
DICE Embeddings

Release 0.1.3.2

Caglar Demir

Jan 06, 2026

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	215
14.3	Classes	215
14.4	Package Contents	215
	Python Module Index	227

DICE Embeddings¹: 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 Demir²

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-CPU, GPUs, TPUs or computing cluster, and
3. **Huggingface**⁵ to ease the deployment of pre-trained models.

Why Pandas⁶ & 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 PyTorch⁷ & 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 Huggingface¹⁰? Seamlessly deploy and share pre-trained embedding models through the Huggingface ecosystem.

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/>

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 automatically detects available GPUs and trains a model with distributed data parallels technique. Under the hood, dicee uses lightning 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 lightning 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.executor 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, 115

(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)

```

    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

```

```
@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

DICE Embeddings - Knowledge Graph Embedding Library.

A library for training and using knowledge graph embedding models with support for various scoring techniques and training strategies.

Submodules:

evaluation: Model evaluation functions and Evaluator class models: KGE model implementations trainer: Training orchestration scripts: Utility scripts

14.1 Submodules

`dicee.__main__`

`dicee.abstracts`

Classes

<i>AbstractTrainer</i>	Abstract class for Trainer class for knowledge graph embedding models
<i>BaseInteractiveKGE</i>	Abstract/base class for using knowledge graph embedding models interactively.
<i>InteractiveQueryDecomposition</i>	
<i>AbstractCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>AbstractPPECallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>BaseInteractiveTrainKGE</i>	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 end 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 $[0,1]^{n \times d}$ 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, shuffle_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.
- **shuffle_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

Callbacks for training lifecycle events.

Provides callback classes for various training events including epoch end, model saving, weight averaging, and evaluation.

Classes

<i>AccumulateEpochLossCallback</i>	Callback to store epoch losses to a CSV file.
<i>PrintCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>KGESaveCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>PseudoLabellingCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>Eval</i>	Abstract class for Callback class for knowledge graph embedding models
<i>KronE</i>	Abstract class for Callback class for knowledge graph embedding models
<i>Perturb</i>	A callback for a three-Level Perturbation
<i>PeriodicEvalCallback</i>	Callback to periodically evaluate the model and optionally save checkpoints during training.
<i>LRScheduler</i>	Callback for managing learning rate scheduling and model snapshots.

Functions

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

Module Contents

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

Bases: `dicee.abstracts.AbstractCallback`

Callback to store epoch losses to a CSV file.

Parameters

path – Directory path where the loss file will be saved.

path

on_fit_end (*trainer, model*) → None

Store epoch loss history to CSV file.

Parameters

- **trainer** – The trainer instance.

- **model** – The model being trained.

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.**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 must be the same. :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 `diccee.callbacks.Perturb` (*level: str = 'input', ratio: float = 0.0, method: str = None, scaler: float = None, frequency=None*)

Bases: `diccee.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 `diccee.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: `diccee.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

dicee.config

Configuration module for DICE embeddings.

Provides the Namespace class with default configuration values for training knowledge graph embedding models.

Classes

<i>Namespace</i>	Extended Namespace with default KGE training configuration.
------------------	---

Module Contents

```
class dicee.config.Namespace (**kwargs)
```

Bases: argparse.Namespace

Extended Namespace with default KGE training configuration.

Provides sensible defaults for all training parameters while allowing easy customization through command-line arguments or direct assignment.

```
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

swag: bool = False
Stochastic weight averaging - Gaussian

ema: bool = False
Exponential Moving Average

twa: bool = False
Trainable 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.

swa_c_epochs: **int** = **1**
Number of epochs to average over for SWA, SWAG, EMA, TWA.

__iter__()

dicee.dataset_classes

Classes

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

Functions

<i>reload_dataset</i> (path, form_of_labelling, ...)	Reload the files from disk to construct the Pytorch dataset
--	---

continues on next page

Table 8 – continued from previous page

`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` (* (*Keyword-only parameters separator (PEP 3102)*), *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_idx`s – mapping.
- `relation_idx`s – 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_idx)
```

Bases: torch.utils.data.Dataset

Dataset for the 1vsALL training strategy

Parameters

- **train_set_idx** – Indexed triples for the training.
- **entity_idx** – mapping.
- **relation_idx** – 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_idx, relation_idx, 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 a 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.

forall $y_i = 1$ s.t. $(h, r \in E_i)$ in KG

Note

TODO

train_set_idx

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

entity_idx

[dictionary] string representation of an entity to its integer id

relation_idx

[dictionary] 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_idxes, relation_idxes,
    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 $N = |E| \times |R|$ y : denotes a multi-label vector in $[0, 1]^{|E|}$ is a binary label.

orall $y_i = 1$ s.t. $(h, r) \in KG$

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_idxes

[dictionary] string representation of an entity to its integer id

relation_idxes

[dictionary] 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_idxes,
relation_idxes, 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.

forall $y_i = 1$ s.t. (h, r, E_i) in KG

At each mini-batch construction, we subsample(y), hence n

$|new_y| \ll |E|$ new_y contains all 1's if $\sum(y) < neg_sample_ratio$ new_y contains

`train_set_idx`

Indexed triples for the training.

`entity_idxes`

mapping.

`relation_idxes`

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 . collect_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_idx

mapping.

relation_idx

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.CVDDataModule (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 **:param-ref:~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):
```

(continues on next page)

(continued from previous page)

```
        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

Static evaluation functions for KGE models.

This module provides backward compatibility by re-exporting from the new `dicee.evaluation` module.

Deprecated since version Use: `dicee.evaluation` submodules instead. This module will be removed in a future version.

Functions

<code>evaluate_link_prediction_performance(→ Dict[str, float])</code>	Evaluate link prediction performance with head and tail prediction.
<code>evaluate_link_prediction_performance_with_.</code>	Evaluate link prediction with reciprocal relations.
<code>evaluate_link_prediction_performance_with_</code>	Evaluate link prediction with BPE encoding (head and tail).
<code>evaluate_link_prediction_performance_with_</code>	Evaluate link prediction with BPE encoding and reciprocals.
<code>evaluate_lp_bpe_k_vs_all(→ Dict[str, float])</code>	Evaluate BPE link prediction with KvsAll scoring.
<code>evaluate_literal_prediction(→ Optional[pandas.DataFrame])</code>	Op- Evaluate trained literal prediction model on a test file.
<code>evaluate_ensemble_link_prediction_performa</code>	Evaluate link prediction performance of an ensemble of KGE models.

Module Contents

`dicee.eval_static_funcs.evaluate_link_prediction_performance(model, triples, er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List]) → Dict[str, float]`

Evaluate link prediction performance with head and tail prediction.

Performs filtered evaluation where known correct answers are filtered out before computing ranks.

Parameters

- **model** – KGE model wrapper with entity/relation mappings.
- **triples** – Test triples as list of (head, relation, tail) strings.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.
- **re_vocab** – Mapping (relation, entity) -> list of valid head entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

`dicee.eval_static_funcs.evaluate_link_prediction_performance_with_reciprocals(model, triples, er_vocab: Dict[Tuple, List]) → Dict[str, float]`

Evaluate link prediction with reciprocal relations.

Optimized for models trained with reciprocal triples where only tail prediction is needed.

Parameters

- **model** – KGE model wrapper.
- **triples** – Test triples as list of (head, relation, tail) strings.

- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.eval_static_funcs.evaluate_link_prediction_performance_with_bpe(model,
    within_entities: List[str], triples: List[Tuple[str]], er_vocab: Dict[Tuple, List],
    re_vocab: Dict[Tuple, List]) → Dict[str, float]
```

Evaluate link prediction with BPE encoding (head and tail).

Parameters

- **model** – KGE model wrapper with BPE support.
- **within_entities** – List of entities to evaluate within.
- **triples** – Test triples as list of (head, relation, tail) tuples.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.
- **re_vocab** – Mapping (relation, entity) -> list of valid head entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.eval_static_funcs.evaluate_link_prediction_performance_with_bpe_reciprocals(
    model, within_entities: List[str], triples: List[List[str]], er_vocab: Dict[Tuple, List])
    → Dict[str, float]
```

Evaluate link prediction with BPE encoding and reciprocals.

Parameters

- **model** – KGE model wrapper with BPE support.
- **within_entities** – List of entities to evaluate within.
- **triples** – Test triples as list of [head, relation, tail] strings.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.eval_static_funcs.evaluate_lp_bpe_k_vs_all(model, triples: List[List[str]],
    er_vocab: Dict | None = None, batch_size: int | None = None,
    func_triple_to_bpe_representation: Callable | None = None,
    str_to_bpe_entity_to_idx: Dict | None = None) → Dict[str, float]
```

Evaluate BPE link prediction with KvsAll scoring.

Parameters

- **model** – The KGE model wrapper.
- **triples** – List of string triples.
- **er_vocab** – Entity-relation vocabulary for filtering.
- **batch_size** – Batch size for processing.
- **func_triple_to_bpe_representation** – Function to convert triples to BPE.
- **str_to_bpe_entity_to_idx** – Mapping from string entities to BPE indices.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

Raises

ValueError – If `batch_size` is not provided.

```
dicee.eval_static_funcs.evaluate_literal_prediction(kge_model, eval_file_path: str = None,
store_lit_preds: bool = True, eval_literals: bool = True, loader_backend: str = 'pandas',
return_attr_error_metrics: bool = False) → pandas.DataFrame | None
```

Evaluate trained literal prediction model on a test file.

Evaluates the literal prediction capabilities of a KGE model by computing MAE and RMSE metrics for each attribute.

