attribute*), 143

name (*dicee.models.Keci attribute*), 131
name (*dicee.models.LFMult attribute*), 145
name (*dicee.models.LFMult1 attribute*), 144
name (*dicee.models.octonion.AConvO attribute*), 86
name (*dicee.models.octonion.ConvO attribute*), 86
name (*dicee.models.octonion.OMult attribute*), 85
name (*dicee.models.OMult attribute*), 128
name (*dicee.models.Pyke attribute*), 113
name (*dicee.models.pykeen_models.PykeenKGE attribute*), 87
name (*dicee.models.PykeenKGE attribute*), 140
name (*dicee.models.QMult attribute*), 122
name (*dicee.models.quaternion.AConvQ attribute*), 90
name (*dicee.models.quaternion.ConvQ attribute*), 90
name (*dicee.models.quaternion.QMult attribute*), 89
name (*dicee.models.real.DistMult attribute*), 91
name (*dicee.models.real.Pyke attribute*), 92
name (*dicee.models.real.Shallom attribute*), 92
name (*dicee.models.real.TransE attribute*), 91
name (*dicee.models.Shallom attribute*), 113
name (*dicee.models.TransE attribute*), 113
name (*dicee.models.transformers.BytE attribute*), 94
name (*dicee.OMult attribute*), 189
name (*dicee.Pyke attribute*), 173
name (*dicee.PykeenKGE attribute*), 191
name (*dicee.QMult attribute*), 188
name (*dicee.Shallom attribute*), 189
name (*dicee.TransE attribute*), 177
named_children() (*dicee.EnsembleKGE method*), 197
named_children() (*dicee.models.ensemble.EnsembleKGE method*), 78
Namespace (*class in dicee.config*), 29
neg_ratio (*dicee.BPE_NegativeSamplingDataset attribute*), 206
neg_ratio (*dicee.config.Namespace attribute*), 30
neg_ratio (*dicee.dataset_classes.BPE_NegativeSamplingDataset attribute*), 34
neg_ratio (*dicee.dataset_classes.KvsSampleDataset attribute*), 39
neg_ratio (*dicee.KvsSampleDataset attribute*), 211
neg_sample_ratio (*dicee.CVDataModule attribute*), 213
neg_sample_ratio (*dicee.dataset_classes.CVDataModule attribute*), 41
neg_sample_ratio (*dicee.dataset_classes.NegSampleDataset attribute*), 40
neg_sample_ratio (*dicee.dataset_classes.OnevsSample attribute*), 38
neg_sample_ratio (*dicee.dataset_classes.TriplePredictionDataset attribute*), 41
neg_sample_ratio (*dicee.NegSampleDataset attribute*), 212
neg_sample_ratio (*dicee.OnevsSample attribute*), 210
neg_sample_ratio (*dicee.TriplePredictionDataset attribute*), 213
negnorm() (*dicee.abstracts.InteractiveQueryDecomposition method*), 16
NegSampleDataset (*class in dicee*), 211
NegSampleDataset (*class in dicee.dataset_classes*), 39
neural_searcher (*in module dicee.scripts.index_serve*), 157
NeuralSearcher (*class in dicee.scripts.index_serve*), 157
NodeTrainer (*class in dicee.trainer.torch_trainer_ddp*), 168
norm_fc1 (*dicee.AConEx attribute*), 183
norm_fc1 (*dicee.AConvO attribute*), 184
norm_fc1 (*dicee.ConEx attribute*), 187
norm_fc1 (*dicee.ConvO attribute*), 186
norm_fc1 (*dicee.models.AConEx attribute*), 117
norm_fc1 (*dicee.models.AConvO attribute*), 130
norm_fc1 (*dicee.models.complex.AConEx attribute*), 75
norm_fc1 (*dicee.models.complex.ConEx attribute*), 75
norm_fc1 (*dicee.models.ConEx attribute*), 116
norm_fc1 (*dicee.models.ConvO attribute*), 129
norm_fc1 (*dicee.models.octonion.AConvO attribute*), 86
norm_fc1 (*dicee.models.octonion.ConvO attribute*), 86
normalization (*dicee.analyse_experiments.Experiment attribute*), 20
normalization (*dicee.config.Namespace attribute*), 30
normalization (*dicee.dataset_classes.LiteralDataset attribute*), 45
normalization (*dicee.LiteralDataset attribute*), 217
normalization_params (*dicee.dataset_classes.LiteralDataset attribute*), 45, 46
normalization_params (*dicee.LiteralDataset attribute*), 217, 218
normalization_type (*dicee.dataset_classes.LiteralDataset attribute*), 46