Parameters

- **kge_model** – Trained KGE model with literal prediction capability.
- **eval_file_path** – Path to the evaluation file containing test literals.
- **store_lit_preds** – If True, stores predictions to CSV file.
- **eval_literals** – If True, evaluates and prints error metrics.
- **loader_backend** – Backend for loading dataset ('pandas' or 'rdflib').
- **return_attr_error_metrics** – If True, returns the metrics DataFrame.

Returns

DataFrame with per-attribute MAE and RMSE if `return_attr_error_metrics` is True, otherwise None.

Raises

- **RuntimeError** – If the KGE model doesn't have a trained literal model.
- **AssertionError** – If model is invalid or test set has no valid data.

Example

```
>>> from dicee import KGE
>>> from dicee.evaluation import evaluate_literal_prediction
>>> model = KGE(path="pretrained_model")
>>> metrics = evaluate_literal_prediction(
...     model,
...     eval_file_path="test_literals.csv",
...     return_attr_error_metrics=True
... )
>>> print(metrics)
```

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

Evaluate link prediction performance of an ensemble of KGE models.

Combines predictions from multiple models using weighted or simple averaging, with optional score normalization.

Parameters

- **models** – List of KGE models (e.g., snapshots from training).
- **triples** – Test triples as numpy array or list, shape (N, 3), with integer indices (head, relation, tail).
- **er_vocab** – Mapping (head_idx, rel_idx) -> list of tail indices for filtered evaluation.

- **weights** – Weights for model averaging. Required if `weighted_averaging` is `True`. Must sum to 1 for proper averaging.
- **batch_size** – Batch size for processing triples.
- **weighted_averaging** – If `True`, use weighted averaging of predictions. If `False`, use simple mean.
- **normalize_scores** – If `True`, normalize scores to `[0, 1]` range per sample before averaging.

Returns

Dictionary with `H@1`, `H@3`, `H@10`, and `MRR` metrics.

Raises

AssertionError – If `weighted_averaging` is `True` but weights are not provided or have wrong length.

Example

```
>>> from dicee.evaluation import evaluate_ensemble_link_prediction_performance
>>> models = [model1, model2, model3]
>>> weights = [0.5, 0.3, 0.2]
>>> results = evaluate_ensemble_link_prediction_performance(
...     models, test_triples, er_vocab,
...     weights=weights, weighted_averaging=True
... )
>>> print(f"MRR: {results['MRR']:.4f}")
```

`dicee.evaluation`

Evaluation module for knowledge graph embedding models.

This module provides comprehensive evaluation capabilities for KGE models, including link prediction, literal prediction, and ensemble evaluation.

Modules:

`link_prediction`: Functions for evaluating link prediction performance
`literal_prediction`: Functions for evaluating literal/attribute prediction
`ensemble`: Functions for ensemble model evaluation
`evaluator`: Main Evaluator class for integrated evaluation
`utils`: Shared utility functions for evaluation

Example

```
>>> from dicee.evaluation import Evaluator
>>> from dicee.evaluation.link_prediction import evaluate_link_prediction_performance
>>> from dicee.evaluation.ensemble import evaluate_ensemble_link_prediction_
↪ performance
```

Submodules

`dicee.evaluation.ensemble`

Ensemble evaluation functions.

This module provides functions for evaluating ensemble models, including weighted averaging and score normalization.

Functions

<code>evaluate_ensemble_link_prediction_performance</code>	Evaluate link prediction performance of an ensemble of KGE models.
--	--

Module Contents

`dicee.evaluation.ensemble.evaluate_ensemble_link_prediction_performance` (*models*: List, *triples*, *er_vocab*: Dict[Tuple, List], *weights*: List[float] | None = None, *batch_size*: int = 512, *weighted_averaging*: bool = True, *normalize_scores*: bool = True) → Dict[str, float]

Evaluate link prediction performance of an ensemble of KGE models.

Combines predictions from multiple models using weighted or simple averaging, with optional score normalization.

Parameters

- **models** – List of KGE models (e.g., snapshots from training).
- **triples** – Test triples as numpy array or list, shape (N, 3), with integer indices (head, relation, tail).
- **er_vocab** – Mapping (head_idx, rel_idx) -> list of tail indices for filtered evaluation.
- **weights** – Weights for model averaging. Required if `weighted_averaging` is True. Must sum to 1 for proper averaging.
- **batch_size** – Batch size for processing triples.
- **weighted_averaging** – If True, use weighted averaging of predictions. If False, use simple mean.
- **normalize_scores** – If True, normalize scores to [0, 1] range per sample before averaging.

Returns

Dictionary with H@1, H@3, H@10, and MRR metrics.

Raises

AssertionError – If `weighted_averaging` is True but `weights` are not provided or have wrong length.

Example

```
>>> from dicee.evaluation import evaluate_ensemble_link_prediction_performance
>>> models = [model1, model2, model3]
>>> weights = [0.5, 0.3, 0.2]
>>> results = evaluate_ensemble_link_prediction_performance(
...     models, test_triples, er_vocab,
...     weights=weights, weighted_averaging=True
... )
>>> print(f"MRR: {results['MRR']:.4f}")
```

`dicee.evaluation.evaluator`

Main Evaluator class for KGE model evaluation.

This module provides the Evaluator class which orchestrates evaluation of knowledge graph embedding models across different datasets and scoring techniques.

Attributes

<code>VALID_SCORING_TECHNIQUES</code>

Classes

<code>Evaluator</code>

Evaluator class for KGE models in various downstream tasks.

Module Contents

`dicee.evaluation.evaluator.VALID_SCORING_TECHNIQUES`

class `dicee.evaluation.evaluator.Evaluator` (*args*, *is_continual_training*: *bool* = *False*)

Evaluator class for KGE models in various downstream tasks.

Orchestrates link prediction evaluation with different scoring techniques including standard evaluation and byte-pair encoding based evaluation.

er_vocab

Entity-relation to tail vocabulary for filtered ranking.

re_vocab

Relation-entity (tail) to head vocabulary.

ee_vocab

Entity-entity to relation vocabulary.

num_entities

Total number of entities in the knowledge graph.

num_relations

Total number of relations in the knowledge graph.

args

Configuration arguments.

report

Dictionary storing evaluation results.

during_training

Whether evaluation is happening during training.

Example

```
>>> from dicee.evaluation import Evaluator
>>> evaluator = Evaluator(args)
>>> results = evaluator.eval(dataset, model, 'EntityPrediction')
>>> print(f"Test MRR: {results['Test']['MRR']:.4f}")
```

re_vocab: Dict | None = None

```

er_vocab: Dict | None = None

ee_vocab: Dict | None = None

func_triple_to_bpe_representation = None

is_continual_training = False

num_entities: int | None = None

num_relations: int | None = None

domain_constraints_per_rel = None

range_constraints_per_rel = None

args

report: Dict

during_training = False

vocab_preparation(dataset) → None
    Prepare vocabularies from the dataset for evaluation.
    Resolves any future objects and saves vocabularies to disk.

```

Parameters

dataset – Knowledge graph dataset with vocabulary attributes.

```

eval(dataset, trained_model, form_of_labelling: str, during_training: bool = False) → Dict | None
    Evaluate the trained model on the dataset.

```

Parameters

- **dataset** – Knowledge graph dataset (KG instance).
- **trained_model** – The trained KGE model.
- **form_of_labelling** – Type of labelling ('EntityPrediction' or 'RelationPrediction').
- **during_training** – Whether evaluation is during training.

Returns

Dictionary of evaluation metrics, or None if evaluation is skipped.

```

eval_rank_of_head_and_tail_entity(*, train_set, valid_set=None, test_set=None, trained_model)
    → None

```

Evaluate with negative sampling scoring.

```

eval_rank_of_head_and_tail_byte_pair_encoded_entity(*, train_set=None, valid_set=None,
    test_set=None, ordered_bpe_entities, trained_model) → None

```

Evaluate with BPE-encoded entities and negative sampling.

```

eval_with_byte(*, raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model,
    form_of_labelling) → None

```

Evaluate Byte model with generation.

```

eval_with_bpe_vs_all(*, raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model,
    form_of_labelling) → None

```

Evaluate with BPE and KvsAll scoring.

eval_with_vs_all (*, train_set, valid_set=None, test_set=None, trained_model, form_of_labelling)
→ None

Evaluate with KvsAll or 1vsAll scoring.

evaluate_lp_k_vs_all (model, triple_idx, info: str | None = None,
form_of_labelling: str | None = None) → Dict[str, float]

Filtered link prediction evaluation with KvsAll scoring.

Parameters

- **model** – The trained model to evaluate.
- **triple_idx** – Integer-indexed test triples.
- **info** – Description to print.
- **form_of_labelling** – ‘EntityPrediction’ or ‘RelationPrediction’.

Returns

Dictionary with H@1, H@3, H@10, and MRR metrics.

evaluate_lp_with_byte (model, triples: List[List[str]], info: str | None = None) → Dict[str, float]

Evaluate BytE model with text generation.

Parameters

- **model** – BytE model.
- **triples** – String triples.
- **info** – Description to print.

Returns

Dictionary with placeholder metrics (-1 values).

evaluate_lp_bpe_k_vs_all (model, triples: List[List[str]], info: str | None = None,
form_of_labelling: str | None = None) → Dict[str, float]

Evaluate BPE model with KvsAll scoring.

Parameters

- **model** – BPE-enabled model.
- **triples** – String triples.
- **info** – Description to print.
- **form_of_labelling** – Type of labelling.

Returns

Dictionary with H@1, H@3, H@10, and MRR metrics.

evaluate_lp (model, triple_idx, info: str) → Dict[str, float]

Evaluate link prediction with negative sampling.

Parameters

- **model** – The model to evaluate.
- **triple_idx** – Integer-indexed triples.
- **info** – Description to print.

Returns

Dictionary with H@1, H@3, H@10, and MRR metrics.

dummy_eval (*trained_model*, *form_of_labelling*: str) → None

Run evaluation from saved data (for continual training).

Parameters

- **trained_model** – The trained model.
- **form_of_labelling** – Type of labelling.

eval_with_data (*dataset*, *trained_model*, *triple_idx*: numpy.ndarray, *form_of_labelling*: str)
→ Dict[str, float]

Evaluate a trained model on a given dataset.

Parameters

- **dataset** – Knowledge graph dataset.
- **trained_model** – The trained model.
- **triple_idx** – Integer-indexed triples to evaluate.
- **form_of_labelling** – Type of labelling.

Returns

Dictionary with evaluation metrics.

Raises

ValueError – If scoring technique is invalid.

dicee.evaluation.link_prediction

Link prediction evaluation functions.

This module provides various functions for evaluating link prediction performance of knowledge graph embedding models.

Functions

<code>evaluate_link_prediction_performance(→ Dict[str, float])</code>	Evaluate link prediction performance with head and tail prediction.
<code>evaluate_link_prediction_performance_with_</code>	Evaluate link prediction with reciprocal relations.
<code>evaluate_link_prediction_performance_with_</code>	Evaluate link prediction with BPE encoding and reciprocals.
<code>evaluate_link_prediction_performance_with_</code>	Evaluate link prediction with BPE encoding (head and tail).
<code>evaluate_lp(→ Dict[str, float])</code>	Evaluate link prediction with batched processing.
<code>evaluate_bpe_lp(→ Dict[str, float])</code>	Evaluate link prediction with BPE-encoded entities.
<code>evaluate_lp_bpe_k_vs_all(→ Dict[str, float])</code>	Evaluate BPE link prediction with KvsAll scoring.

Module Contents

`dicee.evaluation.link_prediction.evaluate_link_prediction_performance` (*model*, *triples*, *er_vocab*: Dict[Tuple, List], *re_vocab*: Dict[Tuple, List]) → Dict[str, float]

Evaluate link prediction performance with head and tail prediction.

Performs filtered evaluation where known correct answers are filtered out before computing ranks.

Parameters

- **model** – KGE model wrapper with entity/relation mappings.

- **triples** – Test triples as list of (head, relation, tail) strings.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.
- **re_vocab** – Mapping (relation, entity) -> list of valid head entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.link_prediction.
```

```
evaluate_link_prediction_performance_with_reciprocals (model, triples,  
er_vocab: Dict[Tuple, List]) → Dict[str, float]
```

Evaluate link prediction with reciprocal relations.

Optimized for models trained with reciprocal triples where only tail prediction is needed.

Parameters

- **model** – KGE model wrapper.
- **triples** – Test triples as list of (head, relation, tail) strings.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.link_prediction.
```

```
evaluate_link_prediction_performance_with_bpe_reciprocals (model,  
within_entities: List[str], triples: List[List[str]], er_vocab: Dict[Tuple, List]) → Dict[str, float]
```

Evaluate link prediction with BPE encoding and reciprocals.

Parameters

- **model** – KGE model wrapper with BPE support.
- **within_entities** – List of entities to evaluate within.
- **triples** – Test triples as list of [head, relation, tail] strings.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.link_prediction.evaluate_link_prediction_performance_with_bpe (  
model, within_entities: List[str], triples: List[Tuple[str]], er_vocab: Dict[Tuple, List],  
re_vocab: Dict[Tuple, List]) → Dict[str, float]
```

Evaluate link prediction with BPE encoding (head and tail).

Parameters

- **model** – KGE model wrapper with BPE support.
- **within_entities** – List of entities to evaluate within.
- **triples** – Test triples as list of (head, relation, tail) tuples.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.
- **re_vocab** – Mapping (relation, entity) -> list of valid head entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.link_prediction.evaluate_lp(model, triple_idx, num_entities: int,
      er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List], info: str = 'Eval Starts',
      batch_size: int = 128, chunk_size: int = 1000) → Dict[str, float]
```

Evaluate link prediction with batched processing.

Memory-efficient evaluation using chunked entity scoring.

Parameters

- **model** – The KGE model to evaluate.
- **triple_idx** – Integer-indexed triples as numpy array.
- **num_entities** – Total number of entities.
- **er_vocab** – Mapping (head_idx, rel_idx) -> list of tail indices.
- **re_vocab** – Mapping (rel_idx, tail_idx) -> list of head indices.
- **info** – Description to print.
- **batch_size** – Batch size for triple processing.
- **chunk_size** – Chunk size for entity scoring.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.link_prediction.evaluate_bpe_lp(model, triple_idx: List[Tuple],
      all_bpe_shaped_entities, er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List],
      info: str = 'Eval Starts') → Dict[str, float]
```

Evaluate link prediction with BPE-encoded entities.

Parameters

- **model** – The KGE model to evaluate.
- **triple_idx** – List of BPE-encoded triple tuples.
- **all_bpe_shaped_entities** – All entities with BPE representations.
- **er_vocab** – Mapping for tail filtering.
- **re_vocab** – Mapping for head filtering.
- **info** – Description to print.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.link_prediction.evaluate_lp_bpe_k_vs_all(model, triples: List[List[str]],
      er_vocab: Dict | None = None, batch_size: int | None = None,
      func_triple_to_bpe_representation: Callable | None = None,
      str_to_bpe_entity_to_idx: Dict | None = None) → Dict[str, float]
```

Evaluate BPE link prediction with KvsAll scoring.

Parameters

- **model** – The KGE model wrapper.
- **triples** – List of string triples.
- **er_vocab** – Entity-relation vocabulary for filtering.
- **batch_size** – Batch size for processing.
- **func_triple_to_bpe_representation** – Function to convert triples to BPE.

- **str_to_bpe_entity_to_idx** – Mapping from string entities to BPE indices.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

Raises

ValueError – If batch_size is not provided.

dicee.evaluation.literal_prediction

Literal prediction evaluation functions.

This module provides functions for evaluating literal/attribute prediction performance of knowledge graph embedding models.

Functions

<code>evaluate_literal_prediction(→ tional[pandas.DataFrame])</code>	Op- Evaluate trained literal prediction model on a test file.
--	---

Module Contents

`dicee.evaluation.literal_prediction.evaluate_literal_prediction(kge_model,
eval_file_path: str = None, store_lit_preds: bool = True, eval_literals: bool = True,
loader_backend: str = 'pandas', return_attr_error_metrics: bool = False)
→ pandas.DataFrame | None`

Evaluate trained literal prediction model on a test file.

Evaluates the literal prediction capabilities of a KGE model by computing MAE and RMSE metrics for each attribute.

Parameters

- **kge_model** – Trained KGE model with literal prediction capability.
- **eval_file_path** – Path to the evaluation file containing test literals.
- **store_lit_preds** – If True, stores predictions to CSV file.
- **eval_literals** – If True, evaluates and prints error metrics.
- **loader_backend** – Backend for loading dataset ('pandas' or 'rdflib').
- **return_attr_error_metrics** – If True, returns the metrics DataFrame.

Returns

DataFrame with per-attribute MAE and RMSE if return_attr_error_metrics is True, otherwise None.

Raises

- **RuntimeError** – If the KGE model doesn't have a trained literal model.
- **AssertionError** – If model is invalid or test set has no valid data.

Example

```
>>> from dicee import KGE
>>> from dicee.evaluation import evaluate_literal_prediction
>>> model = KGE(path="pretrained_model")
>>> metrics = evaluate_literal_prediction(
...     model,
...     eval_file_path="test_literals.csv",
...     return_attr_error_metrics=True
... )
>>> print(metrics)
```

dicee.evaluation.utils

Utility functions for evaluation module.

This module contains shared helper functions used across different evaluation components.

Attributes

`DEFAULT_HITS_RANGE`

`ALL_HITS_RANGE`

Functions

<code>make_iterable_verbose(→ Iterable)</code>	Wrap an iterable with tqdm progress bar if verbose is True.
<code>compute_metrics_from_ranks(→ Dict[str, float])</code>	Compute standard link prediction metrics from ranks.
<code>compute_metrics_from_ranks_simple(→ Dict[str, float])</code>	Compute link prediction metrics without scaling factor.
<code>update_hits(→ None)</code>	Update hits dictionary based on rank.
<code>create_hits_dict(→ Dict[int, List[float]])</code>	Create an initialized hits dictionary.
<code>efficient_zero_grad(→ None)</code>	Efficiently zero gradients using <code>parameter.grad = None</code> .

Module Contents

`dicee.evaluation.utils.DEFAULT_HITS_RANGE: List[int] = [1, 3, 10]`

`dicee.evaluation.utils.ALL_HITS_RANGE: List[int]`

`dicee.evaluation.utils.make_iterable_verbose(iterable_object: Iterable, verbose: bool, desc: str = 'Default', position: int | None = None, leave: bool = True) → Iterable`

Wrap an iterable with tqdm progress bar if verbose is True.

Parameters

- **iterable_object** – The iterable to potentially wrap.
- **verbose** – Whether to show progress bar.
- **desc** – Description for the progress bar.

- **position** – Position of the progress bar.
- **leave** – Whether to leave the progress bar after completion.

Returns

The original iterable or a tqdm-wrapped version.

`dicee.evaluation.utils.compute_metrics_from_ranks (ranks: List[int], num_triples: int, hits_dict: Dict[int, List[float]], scale_factor: int = 1) → Dict[str, float]`

Compute standard link prediction metrics from ranks.