normalization_type (*dicee.LiteralDataset attribute*), 218
normalize_head_entity_embeddings (*dicee.BaseKGE attribute*), 195
normalize_head_entity_embeddings (*dicee.models.base_model.BaseKGE attribute*), 65
normalize_head_entity_embeddings (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 141
normalize_relation_embeddings (*dicee.BaseKGE attribute*), 195
normalize_relation_embeddings (*dicee.models.base_model.BaseKGE attribute*), 65
normalize_relation_embeddings (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 141
normalize_tail_entity_embeddings (*dicee.BaseKGE attribute*), 195
normalize_tail_entity_embeddings (*dicee.models.base_model.BaseKGE attribute*), 65
normalize_tail_entity_embeddings (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 142
normalizer_class (*dicee.BaseKGE attribute*), 195
normalizer_class (*dicee.models.base_model.BaseKGE attribute*), 65
normalizer_class (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 141
num_bpe_entities (*dicee.BPE_NegativeSamplingDataset attribute*), 206
num_bpe_entities (*dicee.dataset_classes.BPE_NegativeSamplingDataset attribute*), 34
num_bpe_entities (*dicee.knowledge_graph.KG attribute*), 52
num_core (*dicee.config.Namespace attribute*), 31
num_data_properties (*dicee.dataset_classes.LiteralDataset attribute*), 45
num_data_properties (*dicee.LiteralDataset attribute*), 217
num_datapoints (*dicee.BPE_NegativeSamplingDataset attribute*), 206
num_datapoints (*dicee.dataset_classes.BPE_NegativeSamplingDataset attribute*), 34
num_datapoints (*dicee.dataset_classes.MultiLabelDataset attribute*), 34
num_datapoints (*dicee.MultiLabelDataset attribute*), 206
num_ent (*dicee.DualE attribute*), 181
num_ent (*dicee.models.DualE attribute*), 146
num_ent (*dicee.models.dualE.DualE attribute*), 77
num_entities (*dicee.BaseKGE attribute*), 195
num_entities (*dicee.CVDataModule attribute*), 213
num_entities (*dicee.dataset_classes.CVDataModule attribute*), 41
num_entities (*dicee.dataset_classes.KvsSampleDataset attribute*), 39
num_entities (*dicee.dataset_classes.LiteralDataset attribute*), 45, 46
num_entities (*dicee.dataset_classes.NegSampleDataset attribute*), 40
num_entities (*dicee.dataset_classes.OnevsSample attribute*), 37, 38
num_entities (*dicee.dataset_classes.TriplePredictionDataset attribute*), 41
num_entities (*dicee.evaluator.Evaluator attribute*), 48
num_entities (*dicee.knowledge_graph.KG attribute*), 52
num_entities (*dicee.KvsSampleDataset attribute*), 211
num_entities (*dicee.LiteralDataset attribute*), 217, 218
num_entities (*dicee.models.base_model.BaseKGE attribute*), 64
num_entities (*dicee.models.BaseKGE attribute*), 107, 110, 114, 119, 125, 138, 141
num_entities (*dicee.NegSampleDataset attribute*), 212
num_entities (*dicee.OnevsSample attribute*), 209, 210
num_entities (*dicee.TriplePredictionDataset attribute*), 213
num_epochs (*dicee.abstracts.AbstractPPECallback attribute*), 17
num_epochs (*dicee.analyse_experiments.Experiment attribute*), 19
num_epochs (*dicee.callbacks.ASWA attribute*), 24
num_epochs (*dicee.config.Namespace attribute*), 30
num_epochs (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 169
num_folds_for_cv (*dicee.config.Namespace attribute*), 31
num_of_data_points (*dicee.dataset_classes.MultiClassClassificationDataset attribute*), 35
num_of_data_points (*dicee.MultiClassClassificationDataset attribute*), 207
num_of_data_properties (*dicee.models.literal.LiteralEmbeddings attribute*), 82, 83
num_of_epochs (*dicee.callbacks.PseudoLabellingCallback attribute*), 24
num_of_output_channels (*dicee.BaseKGE attribute*), 195
num_of_output_channels (*dicee.config.Namespace attribute*), 31
num_of_output_channels (*dicee.models.base_model.BaseKGE attribute*), 65
num_of_output_channels (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 141
num_params (*dicee.analyse_experiments.Experiment attribute*), 19
num_relations (*dicee.BaseKGE attribute*), 195
num_relations (*dicee.CVDataModule attribute*), 213
num_relations (*dicee.dataset_classes.CVDataModule attribute*), 41
num_relations (*dicee.dataset_classes.NegSampleDataset attribute*), 40
num_relations (*dicee.dataset_classes.OnevsSample attribute*), 38
num_relations (*dicee.dataset_classes.TriplePredictionDataset attribute*), 41
num_relations (*dicee.evaluator.Evaluator attribute*), 48
num_relations (*dicee.knowledge_graph.KG attribute*), 52
num_relations (*dicee.models.base_model.BaseKGE attribute*), 64
num_relations (*dicee.models.BaseKGE attribute*), 107, 111, 114, 119, 125, 138, 141

num_relations (*dicee.NegSampleDataset attribute*), 212
num_relations (*dicee.OnevsSample attribute*), 210
num_relations (*dicee.TriplePredictionDataset attribute*), 213
num_sample (*dicee.models.FMult attribute*), 143
num_sample (*dicee.models.function_space.FMult attribute*), 79
num_sample (*dicee.models.function_space.GFMult attribute*), 79
num_sample (*dicee.models.GFMult attribute*), 143
num_tokens (*dicee.BaseKGE attribute*), 195
num_tokens (*dicee.knowledge_graph.KG attribute*), 52
num_tokens (*dicee.models.base_model.BaseKGE attribute*), 65
num_tokens (*dicee.models.BaseKGE attribute*), 107, 111, 114, 119, 125, 138, 141
num_workers (*dicee.CVDataModule attribute*), 213
num_workers (*dicee.dataset_classes.CVDataModule attribute*), 41
numpy_data_type_changer() (*in module dicee*), 198
numpy_data_type_changer() (*in module dicee.static_funcs*), 161

O

octonion_mul() (*in module dicee.models*), 127
octonion_mul() (*in module dicee.models.octonion*), 84
octonion_mul_norm() (*in module dicee.models*), 127
octonion_mul_norm() (*in module dicee.models.octonion*), 84
octonion_normalizer() (*dicee.AConvO static method*), 184
octonion_normalizer() (*dicee.ConvO static method*), 186
octonion_normalizer() (*dicee.models.AConvO static method*), 130
octonion_normalizer() (*dicee.models.ConvO static method*), 129
octonion_normalizer() (*dicee.models.octonion.AConvO static method*), 86
octonion_normalizer() (*dicee.models.octonion.ConvO static method*), 86
octonion_normalizer() (*dicee.models.octonion.OMult static method*), 85
octonion_normalizer() (*dicee.models.OMult static method*), 128
octonion_normalizer() (*dicee.OMult static method*), 189
OMult (*class in dicee*), 188
OMult (*class in dicee.models*), 127
OMult (*class in dicee.models.octonion*), 84
on_epoch_end() (*dicee.callbacks.KGESaveCallback method*), 24
on_epoch_end() (*dicee.callbacks.PseudoLabellingCallback method*), 24
on_fit_end() (*dicee.abstracts.AbstractCallback method*), 17
on_fit_end() (*dicee.abstracts.AbstractPPECallback method*), 18
on_fit_end() (*dicee.abstracts.AbstractTrainer method*), 13
on_fit_end() (*dicee.callbacks.AccumulateEpochLossCallback method*), 21
on_fit_end() (*dicee.callbacks.ASWA method*), 24
on_fit_end() (*dicee.callbacks.Eval method*), 26
on_fit_end() (*dicee.callbacks.KGESaveCallback method*), 23
on_fit_end() (*dicee.callbacks.LRScheduler method*), 28
on_fit_end() (*dicee.callbacks.PeriodicEvalCallback method*), 28
on_fit_end() (*dicee.callbacks.PrintCallback method*), 22
on_fit_end() (*dicee.callbacks.SWA method*), 29
on_fit_start() (*dicee.abstracts.AbstractCallback method*), 16
on_fit_start() (*dicee.abstracts.AbstractPPECallback method*), 18
on_fit_start() (*dicee.abstracts.AbstractTrainer method*), 13
on_fit_start() (*dicee.callbacks.Eval method*), 25
on_fit_start() (*dicee.callbacks.KGESaveCallback method*), 23
on_fit_start() (*dicee.callbacks.KronE method*), 27
on_fit_start() (*dicee.callbacks.PrintCallback method*), 22
on_init_end() (*dicee.abstracts.AbstractCallback method*), 16
on_init_start() (*dicee.abstracts.AbstractCallback method*), 16
on_train_batch_end() (*dicee.abstracts.AbstractCallback method*), 17
on_train_batch_end() (*dicee.abstracts.AbstractTrainer method*), 13
on_train_batch_end() (*dicee.callbacks.Eval method*), 26
on_train_batch_end() (*dicee.callbacks.KGESaveCallback method*), 23
on_train_batch_end() (*dicee.callbacks.LRScheduler method*), 28
on_train_batch_end() (*dicee.callbacks.PrintCallback method*), 22
on_train_batch_start() (*dicee.callbacks.Perturb method*), 27
on_train_epoch_end() (*dicee.abstracts.AbstractCallback method*), 17
on_train_epoch_end() (*dicee.abstracts.AbstractTrainer method*), 13
on_train_epoch_end() (*dicee.callbacks.ASWA method*), 25
on_train_epoch_end() (*dicee.callbacks.Eval method*), 26
on_train_epoch_end() (*dicee.callbacks.KGESaveCallback method*), 23