Parameters

- **ranks** – List of ranks for each prediction.
- **num_triples** – Total number of triples evaluated.
- **hits_dict** – Dictionary mapping hit levels to lists of hits.
- **scale_factor** – Factor to scale the denominator (e.g., 2 for head+tail).

Returns

Dictionary containing H@1, H@3, **H@10**, and MRR metrics.

`dicee.evaluation.utils.compute_metrics_from_ranks_simple (ranks: List[int], num_triples: int, hits_dict: Dict[int, List[float]]) → Dict[str, float]`

Compute link prediction metrics without scaling factor.

Parameters

- **ranks** – List of ranks for each prediction.
- **num_triples** – Total number of triples evaluated.
- **hits_dict** – Dictionary mapping hit levels to lists of hits.

Returns

Dictionary containing H@1, H@3, **H@10**, and MRR metrics.

`dicee.evaluation.utils.update_hits (hits: Dict[int, List[float]], rank: int, hits_range: List[int] | None = None) → None`

Update hits dictionary based on rank.

Parameters

- **hits** – Dictionary to update in-place.
- **rank** – The rank to check against hit levels.
- **hits_range** – List of hit levels to check (default: ALL_HITS_RANGE).

`dicee.evaluation.utils.create_hits_dict (hits_range: List[int] | None = None) → Dict[int, List[float]]`

Create an initialized hits dictionary.

Parameters

hits_range – List of hit levels to initialize (default: ALL_HITS_RANGE).

Returns

Dictionary with empty lists for each hit level.

`dicee.evaluation.utils.efficient_zero_grad (model) → None`

Efficiently zero gradients using `parameter.grad = None`.

This is more efficient than `optimizer.zero_grad()` as it avoids memory operations.

See: https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Parameters

model – PyTorch model to zero gradients for.

Attributes

DEFAULT_HITS_RANGE

ALL_HITS_RANGE

Classes

Evaluator

Evaluator class for KGE models in various downstream tasks.

Functions

<code>evaluate_link_prediction_performance(→ Dict[str, float])</code>	Evaluate link prediction performance with head and tail prediction.
<code>evaluate_link_prediction_performance_with_reciprocal_relations(→ Dict[str, float])</code>	Evaluate link prediction with reciprocal relations.
<code>evaluate_link_prediction_performance_with_bpe_encoding(head_and_tail=True)(→ Dict[str, float])</code>	Evaluate link prediction with BPE encoding (head and tail).
<code>evaluate_link_prediction_performance_with_bpe_encoding_and_reciprocals(→ Dict[str, float])</code>	Evaluate link prediction with BPE encoding and reciprocals.
<code>evaluate_lp(→ Dict[str, float])</code>	Evaluate link prediction with batched processing.
<code>evaluate_lp_bpe_k_vs_all(→ Dict[str, float])</code>	Evaluate BPE link prediction with KvsAll scoring.
<code>evaluate_bpe_lp(→ Dict[str, float])</code>	Evaluate link prediction with BPE-encoded entities.
<code>evaluate_literal_prediction(→ Optional[pandas.DataFrame])</code>	Op- Evaluate trained literal prediction model on a test file.
<code>evaluate_ensemble_link_prediction_performance(→ Dict[str, float])</code>	Evaluate link prediction performance of an ensemble of KGE models.
<code>compute_metrics_from_ranks(→ Dict[str, float])</code>	Compute standard link prediction metrics from ranks.
<code>compute_metrics_from_ranks_simple(→ Dict[str, float])</code>	Compute link prediction metrics without scaling factor.
<code>make_iterable_verbose(→ Iterable)</code>	Wrap an iterable with tqdm progress bar if verbose is True.
<code>update_hits(→ None)</code>	Update hits dictionary based on rank.
<code>create_hits_dict(→ Dict[int, List[float]])</code>	Create an initialized hits dictionary.

Package Contents

class `dicee.evaluation.Evaluator` (*args, is_continual_training: bool = False*)

Evaluator class for KGE models in various downstream tasks.

Orchestrates link prediction evaluation with different scoring techniques including standard evaluation and byte-pair encoding based evaluation.

er_vocab
Entity-relation to tail vocabulary for filtered ranking.

re_vocab
Relation-entity (tail) to head vocabulary.

ee_vocab
Entity-entity to relation vocabulary.

num_entities
Total number of entities in the knowledge graph.

num_relations
Total number of relations in the knowledge graph.

args
Configuration arguments.

report
Dictionary storing evaluation results.

during_training
Whether evaluation is happening during training.

Example

```
>>> from dicee.evaluation import Evaluator
>>> evaluator = Evaluator(args)
>>> results = evaluator.eval(dataset, model, 'EntityPrediction')
>>> print(f"Test MRR: {results['Test']['MRR']:.4f}")
```

```
re_vocab: Dict | None = None
er_vocab: Dict | None = None
ee_vocab: Dict | None = None
func_triple_to_bpe_representation = None
is_continual_training = False
num_entities: int | None = None
num_relations: int | None = None
domain_constraints_per_rel = None
range_constraints_per_rel = None
args
report: Dict
during_training = False
```

vocab_preparation (*dataset*) → None

Prepare vocabularies from the dataset for evaluation.

Resolves any future objects and saves vocabularies to disk.

Parameters

dataset – Knowledge graph dataset with vocabulary attributes.

eval (*dataset, trained_model, form_of_labelling: str, during_training: bool = False*) → Dict | None

Evaluate the trained model on the dataset.

Parameters

- **dataset** – Knowledge graph dataset (KG instance).
- **trained_model** – The trained KGE model.
- **form_of_labelling** – Type of labelling ('EntityPrediction' or 'RelationPrediction').
- **during_training** – Whether evaluation is during training.

Returns

Dictionary of evaluation metrics, or None if evaluation is skipped.

eval_rank_of_head_and_tail_entity (*, *train_set, valid_set=None, test_set=None, trained_model*)
→ None

Evaluate with negative sampling scoring.

eval_rank_of_head_and_tail_byte_pair_encoded_entity (*, *train_set=None, valid_set=None, test_set=None, ordered_bpe_entities, trained_model*) → None

Evaluate with BPE-encoded entities and negative sampling.

eval_with_byte (*, *raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model, form_of_labelling*) → None

Evaluate Byte model with generation.

eval_with_bpe_vs_all (*, *raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model, form_of_labelling*) → None

Evaluate with BPE and KvsAll scoring.

eval_with_vs_all (*, *train_set, valid_set=None, test_set=None, trained_model, form_of_labelling*)
→ None

Evaluate with KvsAll or 1vsAll scoring.

evaluate_lp_k_vs_all (*model, triple_idx, info: str | None = None, form_of_labelling: str | None = None*) → Dict[str, float]

Filtered link prediction evaluation with KvsAll scoring.

Parameters

- **model** – The trained model to evaluate.
- **triple_idx** – Integer-indexed test triples.
- **info** – Description to print.
- **form_of_labelling** – 'EntityPrediction' or 'RelationPrediction'.

Returns

Dictionary with H@1, H@3, H@10, and MRR metrics.

evaluate_lp_with_byte (*model*, *triples*: List[List[str]], *info*: str | None = None) → Dict[str, float]

Evaluate ByteE model with text generation.

Parameters

- **model** – ByteE model.
- **triples** – String triples.
- **info** – Description to print.

Returns

Dictionary with placeholder metrics (-1 values).

evaluate_lp_bpe_k_vs_all (*model*, *triples*: List[List[str]], *info*: str | None = None, *form_of_labelling*: str | None = None) → Dict[str, float]

Evaluate BPE model with KvsAll scoring.

Parameters

- **model** – BPE-enabled model.
- **triples** – String triples.
- **info** – Description to print.
- **form_of_labelling** – Type of labelling.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

evaluate_lp (*model*, *triple_idx*, *info*: str) → Dict[str, float]

Evaluate link prediction with negative sampling.

Parameters

- **model** – The model to evaluate.
- **triple_idx** – Integer-indexed triples.
- **info** – Description to print.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

dummy_eval (*trained_model*, *form_of_labelling*: str) → None

Run evaluation from saved data (for continual training).

Parameters

- **trained_model** – The trained model.
- **form_of_labelling** – Type of labelling.

eval_with_data (*dataset*, *trained_model*, *triple_idx*: numpy.ndarray, *form_of_labelling*: str) → Dict[str, float]

Evaluate a trained model on a given dataset.

Parameters

- **dataset** – Knowledge graph dataset.
- **trained_model** – The trained model.
- **triple_idx** – Integer-indexed triples to evaluate.
- **form_of_labelling** – Type of labelling.

Returns

Dictionary with evaluation metrics.

Raises

ValueError – If scoring technique is invalid.

```
dicee.evaluation.evaluate_link_prediction_performance(model, triples,  
er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List]) → Dict[str, float]
```

Evaluate link prediction performance with head and tail prediction.

Performs filtered evaluation where known correct answers are filtered out before computing ranks.

Parameters

- **model** – KGE model wrapper with entity/relation mappings.
- **triples** – Test triples as list of (head, relation, tail) strings.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.
- **re_vocab** – Mapping (relation, entity) -> list of valid head entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.evaluate_link_prediction_performance_with_reciprocals(model, triples,  
er_vocab: Dict[Tuple, List]) → Dict[str, float]
```

Evaluate link prediction with reciprocal relations.

Optimized for models trained with reciprocal triples where only tail prediction is needed.

Parameters

- **model** – KGE model wrapper.
- **triples** – Test triples as list of (head, relation, tail) strings.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.evaluate_link_prediction_performance_with_bpe(model,  
within_entities: List[str], triples: List[List[str]], er_vocab: Dict[Tuple, List],  
re_vocab: Dict[Tuple, List]) → Dict[str, float]
```

Evaluate link prediction with BPE encoding (head and tail).

Parameters

- **model** – KGE model wrapper with BPE support.
- **within_entities** – List of entities to evaluate within.
- **triples** – Test triples as list of (head, relation, tail) tuples.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.
- **re_vocab** – Mapping (relation, entity) -> list of valid head entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.evaluate_link_prediction_performance_with_bpe_reciprocals(model,  
within_entities: List[str], triples: List[List[str]], er_vocab: Dict[Tuple, List]) → Dict[str, float]
```

Evaluate link prediction with BPE encoding and reciprocals.

Parameters

- **model** – KGE model wrapper with BPE support.
- **within_entities** – List of entities to evaluate within.
- **triples** – Test triples as list of [head, relation, tail] strings.
- **er_vocab** – Mapping (entity, relation) -> list of valid tail entities.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.evaluate_lp(model, triple_idx, num_entities: int, er_vocab: Dict[Tuple, List],  
                             re_vocab: Dict[Tuple, List], info: str = 'Eval Starts', batch_size: int = 128, chunk_size: int = 1000)  
→ Dict[str, float]
```

Evaluate link prediction with batched processing.

Memory-efficient evaluation using chunked entity scoring.

Parameters

- **model** – The KGE model to evaluate.
- **triple_idx** – Integer-indexed triples as numpy array.
- **num_entities** – Total number of entities.
- **er_vocab** – Mapping (head_idx, rel_idx) -> list of tail indices.
- **re_vocab** – Mapping (rel_idx, tail_idx) -> list of head indices.
- **info** – Description to print.
- **batch_size** – Batch size for triple processing.
- **chunk_size** – Chunk size for entity scoring.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

```
dicee.evaluation.evaluate_lp_bpe_k_vs_all(model, triples: List[List[str]],  
                                           er_vocab: Dict | None = None, batch_size: int | None = None,  
                                           func_triple_to_bpe_representation: Callable | None = None,  
                                           str_to_bpe_entity_to_idx: Dict | None = None) → Dict[str, float]
```

Evaluate BPE link prediction with KvsAll scoring.

Parameters

- **model** – The KGE model wrapper.
- **triples** – List of string triples.
- **er_vocab** – Entity-relation vocabulary for filtering.
- **batch_size** – Batch size for processing.
- **func_triple_to_bpe_representation** – Function to convert triples to BPE.
- **str_to_bpe_entity_to_idx** – Mapping from string entities to BPE indices.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

Raises

ValueError – If batch_size is not provided.

`dicee.evaluation.evaluate_bpe_lp(model, triple_idx: List[Tuple], all_bpe_shaped_entities, er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List], info: str = 'Eval Starts') → Dict[str, float]`

Evaluate link prediction with BPE-encoded entities.

Parameters

- **model** – The KGE model to evaluate.
- **triple_idx** – List of BPE-encoded triple tuples.
- **all_bpe_shaped_entities** – All entities with BPE representations.
- **er_vocab** – Mapping for tail filtering.
- **re_vocab** – Mapping for head filtering.
- **info** – Description to print.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

`dicee.evaluation.evaluate_literal_prediction(kge_model, eval_file_path: str = None, store_lit_preds: bool = True, eval_literals: bool = True, loader_backend: str = 'pandas', return_attr_error_metrics: bool = False) → pandas.DataFrame | None`

Evaluate trained literal prediction model on a test file.

Evaluates the literal prediction capabilities of a KGE model by computing MAE and RMSE metrics for each attribute.

Parameters

- **kge_model** – Trained KGE model with literal prediction capability.
- **eval_file_path** – Path to the evaluation file containing test literals.
- **store_lit_preds** – If True, stores predictions to CSV file.
- **eval_literals** – If True, evaluates and prints error metrics.
- **loader_backend** – Backend for loading dataset ('pandas' or 'rdflib').
- **return_attr_error_metrics** – If True, returns the metrics DataFrame.

Returns

DataFrame with per-attribute MAE and RMSE if `return_attr_error_metrics` is True, otherwise None.

Raises

- **RuntimeError** – If the KGE model doesn't have a trained literal model.
- **AssertionError** – If model is invalid or test set has no valid data.

Example

```
>>> from dicee import KGE
>>> from dicee.evaluation import evaluate_literal_prediction
>>> model = KGE(path="pretrained_model")
>>> metrics = evaluate_literal_prediction(
...     model,
...     eval_file_path="test_literals.csv",
...     return_attr_error_metrics=True
... )
>>> print(metrics)
```

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

Evaluate link prediction performance of an ensemble of KGE models.

Combines predictions from multiple models using weighted or simple averaging, with optional score normalization.

Parameters

- **models** – List of KGE models (e.g., snapshots from training).
- **triples** – Test triples as numpy array or list, shape (N, 3), with integer indices (head, relation, tail).
- **er_vocab** – Mapping (head_idx, rel_idx) -> list of tail indices for filtered evaluation.
- **weights** – Weights for model averaging. Required if `weighted_averaging` is `True`. Must sum to 1 for proper averaging.
- **batch_size** – Batch size for processing triples.
- **weighted_averaging** – If `True`, use weighted averaging of predictions. If `False`, use simple mean.
- **normalize_scores** – If `True`, normalize scores to [0, 1] range per sample before averaging.

Returns

Dictionary with H@1, H@3, H@10, and MRR metrics.

Raises

AssertionError – If `weighted_averaging` is `True` but weights are not provided or have wrong length.

Example

```
>>> from dicee.evaluation import evaluate_ensemble_link_prediction_performance
>>> models = [model1, model2, model3]
>>> weights = [0.5, 0.3, 0.2]
>>> results = evaluate_ensemble_link_prediction_performance(
...     models, test_triples, er_vocab,
...     weights=weights, weighted_averaging=True
... )
>>> print(f"MRR: {results['MRR']:.4f}")
```

`dicee.evaluation.compute_metrics_from_ranks` (*ranks: List[int], num_triples: int, hits_dict: Dict[int, List[float]], scale_factor: int = 1*) → Dict[str, float]

Compute standard link prediction metrics from ranks.

Parameters

- **ranks** – List of ranks for each prediction.
- **num_triples** – Total number of triples evaluated.
- **hits_dict** – Dictionary mapping hit levels to lists of hits.
- **scale_factor** – Factor to scale the denominator (e.g., 2 for head+tail).

Returns

Dictionary containing H@1, H@3, H@10, and MRR metrics.

`dicee.evaluation.compute_metrics_from_ranks_simple` (*ranks*: List[int], *num_triples*: int, *hits_dict*: Dict[int, List[float]]) → Dict[str, float]

Compute link prediction metrics without scaling factor.

Parameters

- **ranks** – List of ranks for each prediction.
- **num_triples** – Total number of triples evaluated.
- **hits_dict** – Dictionary mapping hit levels to lists of hits.

Returns

Dictionary containing H@1, H@3, **H@10**, and MRR metrics.

`dicee.evaluation.make_iterable_verbose` (*iterable_object*: Iterable, *verbose*: bool, *desc*: str = 'Default', *position*: int | None = None, *leave*: bool = True) → Iterable

Wrap an iterable with tqdm progress bar if verbose is True.

Parameters

- **iterable_object** – The iterable to potentially wrap.
- **verbose** – Whether to show progress bar.
- **desc** – Description for the progress bar.
- **position** – Position of the progress bar.
- **leave** – Whether to leave the progress bar after completion.

Returns

The original iterable or a tqdm-wrapped version.

`dicee.evaluation.update_hits` (*hits*: Dict[int, List[float]], *rank*: int, *hits_range*: List[int] | None = None) → None

Update hits dictionary based on rank.

Parameters

- **hits** – Dictionary to update in-place.
- **rank** – The rank to check against hit levels.
- **hits_range** – List of hit levels to check (default: ALL_HITS_RANGE).

`dicee.evaluation.create_hits_dict` (*hits_range*: List[int] | None = None) → Dict[int, List[float]]

Create an initialized hits dictionary.

Parameters

hits_range – List of hit levels to initialize (default: ALL_HITS_RANGE).

Returns

Dictionary with empty lists for each hit level.

`dicee.evaluation.DEFAULT_HITS_RANGE`: List[int] = [1, 3, 10]

`dicee.evaluation.ALL_HITS_RANGE`: List[int]

dicee.evaluator

Evaluator module for knowledge graph embedding models.

This module provides backward compatibility by re-exporting from the new `dicee.evaluation` module.

Deprecated since version Use: `dicee.evaluation.Evaluator` instead. This module will be removed in a future version.

Classes

<i>Evaluator</i>	Evaluator class for KGE models in various downstream tasks.
------------------	---

Module Contents

class `dicee.evaluator.Evaluator` (*args*, *is_continual_training*: *bool* = *False*)

Evaluator class for KGE models in various downstream tasks.

Orchestrates link prediction evaluation with different scoring techniques including standard evaluation and byte-pair encoding based evaluation.

er_vocab

Entity-relation to tail vocabulary for filtered ranking.

re_vocab

Relation-entity (tail) to head vocabulary.

ee_vocab

Entity-entity to relation vocabulary.

num_entities

Total number of entities in the knowledge graph.

num_relations

Total number of relations in the knowledge graph.

args

Configuration arguments.

report

Dictionary storing evaluation results.

during_training

Whether evaluation is happening during training.