on_train_epoch_end() (*dicee.callbacks.PeriodicEvalCallback method*), 28
on_train_epoch_end() (*dicee.callbacks.PrintCallback method*), 22
on_train_epoch_end() (*dicee.callbacks.SWA method*), 29
on_train_epoch_end() (*dicee.models.base_model.BaseKGELightning method*), 60
on_train_epoch_end() (*dicee.models.BaseKGELightning method*), 103
on_train_epoch_start() (*dicee.abstracts.AbstractTrainer method*), 13
on_train_epoch_start() (*dicee.callbacks.SWA method*), 29
on_train_start() (*dicee.callbacks.LRScheduler method*), 28
OnevsAllDataset (*class in dicee*), 207
OnevsAllDataset (*class in dicee.dataset_classes*), 35
OnevsSample (*class in dicee*), 209
OnevsSample (*class in dicee.dataset_classes*), 37
optim (*dicee.config.Namespace attribute*), 30
optimizer (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 168
optimizer (*dicee.trainer.torch_trainer.TorchTrainer attribute*), 167
optimizer_name (*dicee.BaseKGE attribute*), 195
optimizer_name (*dicee.models.base_model.BaseKGE attribute*), 65
optimizer_name (*dicee.models.BaseKGE attribute*), 107, 111, 114, 119, 125, 138, 141
ordered_bpe_entities (*dicee.BPE_NegativeSamplingDataset attribute*), 206
ordered_bpe_entities (*dicee.dataset_classes.BPE_NegativeSamplingDataset attribute*), 34
ordered_bpe_entities (*dicee.knowledge_graph.KG attribute*), 53
ordered_shaped_bpe_tokens (*dicee.knowledge_graph.KG attribute*), 52

P

p (*dicee.config.Namespace attribute*), 31
p (*dicee.DeCaL attribute*), 178
p (*dicee.Keci attribute*), 175
p (*dicee.models.clifford.DeCaL attribute*), 71
p (*dicee.models.clifford.Keci attribute*), 68
p (*dicee.models.DeCaL attribute*), 134
p (*dicee.models.Keci attribute*), 131
padding (*dicee.knowledge_graph.KG attribute*), 52
pandas_dataframe_indexer() (*in module dicee.read_preprocess_save_load_kg.util*), 152
param_init (*dicee.BaseKGE attribute*), 195
param_init (*dicee.models.base_model.BaseKGE attribute*), 65
param_init (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 142
parameters() (*dicee.abstracts.BaseInteractiveKGE method*), 16
parameters() (*dicee.EnsembleKGE method*), 197
parameters() (*dicee.models.ensemble.EnsembleKGE method*), 78
path (*dicee.abstracts.AbstractPPECallback attribute*), 17
path (*dicee.callbacks.AccumulateEpochLossCallback attribute*), 21
path (*dicee.callbacks.ASWA attribute*), 24
path (*dicee.callbacks.Eval attribute*), 25
path (*dicee.callbacks.KGESaveCallback attribute*), 23
path_dataset_folder (*dicee.analyse_experiments.Experiment attribute*), 19
path_for_deserialization (*dicee.knowledge_graph.KG attribute*), 52
path_for_serialization (*dicee.knowledge_graph.KG attribute*), 52
path_single_kg (*dicee.config.Namespace attribute*), 30
path_single_kg (*dicee.knowledge_graph.KG attribute*), 52
path_to_store_single_run (*dicee.config.Namespace attribute*), 29
PeriodicEvalCallback (*class in dicee.callbacks*), 27
Perturb (*class in dicee.callbacks*), 27
polars_dataframe_indexer() (*in module dicee.read_preprocess_save_load_kg.util*), 151
poly_NN() (*dicee.LFMult method*), 190
poly_NN() (*dicee.models.function_space.LFMult method*), 81
poly_NN() (*dicee.models.LFMult method*), 145
polynomial() (*dicee.LFMult method*), 191
polynomial() (*dicee.models.function_space.LFMult method*), 82
polynomial() (*dicee.models.LFMult method*), 146
pop() (*dicee.LFMult method*), 191
pop() (*dicee.models.function_space.LFMult method*), 82
pop() (*dicee.models.LFMult method*), 146
pq (*dicee.analyse_experiments.Experiment attribute*), 20
predict() (*dicee.KGE method*), 202
predict() (*dicee.knowledge_graph_embeddings.KGE method*), 55
predict_dataloader() (*dicee.models.base_model.BaseKGELightning method*), 61
predict_dataloader() (*dicee.models.BaseKGELightning method*), 104