Example

```
>>> from dicee.evaluation import Evaluator
>>> evaluator = Evaluator(args)
>>> results = evaluator.eval(dataset, model, 'EntityPrediction')
>>> print(f"Test MRR: {results['Test']['MRR']:.4f}")
```

re_vocab: Dict | None = None

er_vocab: Dict | None = None

ee_vocab: Dict | None = None

func_triple_to_bpe_representation = None

```

is_continual_training = False

num_entities: int | None = None

num_relations: int | None = None

domain_constraints_per_rel = None

range_constraints_per_rel = None

args

report: Dict

during_training = False

vocab_preparation(dataset) → None
    Prepare vocabularies from the dataset for evaluation.
    Resolves any future objects and saves vocabularies to disk.

```

Parameters

dataset – Knowledge graph dataset with vocabulary attributes.

```

eval(dataset, trained_model, form_of_labelling: str, during_training: bool = False) → Dict | None
    Evaluate the trained model on the dataset.

```

Parameters

- **dataset** – Knowledge graph dataset (KG instance).
- **trained_model** – The trained KGE model.
- **form_of_labelling** – Type of labelling ('EntityPrediction' or 'RelationPrediction').
- **during_training** – Whether evaluation is during training.

Returns

Dictionary of evaluation metrics, or None if evaluation is skipped.

```

eval_rank_of_head_and_tail_entity(*, train_set, valid_set=None, test_set=None, trained_model)
    → None

```

Evaluate with negative sampling scoring.

```

eval_rank_of_head_and_tail_byte_pair_encoded_entity(*, train_set=None, valid_set=None,
    test_set=None, ordered_bpe_entities, trained_model) → None

```

Evaluate with BPE-encoded entities and negative sampling.

```

eval_with_byte(*, raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model,
    form_of_labelling) → None

```

Evaluate Byte model with generation.

```

eval_with_bpe_vs_all(*, raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model,
    form_of_labelling) → None

```

Evaluate with BPE and KvsAll scoring.

```

eval_with_vs_all(*, train_set, valid_set=None, test_set=None, trained_model, form_of_labelling)
    → None

```

Evaluate with KvsAll or 1vsAll scoring.

evaluate_lp_k_vs_all (*model*, *triple_idx*, *info*: *str* | *None* = *None*,
 form_of_labelling: *str* | *None* = *None*) → Dict[str, float]

Filtered link prediction evaluation with KvsAll scoring.

Parameters

- **model** – The trained model to evaluate.
- **triple_idx** – Integer-indexed test triples.
- **info** – Description to print.
- **form_of_labelling** – ‘EntityPrediction’ or ‘RelationPrediction’.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

evaluate_lp_with_byte (*model*, *triples*: List[List[str]], *info*: *str* | *None* = *None*) → Dict[str, float]

Evaluate BytE model with text generation.

Parameters

- **model** – BytE model.
- **triples** – String triples.
- **info** – Description to print.

Returns

Dictionary with placeholder metrics (-1 values).

evaluate_lp_bpe_k_vs_all (*model*, *triples*: List[List[str]], *info*: *str* | *None* = *None*,
 form_of_labelling: *str* | *None* = *None*) → Dict[str, float]

Evaluate BPE model with KvsAll scoring.

Parameters

- **model** – BPE-enabled model.
- **triples** – String triples.
- **info** – Description to print.
- **form_of_labelling** – Type of labelling.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

evaluate_lp (*model*, *triple_idx*, *info*: *str*) → Dict[str, float]

Evaluate link prediction with negative sampling.

Parameters

- **model** – The model to evaluate.
- **triple_idx** – Integer-indexed triples.
- **info** – Description to print.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

dummy_eval (*trained_model*, *form_of_labelling*: *str*) → None

Run evaluation from saved data (for continual training).

Parameters

- **trained_model** – The trained model.
- **form_of_labelling** – Type of labelling.

eval_with_data (*dataset, trained_model, triple_idx: numpy.ndarray, form_of_labelling: str*)
→ Dict[str, float]

Evaluate a trained model on a given dataset.

Parameters

- **dataset** – Knowledge graph dataset.
- **trained_model** – The trained model.
- **triple_idx** – Integer-indexed triples to evaluate.
- **form_of_labelling** – Type of labelling.

Returns

Dictionary with evaluation metrics.

Raises

ValueError – If scoring technique is invalid.

dicee.executer

Executor module for training, retraining and evaluating KGE models.

This module provides the Execute and ContinuousExecute classes for managing the full lifecycle of knowledge graph embedding model training.

Classes

<i>Execute</i>	Executor class for training, retraining and evaluating KGE models.
<i>ContinuousExecute</i>	A subclass of Execute Class for retraining

Module Contents

class `dicee.executer.Execute` (*args, continuous_training: bool = False*)

Executor class for training, retraining and evaluating KGE models.

Handles the complete workflow: 1. Loading & Preprocessing & Serializing input data 2. Training & Validation & Testing 3. Storing all necessary information

args

Processed input arguments.

distributed

Whether distributed training is enabled.

rank

Process rank in distributed training.

world_size

Total number of processes.

local_rank

Local GPU rank.

trainer
Training handler instance.

trained_model
The trained model after training completes.

knowledge_graph
The loaded knowledge graph.

report
Dictionary storing training metrics and results.

evaluator
Model evaluation handler.

distributed

args

is_continual_training = False

trainer: *dicce.trainer.DICE_Trainer* | None = None

trained_model = None

knowledge_graph: *dicce.knowledge_graph.KG* | None = None

report: Dict

evaluator: *dicce.evaluator.Evaluator* | None = None

start_time: float | None = None

is_rank_zero() → bool

cleanup()

setup_executor() → None
Set up storage directories for the experiment.
Creates or reuses experiment directories based on configuration. Saves the configuration to a JSON file.

create_and_store_kg() → None
Create knowledge graph and store as memory-mapped file.
Only executed on rank 0 in distributed training. Skips if memmap already exists.

load_from_memmap() → None
Load knowledge graph from memory-mapped file.

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

Knowledge Graph module for data loading and preprocessing.

Provides the KG class for handling knowledge graph data including loading, preprocessing, and indexing operations.

Classes

<i>KG</i>	Knowledge Graph container and processor.
-----------	--

Module Contents

```
class dicee.knowledge_graph.KG (dataset_dir: str | None = None, byte_pair_encoding: bool = False,  
padding: bool = False, add_noise_rate: float | None = None, sparql_endpoint: str | None = None,  
path_single_kg: str | None = None, path_for_deserialization: str | None = None,  
add_reciprocal: bool | None = None, eval_model: str | None = None,  
read_only_few: int | None = None, sample_triples_ratio: float | None = None,  
path_for_serialization: str | None = None, entity_to_idx: Dict | None = None,  
relation_to_idx: Dict | None = None, backend: str | None = None,  
training_technique: str | None = None, separator: str | None = None)
```

Knowledge Graph container and processor.

Handles loading, preprocessing, and indexing of knowledge graph data from various sources including files, SPARQL endpoints, and serialized formats.

dataset_dir

Path to directory containing train/valid/test files.

num_entities

Total number of unique entities.

num_relations

Total number of unique relations.

train_set

Indexed training triples as numpy array.

valid_set

Indexed validation triples (optional).

test_set

Indexed test triples (optional).

entity_to_idx

Mapping from entity strings to indices.

relation_to_idx

Mapping from relation strings to indices.

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: int | None = None
num_relations: int | None = 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
separator = None
raw_train_set = None
raw_valid_set = None
raw_test_set = None
train_set = None
valid_set = None
test_set = None
idx_entity_to_bpe_shaped: Dict
enc
num_tokens
num_bpe_entities: int | None = None
padding = False
dummy_id
max_length_subword_tokens: int | None = None

```

```

train_set_target = None

target_dim: int | None = None

train_target_indices = None

ordered_bpe_entities = None

description_of_input = None

describe() → None
    Generate a description string of the dataset statistics.

property entities_str: List[str]
    Get list of all entity strings.

property relations_str: List[str]
    Get list of all relation strings.

exists(h: str, r: str, t: str) → bool
    Check if a triple exists in the training set.

Parameters
    • h – Head entity string.
    • r – Relation string.
    • t – Tail entity string.

Returns
    True if the triple exists, False otherwise.

__iter__() → Iterator[Tuple[str, str, str]]
    Iterate over training triples as string tuples.

__len__() → int
    Return number of triples in the raw training set.

func_triple_to_bpe_representation(triple: List[str])

```

dicee.knowledge_graph_embeddings

Classes

<i>KGE</i>	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.

$\text{argmax}_{\{e \in E\}} f(e, r, t)$, 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.

$\text{argmax}_{\{r \in R\}} f(h, r, t)$, 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

$\text{argmax}_{\{e \in E\}} f(h, r, e)$, 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

$$\{(e,r,x) \mid f(e,r,x) > \text{confidence_land}(e,r,x)\}$$

otin G

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

ADOPT Optimizer Implementation.

This module implements the ADOPT (Adaptive Optimization with Precise Tracking) algorithm, an advanced optimization method for training neural networks.

ADOPT Overview:

ADOPT is an adaptive learning rate optimization algorithm that combines the benefits of momentum-based methods with per-parameter learning rate adaptation. Unlike Adam, which applies momentum to raw gradients, ADOPT normalizes gradients first and then applies momentum, leading to more stable training dynamics.

Key Features: - Gradient normalization before momentum application - Adaptive per-parameter learning rates - Optional gradient clipping that grows with training steps - Support for decoupled weight decay (AdamW-style) - Multiple execution modes: single-tensor, multi-tensor (foreach), and fused (planned)

Algorithm Comparison:

Adam: $m = \beta_1 * m + (1 - \beta_1) * g$, $\theta = \theta - \alpha * m / \sqrt{v}$ ADOPT: $m = \beta_1 * m + (1 - \beta_1) * g / \sqrt{v}$, $\theta = \theta - \alpha * m$

The key difference is that ADOPT normalizes gradients before momentum, which provides better stability and can lead to improved convergence.

Classes:

- ADOPT: Main optimizer class (extends torch.optim.Optimizer)

Functions:

- `adopt`: Functional API for ADOPT algorithm computation
- `_single_tensor_adopt`: Single-tensor implementation (TorchScript compatible)
- `_multi_tensor_adopt`: Multi-tensor implementation using foreach operations

Performance:

- Single-tensor: Default, compatible with `torch.jit.script`
- Multi-tensor (foreach): 2-3x faster on GPU through vectorization
- Fused (planned): Would provide maximum performance via specialized kernels

Example:

```
>>> import torch
>>> from dicee.models.adopt import ADOPT
>>>
>>> model = torch.nn.Linear(10, 1)
>>> optimizer = ADOPT(model.parameters(), lr=0.001, weight_decay=0.01, decouple=True)
>>>
>>> # Training loop
>>> for epoch in range(num_epochs):
...     optimizer.zero_grad()
...     output = model(input)
...     loss = criterion(output, target)
...     loss.backward()
...     optimizer.step()
```

References:

Original implementation: <https://github.com/iShohei220/adopt>

Notes:

This implementation is based on the original ADOPT implementation and adapted to work with the PyTorch optimizer interface and the dicee framework.

Classes

<i>ADOPT</i>	ADOPT Optimizer.
--------------	------------------

Functions

<i>adopt</i> (params, grads, exp_avgs, exp_avg_sqs, state_steps)	Functional API that performs ADOPT algorithm computation.
--	---

Module Contents

```
class dicee.models.adapt.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

ADOPT Optimizer.

ADOPT is an adaptive learning rate optimization algorithm that combines momentum-based updates with adaptive per-parameter learning rates. It uses exponential moving averages of gradients and squared gradients, with gradient clipping for stability.

The algorithm performs the following key operations: 1. Normalizes gradients by the square root of the second moment estimate 2. Applies optional gradient clipping based on the training step 3. Updates parameters using momentum-smoothed normalized gradients 4. Supports decoupled weight decay (AdamW-style) or L2 regularization

Mathematical formulation:

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \text{clip}(g_t / \sqrt{v_t}) \quad v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2 \quad \theta_t = \theta_{t-1} - \alpha * m_t$$

where:

- θ_t : parameter at step t
- g_t : gradient at step t
- m_t : first moment estimate (momentum)
- v_t : second moment estimate (variance)
- α : learning rate
- β_1, β_2 : exponential decay rates
- $\text{clip}()$: optional gradient clipping function

Reference:

Original implementation: <https://github.com/iShohei220/adopt>

Parameters

- **params** (*ParamsT*) – Iterable of parameters to optimize or dicts defining parameter groups.
- **lr** (*float or Tensor, optional*) – Learning rate. Can be a float or 1-element Tensor. Default: 1e-3
- **betas** (*Tuple[float, float], optional*) – Coefficients (β_1, β_2) for computing running averages of gradient and its square. β_1 controls momentum, β_2 controls variance. Default: (0.9, 0.9999)
- **eps** (*float, optional*) – Term added to denominator to improve numerical stability. Default: 1e-6
- **clip_lambda** (*Callable[[int], float], optional*) – Function that takes the step number and returns the gradient clipping threshold. Common choices: - lambda step: step**0.25 (default, gradually increases clipping threshold) - lambda step: 1.0 (constant clipping) - None (no clipping) Default: lambda step: step**0.25
- **weight_decay** (*float, optional*) – Weight decay coefficient (L2 penalty). Default: 0.0

- **decouple** (*bool, optional*) – If True, uses decoupled weight decay (AdamW-style), applying weight decay directly to parameters. If False, adds weight decay to gradients (L2 regularization). Default: False
- **foreach** (*bool, optional*) – If True, uses the faster foreach implementation for multi-tensor operations. Default: None (auto-select)
- **maximize** (*bool, optional*) – If True, maximizes parameters instead of minimizing. Useful for reinforcement learning. Default: False
- **capturable** (*bool, optional*) – If True, the optimizer is safe to capture in a CUDA graph. Requires learning rate as Tensor. Default: False
- **differentiable** (*bool, optional*) – If True, the optimization step can be differentiated. Useful for meta-learning. Default: False
- **fused** (*bool, optional*) – If True, uses fused kernel implementation (currently not supported). Default: None

Raises

- **ValueError** – If learning rate, epsilon, betas, or weight_decay are invalid.
- **RuntimeError** – If fused is enabled (not currently supported).
- **RuntimeError** – If lr is a Tensor with foreach=True and capturable=False.

Example

```
>>> # Basic usage
>>> optimizer = ADOPT(model.parameters(), lr=0.001)
>>> optimizer.zero_grad()
>>> loss.backward()
>>> optimizer.step()
```

```
>>> # With decoupled weight decay
>>> optimizer = ADOPT(model.parameters(), lr=0.001, weight_decay=0.01,
↳decouple=True)
```

```
>>> # Custom gradient clipping
>>> optimizer = ADOPT(model.parameters(), clip_lambda=lambda step: max(1.0,
↳step**0.5))
```

Note

- For most use cases, the default hyperparameters work well
- Consider using decouple=True for better generalization (similar to AdamW)
- The clip_lambda function helps stabilize training in early steps

clip_lambda

__setstate__ (*state*)

Restore optimizer state from a checkpoint.

This method handles backward compatibility when loading optimizer state from older versions. It ensures all required fields are present with default values and properly converts step counters to tensors if needed.

Key responsibilities: 1. Set default values for newly added hyperparameters 2. Convert old-style scalar step counters to tensor format 3. Place step tensors on appropriate devices based on capturable/fused modes

Parameters

state (*dict*) – Optimizer state dictionary (typically from `torch.load()`).

Note

- This enables loading checkpoints saved with older ADOPT versions
- Step counters are converted to appropriate device/dtype for compatibility
- Capturable and fused modes require step tensors on parameter devices

step (*closure=None*)

Perform a single optimization step.

This method executes one iteration of the ADOPT optimization algorithm across all parameter groups. It orchestrates the following workflow:

1. Optionally evaluates a closure to recompute the loss (useful for algorithms like LBFGS or when loss needs multiple evaluations)
2. For each parameter group: - Collects parameters with gradients and their associated state - Extracts hyperparameters (betas, learning rate, etc.) - Calls the functional `adopt()` API to perform the actual update
3. Returns the loss value if a closure was provided

The functional API (`adopt()`) handles three execution modes: - Single-tensor: Updates one parameter at a time (default, JIT-compatible) - Multi-tensor (foreach): Batches operations for better performance - Fused: Uses fused CUDA kernels (not yet implemented)

Gradient scaling support: This method is compatible with automatic mixed precision (AMP) training. It can access `grad_scale` and `found_inf` attributes for gradient unscaling and inf/nan detection when used with `GradScaler`.

Parameters

closure (*Callable, optional*) – A callable that reevaluates the model and returns the loss. The closure should: - Enable gradients (`torch.enable_grad()`) - Compute forward pass - Compute loss - Compute backward pass - Return the loss value Example: `lambda: (loss := model(x), loss.backward(), loss)[-1]` Default: `None`

Returns

The loss value returned by the closure, or `None` if no closure was provided.

Return type

`Optional[Tensor]`

Example

```
>>> # Standard usage
>>> loss = criterion(model(input), target)
>>> loss.backward()
>>> optimizer.step()
```

```
>>> # With closure (e.g., for line search)
>>> def closure():
...     optimizer.zero_grad()
...     output = model(input)
...     loss = criterion(output, target)
...     loss.backward()
...     return loss
>>> loss = optimizer.step(closure)
```

Note

- Call `zero_grad()` before computing gradients for the next step
- CUDA graph capture is checked for safety when `capturable=True`
- The method is thread-safe for different parameter groups

`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.

This is the main functional interface for the ADOPT optimization algorithm. It dispatches to one of three implementations based on the execution mode:

1. **Single-tensor mode** (default): Updates parameters one at a time - Compatible with `torch.jit.script` - More flexible but slower - Used when `foreach=False` or automatically for small models
2. **Multi-tensor (foreach) mode**: Batches operations across tensors - 2-3x faster on GPU through vectorization - Groups tensors by device/dtype automatically - Used when `foreach=True`
3. **Fused mode**: Uses specialized fused kernels (not yet implemented) - Would provide maximum performance - Currently raises `RuntimeError` if enabled

Algorithm overview (ADOPT):

ADOPT adapts learning rates per-parameter while using momentum on normalized gradients. The key innovation is normalizing gradients before momentum, which provides more stable training than standard Adam.