predict_literals() (*dicee.KGE method*), 203
predict_literals() (*dicee.knowledge_graph_embeddings.KGE method*), 56
predict_missing_head_entity() (*dicee.KGE method*), 200
predict_missing_head_entity() (*dicee.knowledge_graph_embeddings.KGE method*), 53
predict_missing_relations() (*dicee.KGE method*), 201
predict_missing_relations() (*dicee.knowledge_graph_embeddings.KGE method*), 54
predict_missing_tail_entity() (*dicee.KGE method*), 201
predict_missing_tail_entity() (*dicee.knowledge_graph_embeddings.KGE method*), 54
predict_topk() (*dicee.KGE method*), 202
predict_topk() (*dicee.knowledge_graph_embeddings.KGE method*), 55
prepare_data() (*dicee.CVDataModule method*), 215
prepare_data() (*dicee.dataset_classes.CVDataModule method*), 43
preprocess_with_byte_pair_encoding() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 154
preprocess_with_byte_pair_encoding() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 149
preprocess_with_byte_pair_encoding_with_padding() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 154
preprocess_with_byte_pair_encoding_with_padding() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 149
preprocess_with_pandas() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 154
preprocess_with_pandas() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 149
preprocess_with_polars() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 155
preprocess_with_polars() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 149
preprocesses_input_args() (*in module dicee.static_preprocess_funcs*), 163
PreprocessKG (*class in dicee.read_preprocess_save_load_kg*), 154
PreprocessKG (*class in dicee.read_preprocess_save_load_kg.preprocess*), 149
PrintCallback (*class in dicee.callbacks*), 21
process (*dicee.trainer.torch_trainer.TorchTrainer attribute*), 167
PseudoLabellingCallback (*class in dicee.callbacks*), 24
Pyke (*class in dicee*), 173
Pyke (*class in dicee.models*), 113
Pyke (*class in dicee.models.real*), 92
pykeen_model_kwargs (*dicee.config.Namespace attribute*), 31
PykeenKGE (*class in dicee*), 191
PykeenKGE (*class in dicee.models*), 139
PykeenKGE (*class in dicee.models.pykeen_models*), 87

Q

q (*dicee.config.Namespace attribute*), 31
q (*dicee.DeCaL attribute*), 178
q (*dicee.Keci attribute*), 175
q (*dicee.models.clifford.DeCaL attribute*), 71
q (*dicee.models.clifford.Keci attribute*), 68
q (*dicee.models.DeCaL attribute*), 134
q (*dicee.models.Keci attribute*), 131
qdrant_client (*dicee.scripts.index_serve.NeuralSearcher attribute*), 157
QMult (*class in dicee*), 187
QMult (*class in dicee.models*), 121
QMult (*class in dicee.models.quaternion*), 88
quaternion_mul() (*in module dicee.models*), 118
quaternion_mul() (*in module dicee.models.static_funcs*), 92
quaternion_mul_with_unit_norm() (*in module dicee.models*), 121
quaternion_mul_with_unit_norm() (*in module dicee.models.quaternion*), 88
quaternion_multiplication_followed_by_inner_product() (*dicee.models.QMult method*), 122
quaternion_multiplication_followed_by_inner_product() (*dicee.models.quaternion.QMult method*), 89
quaternion_multiplication_followed_by_inner_product() (*dicee.QMult method*), 188
quaternion_normalizer() (*dicee.models.QMult static method*), 122
quaternion_normalizer() (*dicee.models.quaternion.QMult static method*), 89
quaternion_normalizer() (*dicee.QMult static method*), 188
queries (*dicee.scripts.index_serve.StringListRequest attribute*), 158
query_name_to_struct (*dicee.query_generator.QueryGenerator attribute*), 147
query_name_to_struct (*dicee.QueryGenerator attribute*), 219
QueryGenerator (*class in dicee*), 218
QueryGenerator (*class in dicee.query_generator*), 147

R

r (*dicee.DeCaL attribute*), 178
r (*dicee.Keci attribute*), 175
r (*dicee.models.clifford.DeCaL attribute*), 71
r (*dicee.models.clifford.Keci attribute*), 68

r (*dicee.models.DeCaL attribute*), 134
r (*dicee.models.Keci attribute*), 131
random_prediction() (*in module dicee*), 198
random_prediction() (*in module dicee.static_funcs*), 161
random_seed (*dicee.config.Namespace attribute*), 31
ratio (*dicee.callbacks.Perturb attribute*), 27
re (*dicee.DeCaL attribute*), 178
re (*dicee.models.clifford.DeCaL attribute*), 71
re (*dicee.models.DeCaL attribute*), 134
re_vocab (*dicee.evaluator.Evaluator attribute*), 48
read_from_disk() (*in module dicee.read_preprocess_save_load_kg.util*), 153
read_from_triple_store() (*in module dicee.read_preprocess_save_load_kg.util*), 153
read_only_few (*dicee.config.Namespace attribute*), 31
read_only_few (*dicee.knowledge_graph.KG attribute*), 52
read_or_load_kg() (*in module dicee*), 198
read_or_load_kg() (*in module dicee.static_funcs*), 161
read_with_pandas() (*in module dicee.read_preprocess_save_load_kg.util*), 153
read_with_polars() (*in module dicee.read_preprocess_save_load_kg.util*), 153
ReadFromDisk (*class in dicee.read_preprocess_save_load_kg*), 155
ReadFromDisk (*class in dicee.read_preprocess_save_load_kg.read_from_disk*), 149
reducer (*dicee.scripts.index_serve.StringListRequest attribute*), 158
rel2id (*dicee.query_generator.QueryGenerator attribute*), 147
rel2id (*dicee.QueryGenerator attribute*), 218
relation_embeddings (*dicee.AConvQ attribute*), 184
relation_embeddings (*dicee.ConvQ attribute*), 185
relation_embeddings (*dicee.DeCaL attribute*), 178
relation_embeddings (*dicee.DualE attribute*), 181
relation_embeddings (*dicee.LFMult attribute*), 190
relation_embeddings (*dicee.models.AConvQ attribute*), 124
relation_embeddings (*dicee.models.clifford.DeCaL attribute*), 71
relation_embeddings (*dicee.models.ConvQ attribute*), 123
relation_embeddings (*dicee.models.DeCaL attribute*), 134
relation_embeddings (*dicee.models.DualE attribute*), 146
relation_embeddings (*dicee.models.dualE.DualE attribute*), 77
relation_embeddings (*dicee.models.FMult attribute*), 143
relation_embeddings (*dicee.models.FMult2 attribute*), 144
relation_embeddings (*dicee.models.function_space.FMult attribute*), 79
relation_embeddings (*dicee.models.function_space.FMult2 attribute*), 80
relation_embeddings (*dicee.models.function_space.GFMult attribute*), 79
relation_embeddings (*dicee.models.function_space.LFMult attribute*), 81
relation_embeddings (*dicee.models.function_space.LFMult1 attribute*), 80
relation_embeddings (*dicee.models.GFMult attribute*), 143
relation_embeddings (*dicee.models.LFMult attribute*), 145
relation_embeddings (*dicee.models.LFMult1 attribute*), 144
relation_embeddings (*dicee.models.pykeen_models.PykeenKGE attribute*), 87
relation_embeddings (*dicee.models.PykeenKGE attribute*), 140
relation_embeddings (*dicee.models.quaternion.AConvQ attribute*), 90
relation_embeddings (*dicee.models.quaternion.ConvQ attribute*), 90
relation_embeddings (*dicee.PykeenKGE attribute*), 191
relation_to_idx (*dicee.knowledge_graph.KG attribute*), 52
relations_str (*dicee.knowledge_graph.KG property*), 53
reload_dataset() (*in module dicee*), 205
reload_dataset() (*in module dicee.dataset_classes*), 33
report (*dicee.DICE_Trainer attribute*), 199
report (*dicee.evaluator.Evaluator attribute*), 48
report (*dicee.Execute attribute*), 204
report (*dicee.executer.Execute attribute*), 50
report (*dicee.trainer.DICE_Trainer attribute*), 169
report (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 164
reports (*dicee.callbacks.Eval attribute*), 25
reports (*dicee.callbacks.PeriodicEvalCallback attribute*), 27
requires_grad_for_interactions (*dicee.CKeci attribute*), 174
requires_grad_for_interactions (*dicee.Keci attribute*), 175
requires_grad_for_interactions (*dicee.models.CKeci attribute*), 133
requires_grad_for_interactions (*dicee.models.clifford.CKeci attribute*), 70
requires_grad_for_interactions (*dicee.models.clifford.Keci attribute*), 68
requires_grad_for_interactions (*dicee.models.Keci attribute*), 131
resid_dropout (*dicee.models.transformers.CausalSelfAttention attribute*), 96

residual_convolution() (*dicee.AConEx method*), 183
residual_convolution() (*dicee.AConvO method*), 184
residual_convolution() (*dicee.AConvQ method*), 184
residual_convolution() (*dicee.ConEx method*), 187
residual_convolution() (*dicee.ConvO method*), 186
residual_convolution() (*dicee.ConvQ method*), 185
residual_convolution() (*dicee.models.AConEx method*), 117
residual_convolution() (*dicee.models.AConvO method*), 130
residual_convolution() (*dicee.models.AConvQ method*), 124
residual_convolution() (*dicee.models.complex.AConEx method*), 75
residual_convolution() (*dicee.models.complex.ConEx method*), 75
residual_convolution() (*dicee.models.ConEx method*), 116
residual_convolution() (*dicee.models.ConvO method*), 129
residual_convolution() (*dicee.models.ConvQ method*), 123
residual_convolution() (*dicee.models.octonion.AConvO method*), 86
residual_convolution() (*dicee.models.octonion.ConvO method*), 86
residual_convolution() (*dicee.models.quaternion.AConvQ method*), 91
residual_convolution() (*dicee.models.quaternion.ConvQ method*), 90
retrieve_embedding() (*dicee.scripts.index_serve.NeuralSearcher method*), 157
retrieve_embeddings() (*in module dicee.scripts.index_serve*), 157
return_multi_hop_query_results() (*dicee.KGE method*), 202
return_multi_hop_query_results() (*dicee.knowledge_graph_embeddings.KGE method*), 55
root() (*in module dicee.scripts.index_serve*), 157
roots (*dicee.models.FMult attribute*), 143
roots (*dicee.models.function_space.FMult attribute*), 79
roots (*dicee.models.function_space.GFMult attribute*), 79
roots (*dicee.models.GFMult attribute*), 143
runtime (*dicee.analyse_experiments.Experiment attribute*), 20

S

sample_counter (*dicee.abstracts.AbstractPPECallback attribute*), 17
sample_entity() (*dicee.abstracts.BaseInteractiveKGE method*), 15
sample_relation() (*dicee.abstracts.BaseInteractiveKGE method*), 15
sample_triples_ratio (*dicee.config.Namespace attribute*), 31
sample_triples_ratio (*dicee.knowledge_graph.KG attribute*), 52
sampling_ratio (*dicee.dataset_classes.LiteralDataset attribute*), 45, 46
sampling_ratio (*dicee.LiteralDataset attribute*), 217, 218
sanity_check_callback_args() (*in module dicee.sanity_checkers*), 156
sanity_checking_with_arguments() (*in module dicee.sanity_checkers*), 156
save() (*dicee.abstracts.BaseInteractiveKGE method*), 15
save() (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk method*), 155
save() (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk method*), 150
save_checkpoint() (*dicee.abstracts.AbstractTrainer static method*), 14
save_checkpoint_model() (*in module dicee*), 198
save_checkpoint_model() (*in module dicee.static_funcs*), 161
save_embeddings() (*in module dicee*), 198
save_embeddings() (*in module dicee.static_funcs*), 161
save_embeddings_as_csv (*dicee.config.Namespace attribute*), 29
save_every_n_epochs (*dicee.config.Namespace attribute*), 32
save_experiment() (*dicee.analyse_experiments.Experiment method*), 20
save_model_at_every_epoch (*dicee.config.Namespace attribute*), 31
save_model_every_n_epoch (*dicee.callbacks.PeriodicEvalCallback attribute*), 27
save_numpy_ndarray() (*in module dicee*), 198
save_numpy_ndarray() (*in module dicee.read_preprocess_save_load_kg.util*), 153
save_numpy_ndarray() (*in module dicee.static_funcs*), 161
save_pickle() (*in module dicee*), 197
save_pickle() (*in module dicee.read_preprocess_save_load_kg.util*), 154
save_pickle() (*in module dicee.static_funcs*), 161
save_queries() (*dicee.query_generator.QueryGenerator method*), 148
save_queries() (*dicee.QueryGenerator method*), 219
save_queries_and_answers() (*dicee.query_generator.QueryGenerator static method*), 148
save_queries_and_answers() (*dicee.QueryGenerator static method*), 219
save_trained_model() (*dicee.Execute method*), 204
save_trained_model() (*dicee.executer.Execute method*), 50
scalar_batch_NN() (*dicee.LFMult method*), 190
scalar_batch_NN() (*dicee.models.function_space.LFMult method*), 81
scalar_batch_NN() (*dicee.models.LFMult method*), 145

scaler (*dicee.callbacks.Perturb attribute*), 27
scaler (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 169
scheduler (*dicee.callbacks.LRScheduler attribute*), 28
score() (*dicee.ComplEx static method*), 183
score() (*dicee.DistMult method*), 174
score() (*dicee.Keci method*), 177
score() (*dicee.models.clifford.Keci method*), 70
score() (*dicee.models.ComplEx static method*), 118
score() (*dicee.models.complex.ComplEx static method*), 76
score() (*dicee.models.DistMult method*), 113
score() (*dicee.models.Keci method*), 133
score() (*dicee.models.octonion.OMult method*), 85
score() (*dicee.models.OMult method*), 128
score() (*dicee.models.QMult method*), 123
score() (*dicee.models.quaternion.QMult method*), 89
score() (*dicee.models.real.DistMult method*), 91
score() (*dicee.models.real.TransE method*), 92
score() (*dicee.models.TransE method*), 113
score() (*dicee.OMult method*), 189
score() (*dicee.QMult method*), 188
score() (*dicee.TransE method*), 177
score_func (*dicee.models.FMult2 attribute*), 144
score_func (*dicee.models.function_space.FMult2 attribute*), 80
scoring_technique (*dicee.analyse_experiments.Experiment attribute*), 20
scoring_technique (*dicee.config.Namespace attribute*), 30
search() (*dicee.scripts.index_serve.NeuralSearcher method*), 157
search_embeddings() (*in module dicee.scripts.index_serve*), 157
search_embeddings_batch() (*in module dicee.scripts.index_serve*), 159
seed (*dicee.query_generator.QueryGenerator attribute*), 147
seed (*dicee.QueryGenerator attribute*), 218
select_model() (*in module dicee*), 197
select_model() (*in module dicee.static_funcs*), 161
selected_optimizer (*dicee.BaseKGE attribute*), 195
selected_optimizer (*dicee.models.base_model.BaseKGE attribute*), 65
selected_optimizer (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 141
separator (*dicee.config.Namespace attribute*), 30
separator (*dicee.knowledge_graph.KG attribute*), 53
sequential_vocabulary_construction() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 155
sequential_vocabulary_construction() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 149
serve() (*in module dicee.scripts.index_serve*), 159
set_global_seed() (*dicee.query_generator.QueryGenerator method*), 148
set_global_seed() (*dicee.QueryGenerator method*), 219
set_model_eval_mode() (*dicee.abstracts.BaseInteractiveKGE method*), 15
set_model_train_mode() (*dicee.abstracts.BaseInteractiveKGE method*), 14
setup() (*dicee.CVDataModule method*), 214
setup() (*dicee.dataset_classes.CVDataModule method*), 42
setup_executor() (*dicee.Execute method*), 204
setup_executor() (*dicee.executer.Execute method*), 50
Shallom (*class in dicee*), 189
Shallom (*class in dicee.models*), 113
Shallom (*class in dicee.models.real*), 92
shallom (*dicee.models.real.Shallom attribute*), 92
shallom (*dicee.models.Shallom attribute*), 113
shallom (*dicee.Shallom attribute*), 189
single_hop_query_answering() (*dicee.KGE method*), 202
single_hop_query_answering() (*dicee.knowledge_graph_embeddings.KGE method*), 55
snapshot_dir (*dicee.callbacks.LRScheduler attribute*), 28
snapshot_loss (*dicee.callbacks.LRScheduler attribute*), 28
sparql_endpoint (*dicee.config.Namespace attribute*), 30
sparql_endpoint (*dicee.knowledge_graph.KG attribute*), 52
start() (*dicee.DICE_Trainer method*), 200
start() (*dicee.Execute method*), 205
start() (*dicee.executer.Execute method*), 51
start() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 154
start() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 149
start() (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method*), 149
start() (*dicee.read_preprocess_save_load_kg.ReadFromDisk method*), 155
start() (*dicee.trainer.DICE_Trainer method*), 170

start() (*dicee.trainer.dice_trainer.DICE_Trainer method*), 165
start_time (*dicee.callbacks.PrintCallback attribute*), 22
start_time (*dicee.Execute attribute*), 204
start_time (*dicee.executer.Execute attribute*), 50
state_dict() (*dicee.EnsembleKGE method*), 197
state_dict() (*dicee.models.ensemble.EnsembleKGE method*), 78
step() (*dicee.EnsembleKGE method*), 197
step() (*dicee.models.ADOPT method*), 101
step() (*dicee.models.adopt.ADOPT method*), 58
step() (*dicee.models.ensemble.EnsembleKGE method*), 78
step_count (*dicee.callbacks.LRScheduler attribute*), 28
storage_path (*dicee.config.Namespace attribute*), 29
storage_path (*dicee.DICE_Trainer attribute*), 199
storage_path (*dicee.trainer.DICE_Trainer attribute*), 169
storage_path (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 165
store() (*in module dicee*), 198
store() (*in module dicee.static_funcs*), 161
store_ensemble() (*dicee.abstracts.AbstractPPECallback method*), 18
strategy (*dicee.abstracts.AbstractTrainer attribute*), 13
StringListRequest (*class in dicee.scripts.index_serve*), 157
SWA (*class in dicee.callbacks*), 28
swa (*dicee.config.Namespace attribute*), 32
swa_c_epochs (*dicee.callbacks.SWA attribute*), 29
swa_lr (*dicee.callbacks.SWA attribute*), 29
swa_model (*dicee.callbacks.SWA attribute*), 29
swa_n (*dicee.callbacks.SWA attribute*), 29
swa_start_epoch (*dicee.callbacks.SWA attribute*), 29
swa_start_epoch (*dicee.config.Namespace attribute*), 32

T

T() (*dicee.DualE method*), 182
T() (*dicee.models.DualE method*), 147
T() (*dicee.models.dualE.DualE method*), 78
t_conorm() (*dicee.abstracts.InteractiveQueryDecomposition method*), 16
t_norm() (*dicee.abstracts.InteractiveQueryDecomposition method*), 16
target_dim (*dicee.AllvsAll attribute*), 209
target_dim (*dicee.dataset_classes.AllvsAll attribute*), 37
target_dim (*dicee.dataset_classes.MultiLabelDataset attribute*), 34
target_dim (*dicee.dataset_classes.OnevsAllDataset attribute*), 35
target_dim (*dicee.knowledge_graph.KG attribute*), 52
target_dim (*dicee.MultiLabelDataset attribute*), 206
target_dim (*dicee.OnevsAllDataset attribute*), 207
temperature (*dicee.BytE attribute*), 193
temperature (*dicee.models.transformers.BytE attribute*), 94
tensor_t_norm() (*dicee.abstracts.InteractiveQueryDecomposition method*), 16
TensorParallel (*class in dicee.trainer.model_parallelism*), 166
test_dataloader() (*dicee.models.base_model.BaseKGELightning method*), 60
test_dataloader() (*dicee.models.BaseKGELightning method*), 103
test_epoch_end() (*dicee.models.base_model.BaseKGELightning method*), 60
test_epoch_end() (*dicee.models.BaseKGELightning method*), 103
test_h1 (*dicee.analyse_experiments.Experiment attribute*), 20
test_h3 (*dicee.analyse_experiments.Experiment attribute*), 20
test_h10 (*dicee.analyse_experiments.Experiment attribute*), 20
test_mrr (*dicee.analyse_experiments.Experiment attribute*), 20
test_path (*dicee.query_generator.QueryGenerator attribute*), 147
test_path (*dicee.QueryGenerator attribute*), 218
timeit() (*in module dicee*), 197, 205
timeit() (*in module dicee.read_preprocess_save_load_kg.util*), 153
timeit() (*in module dicee.static_funcs*), 161
timeit() (*in module dicee.static_preprocess_funcs*), 163
to() (*dicee.EnsembleKGE method*), 197
to() (*dicee.KGE method*), 200
to() (*dicee.knowledge_graph_embeddings.KGE method*), 53
to() (*dicee.models.ensemble.EnsembleKGE method*), 78
to_df() (*dicee.analyse_experiments.Experiment method*), 20
topk (*dicee.BytE attribute*), 193
topk (*dicee.models.transformers.BytE attribute*), 94

topk (*dicee.scripts.index_serve.NeuralSearcher attribute*), 157
torch_ordered_shaped_bpe_entities (*dicee.dataset_classes.MultiLabelDataset attribute*), 34
torch_ordered_shaped_bpe_entities (*dicee.MultiLabelDataset attribute*), 206
TorchDDPTrainer (*class in dicee.trainer.torch_trainer_ddp*), 168
TorchTrainer (*class in dicee.trainer.torch_trainer*), 167
total_epochs (*dicee.callbacks.LRScheduler attribute*), 28
total_steps (*dicee.callbacks.LRScheduler attribute*), 28
train() (*dicee.abstracts.BaseInteractiveTrainKGE method*), 18
train() (*dicee.trainer.torch_trainer_ddp.NodeTrainer method*), 169
train_data (*dicee.AllvsAll attribute*), 209
train_data (*dicee.dataset_classes.AllvsAll attribute*), 37
train_data (*dicee.dataset_classes.KvsAll attribute*), 36
train_data (*dicee.dataset_classes.KvsSampleDataset attribute*), 39
train_data (*dicee.dataset_classes.MultiClassClassificationDataset attribute*), 35
train_data (*dicee.dataset_classes.OnevsAllDataset attribute*), 35
train_data (*dicee.dataset_classes.OnevsSample attribute*), 37, 38
train_data (*dicee.KvsAll attribute*), 208
train_data (*dicee.KvsSampleDataset attribute*), 211
train_data (*dicee.MultiClassClassificationDataset attribute*), 207
train_data (*dicee.OnevsAllDataset attribute*), 207
train_data (*dicee.OnevsSample attribute*), 209, 210
train_dataloader() (*dicee.CVDataModule method*), 213
train_dataloader() (*dicee.dataset_classes.CVDataModule method*), 41
train_dataloader() (*dicee.models.base_model.BaseKGELightning method*), 62
train_dataloader() (*dicee.models.BaseKGELightning method*), 105
train_dataloaders (*dicee.trainer.torch_trainer.TorchTrainer attribute*), 167
train_dataset_loader (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 169
train_file_path (*dicee.dataset_classes.LiteralDataset attribute*), 45
train_file_path (*dicee.LiteralDataset attribute*), 217
train_h1 (*dicee.analyse_experiments.Experiment attribute*), 20
train_h3 (*dicee.analyse_experiments.Experiment attribute*), 20
train_h10 (*dicee.analyse_experiments.Experiment attribute*), 20
train_indices_target (*dicee.dataset_classes.MultiLabelDataset attribute*), 34
train_indices_target (*dicee.MultiLabelDataset attribute*), 206
train_k_vs_all() (*dicee.abstracts.BaseInteractiveTrainKGE method*), 18
train_literals() (*dicee.abstracts.BaseInteractiveTrainKGE method*), 18
train_mode (*dicee.EnsembleKGE attribute*), 196
train_mode (*dicee.models.ensemble.EnsembleKGE attribute*), 78
train_mrr (*dicee.analyse_experiments.Experiment attribute*), 20
train_path (*dicee.query_generator.QueryGenerator attribute*), 147
train_path (*dicee.QueryGenerator attribute*), 218
train_set (*dicee.BPE_NegativeSamplingDataset attribute*), 206
train_set (*dicee.dataset_classes.BPE_NegativeSamplingDataset attribute*), 34
train_set (*dicee.dataset_classes.MultiLabelDataset attribute*), 34
train_set (*dicee.dataset_classes.NegSampleDataset attribute*), 40
train_set (*dicee.dataset_classes.TriplePredictionDataset attribute*), 41
train_set (*dicee.MultiLabelDataset attribute*), 206
train_set (*dicee.NegSampleDataset attribute*), 212
train_set (*dicee.TriplePredictionDataset attribute*), 213
train_set_idx (*dicee.CVDataModule attribute*), 213
train_set_idx (*dicee.dataset_classes.CVDataModule attribute*), 41
train_set_target (*dicee.knowledge_graph.KG attribute*), 52
train_target (*dicee.AllvsAll attribute*), 209
train_target (*dicee.dataset_classes.AllvsAll attribute*), 37
train_target (*dicee.dataset_classes.KvsAll attribute*), 36
train_target (*dicee.dataset_classes.KvsSampleDataset attribute*), 39
train_target (*dicee.KvsAll attribute*), 208
train_target (*dicee.KvsSampleDataset attribute*), 211
train_target_indices (*dicee.knowledge_graph.KG attribute*), 53
train_triples (*dicee.dataset_classes.NegSampleDataset attribute*), 40
train_triples (*dicee.NegSampleDataset attribute*), 212
train_triples() (*dicee.abstracts.BaseInteractiveTrainKGE method*), 18
trained_model (*dicee.Execute attribute*), 204
trained_model (*dicee.executer.Execute attribute*), 50
trainer (*dicee.config.Namespace attribute*), 30
trainer (*dicee.DICE_Trainer attribute*), 199
trainer (*dicee.Execute attribute*), 204
trainer (*dicee.executer.Execute attribute*), 50

trainer (*dicee.trainer.DICE_Trainer attribute*), 169
trainer (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 164
trainer (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 168
training_step (*dicee.trainer.torch_trainer.TorchTrainer attribute*), 167
training_step() (*dicee.BytE method*), 193
training_step() (*dicee.models.base_model.BaseKGELightning method*), 59
training_step() (*dicee.models.BaseKGELightning method*), 102
training_step() (*dicee.models.transformers.BytE method*), 94
training_step_outputs (*dicee.models.base_model.BaseKGELightning attribute*), 59
training_step_outputs (*dicee.models.BaseKGELightning attribute*), 102
training_technique (*dicee.knowledge_graph.KG attribute*), 52
TransE (*class in dicee*), 177
TransE (*class in dicee.models*), 113
TransE (*class in dicee.models.real*), 91
transfer_batch_to_device() (*dicee.CVDataModule method*), 214
transfer_batch_to_device() (*dicee.dataset_classes.CVDataModule method*), 42
transformer (*dicee.BytE attribute*), 193
transformer (*dicee.models.transformers.BytE attribute*), 94
transformer (*dicee.models.transformers.GPT attribute*), 99
trapezoid() (*dicee.models.FMult2 method*), 144
trapezoid() (*dicee.models.function_space.FMult2 method*), 80
tri_score() (*dicee.LFMult method*), 190
tri_score() (*dicee.models.function_space.LFMult method*), 81
tri_score() (*dicee.models.function_space.LFMult1 method*), 81
tri_score() (*dicee.models.LFMult method*), 145
tri_score() (*dicee.models.LFMult1 method*), 145
triple_score() (*dicee.KGE method*), 202
triple_score() (*dicee.knowledge_graph_embeddings.KGE method*), 55
TriplePredictionDataset (*class in dicee*), 212
TriplePredictionDataset (*class in dicee.dataset_classes*), 40
tuple2list() (*dicee.query_generator.QueryGenerator method*), 148
tuple2list() (*dicee.QueryGenerator method*), 219

U

unlabelled_size (*dicee.callbacks.PseudoLabellingCallback attribute*), 24
unmap() (*dicee.query_generator.QueryGenerator method*), 148
unmap() (*dicee.QueryGenerator method*), 219
unmap_query() (*dicee.query_generator.QueryGenerator method*), 148
unmap_query() (*dicee.QueryGenerator method*), 219

V

val_aswa (*dicee.callbacks.ASWA attribute*), 24
val_dataloader() (*dicee.models.base_model.BaseKGELightning method*), 61
val_dataloader() (*dicee.models.BaseKGELightning method*), 104
val_h1 (*dicee.analyse_experiments.Experiment attribute*), 20
val_h3 (*dicee.analyse_experiments.Experiment attribute*), 20
val_h10 (*dicee.analyse_experiments.Experiment attribute*), 20
val_mrr (*dicee.analyse_experiments.Experiment attribute*), 20
val_path (*dicee.query_generator.QueryGenerator attribute*), 147
val_path (*dicee.QueryGenerator attribute*), 218
validate_knowledge_graph() (*in module dicee.sanity_checkers*), 156
vocab_preparation() (*dicee.evaluator.Evaluator method*), 49
vocab_size (*dicee.models.transformers.GPTConfig attribute*), 98
vocab_to_parquet() (*in module dicee*), 198
vocab_to_parquet() (*in module dicee.static_funcs*), 162
vtp_score() (*dicee.LFMult method*), 190
vtp_score() (*dicee.models.function_space.LFMult method*), 81
vtp_score() (*dicee.models.function_space.LFMult1 method*), 81
vtp_score() (*dicee.models.LFMult method*), 145
vtp_score() (*dicee.models.LFMult1 method*), 145

W

warmup_steps (*dicee.callbacks.LRScheduler attribute*), 28
weight (*dicee.models.transformers.LayerNorm attribute*), 95
weight_decay (*dicee.BaseKGE attribute*), 195
weight_decay (*dicee.config.Namespace attribute*), 30

weight_decay (*dicee.models.base_model.BaseKGE attribute*), 65
weight_decay (*dicee.models.BaseKGE attribute*), 108, 111, 115, 119, 125, 138, 141
weights (*dicee.models.FMult attribute*), 143
weights (*dicee.models.function_space.FMult attribute*), 79
weights (*dicee.models.function_space.GFMult attribute*), 80
weights (*dicee.models.GFMult attribute*), 144
write_csv_from_model_parallel() (*in module dicee*), 199
write_csv_from_model_parallel() (*in module dicee.static_funcs*), 162
write_links() (*dicee.query_generator.QueryGenerator method*), 148
write_links() (*dicee.QueryGenerator method*), 219
write_report() (*dicee.Execute method*), 205
write_report() (*dicee.executer.Execute method*), 51

X

x_values (*dicee.LFMult attribute*), 190
x_values (*dicee.models.function_space.LFMult attribute*), 81
x_values (*dicee.models.LFMult attribute*), 145