Mathematical formulation:

```
# Normalize gradient by its historical variance normed_g_t = g_t / √(v_t + ε)
# Optional gradient clipping for stability normed_g_t = clip(normed_g_t, threshold(t))
# Momentum on normalized gradients (key difference from Adam) m_t = β1 * m_{t-1} + (1 - β1) * normed_g_t
# Parameter update θ_t = θ_{t-1} - α * m_t
# Update variance estimate v_t = β2 * v_{t-1} + (1 - β2) * g_t2
```

where:

- θ : parameters

- g : gradients
- m : first moment (momentum of normalized gradients)
- v : second moment (variance of raw gradients)
- α : learning rate
- β_1, β_2 : exponential decay rates
- ϵ : numerical stability constant
- `clip()`: gradient clipping function based on step

Automatic mode selection:

When `foreach` and `fused` are both `None` (default), the function automatically selects the best implementation based on: - Parameter types and devices - Whether differentiable mode is enabled - Learning rate type (float vs Tensor) - Capturable mode requirements

param params

Parameters to optimize.

type params

List[Tensor]

param grads

Gradients for each parameter.

type grads

List[Tensor]

param exp_avgs

First moment estimates (momentum).

type exp_avgs

List[Tensor]

param exp_avg_sqs

Second moment estimates (variance).

type exp_avg_sqs

List[Tensor]

param state_steps

Step counters (must be singleton tensors).

type state_steps

List[Tensor]

param foreach

Whether to use multi-tensor implementation. `None`: auto-select based on configuration (default).

type foreach

Optional[bool]

param capturable

If `True`, ensure CUDA graph capture safety.

type capturable

bool

param differentiable

If `True`, allow gradients through optimization step.

type differentiable
bool

param fused
If True, use fused kernels (not implemented).

type fused
Optional[bool]

param grad_scale
Gradient scaler for AMP training.

type grad_scale
Optional[Tensor]

param found_inf
Flag for inf/nan detection in AMP.

type found_inf
Optional[Tensor]

param has_complex
Whether any parameters are complex-valued.

type has_complex
bool

param beta1
Exponential decay rate for first moment (momentum). Typical range: 0.9-0.95.

type beta1
float

param beta2
Exponential decay rate for second moment (variance). Typical range: 0.999-0.9999 (higher than Adam).

type beta2
float

param lr
Learning rate. Can be a scalar Tensor for dynamic learning rate with capturable=True.

type lr
Union[float, Tensor]

param clip_lambda
Function that maps step number to gradient clipping threshold. None disables clipping.

type clip_lambda
Optional[Callable[[int], float]]

param weight_decay
Weight decay coefficient (L2 penalty).

type weight_decay
float

param decouple
If True, use decoupled weight decay (AdamW-style). Recommended for better generalization.

type decouple
bool

param eps

Small constant for numerical stability in normalization.

type eps

float

param maximize

If True, maximize objective instead of minimize.

type maximize

bool

raises RuntimeError

If torch.jit.script is used with foreach or fused.

raises RuntimeError

If state_steps contains non-tensor elements.

raises RuntimeError

If fused=True (not yet implemented).

raises RuntimeError

If lr is Tensor with foreach=True and capturable=False.

Example

```
>>> # Typically called by ADOPT optimizer, not directly
>>> adopt (
...     params=[p1, p2],
...     grads=[g1, g2],
...     exp_avgs=[m1, m2],
...     exp_avg_sqs=[v1, v2],
...     state_steps=[step1, step2],
...     beta1=0.9,
...     beta2=0.9999,
...     lr=0.001,
...     clip_lambda=lambda s: s**0.25,
...     weight_decay=0.01,
...     decouple=True,
...     eps=1e-6,
...     maximize=False,
... )
```

Note

- For distributed training, this API is compatible with torch/distributed/optim
- The foreach mode is generally preferred for GPU training
- Complex parameters are handled transparently by viewing as real
- First optimization step only initializes variance, doesn't update parameters

See also

- ADOPT class: High-level optimizer interface
- `_single_tensor_adopt`: Single-tensor implementation details
- `_multi_tensor_adopt`: Multi-tensor implementation details

dicее.models.base_model

Classes

<code>BaseKGELightning</code>	Base class for all neural network modules.
<code>BaseKGE</code>	Base class for all neural network modules.
<code>IdentityClass</code>	Base class for all neural network modules.

Module Contents

class dicее.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 sub-modules 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 **:param-ref:~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 sub-modules 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

$\mathbf{x} (B \times 2 \times T)$

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* × 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)
        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

<i>Keci</i>	Base class for all neural network modules.
<i>CKeci</i>	Without learning dimension scaling
<i>DeCaL</i>	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 sub-modules 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 $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_{ir_k} - h_{kr_i}) e_i e_k$

σ_{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 $\sigma_{qq} = \sum_{j=1}^{q-1} \sum_{k=j+1}^q (h_{jr_k} - h_{kr_j}) e_j e_k$ 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*)

$\sum_{i=1}^{p-1} \sum_{j=p+1}^{p+q} (h_{ir_j} - h_{jr_i}) e_i e_j$

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
    
$$h = h_0 + \sum_{i=1}^p h_i e_i + \sum_{j=p+1}^{p+q} h_j e_j$$


$$r = r_0 + \sum_{i=1}^p r_i e_i + \sum_{j=p+1}^{p+q} r_j e_j$$


$$e_i^2 = +1 \text{ for } i \leq p, e_j^2 = -1 \text{ for } p < j \leq p+q, e_i e_j = -e_j e_i \text{ for } i \neq j$$


$$h r = \sigma_0 + \sigma_p + \sigma_q + \sigma_{pp} + \sigma_{pq} + \sigma_{qq}$$

    where
    (1)  $\sigma_0 = h_0 r_0 + \sum_{i=1}^p (h_i r_i) e_i - \sum_{j=p+1}^{p+q} (h_j r_j) e_j$ 
    (2)  $\sigma_p = \sum_{i=1}^p (h_i r_0 + h_0 r_i) e_i$ 
    (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 functions are identical **Parameter** ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, **IEI**) 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 sub-modules as regular attributes:

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

(continues on next 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.

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,

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

and return:

$$\sigma_0 t = \sigma_0 \cdot t_0 = s0 + s1 - s2s3, s4 \text{ and } s5$$

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

$$\sigma_p p = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (h_i r_{i'} - h_{i'} r_i) (\text{model the interactions between } e_i \text{ and } e_{i'} \text{ for } 1 \leq i, i' \leq p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{j'} r_j)$$

For different base vector interactions, we have

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

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 functions are identical Parameter ——— x: torch.LongTensor with (n,) shape :rtype: torch.FloatTensor with (n, **IEI**) 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:

$$\sigma_{p,p} = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (x_i y_{i'} - x_{i'} y_i)$$

σ_{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

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

σ_{qq} 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)

```

dicее.models.complex

Classes

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

Module Contents

```

class dicее.models.complex.ConEx(args)
    Bases: dicее.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 sub-modules 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*) \rightarrow *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

continues on next page

Table 31 – continued from previous page

<i>GFMult</i>	Learning Knowledge Neural Graphs
<i>FMult2</i>	Learning Knowledge Neural Graphs
<i>LFMult1</i>	Embedding with trigonometric functions. We represent all entities and relations in the complex number space as:
<i>LFMult</i>	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*) \rightarrow torch.FloatTensor

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

forward_triples (*idx_triple: torch.Tensor*) \rightarrow 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*) \rightarrow torch.FloatTensor

function (*list_W, list_b*)

trapezoid (*list_W, list_b*)

forward_triples (*idx_triple: torch.Tensor*) \rightarrow 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: $f(x) = \sum_{k=0}^{k=d-1} w_k e^{kix}$. 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:  $f(x) = \sum_{i=0}^{d-1} a_i x^i$  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.  $h = \text{sigma}(wh^T x + bh)$ ,  $r = \text{sigma}(wr^T x + br)$ ,  $t = \text{sigma}(wt^T x + bt)$ 

    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:

        
$$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*b_j*c_k\}\{1+(i+j+k)\%d\}$$

        1. generate the range for i,j and k from [0 d-1]
        2. perform  $\text{dfrac}\{a_i*b_j*c_k\}\{1+(i+j+k)\%d\}$  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:

        
$$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*c_j*b_k - b_i*c_j*a_k\}\{(1+(i+j)\%d)(1+k)\}$$

        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 $[0, 1, \dots, d]$ and return a vector tensor ($\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d$,

$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d$)

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 $[0, 1, \dots, d]$

and return a tensor ($\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d$,

$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d$)

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

<i>OMult</i>	Base class for all neural network modules.
<i>ConvO</i>	Base class for all neural network modules.

continues on next page

Table 33 – continued from previous page

<i>AConvO</i>	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 sub-modules 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)
abstractmethod forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)
```

dicее.models.quaternion

Classes

<i>QMult</i>	Base class for all neural network modules.
<i>ConvQ</i>	Convolutional Quaternion Knowledge Graph Embeddings
<i>ACConvQ</i>	Additive Convolutional Quaternion Knowledge Graph Embeddings

Functions

```
quaternion_mul_with_unit_norm(*, Q_1, Q_2)
```

Module Contents

```
dicее.models.quaternion.quaternion_mul_with_unit_norm(*, Q_1, Q_2)
```

```
class dicее.models.quaternion.QMult (args)
```

Bases: *dicее.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 sub-modules 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

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

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

class dicee.models.real.CoKEConfig
    Configuration for the CoKE (Contextualized Knowledge Graph Embedding) model.
    block_size
        Sequence length for transformer (3 for triples: head, relation, tail)
    vocab_size
        Total vocabulary size (num_entities + num_relations)
    n_layer
        Number of transformer layers

```

n_head
Number of attention heads per layer

n_embd
Embedding dimension (set to match model embedding_dim)

dropout
Dropout rate applied throughout the model

bias
Whether to use bias in linear layers

causal
Whether to use causal masking (False for bidirectional attention)

block_size: int = 3

vocab_size: int = None

n_layer: int = 6

n_head: int = 8

n_embd: int = None

dropout: float = 0.3

bias: bool = True

causal: bool = False

```
class dicee.models.real.CoKE(args, config: CoKEConfig = CoKEConfig())
```

Bases: *`dicee.models.base_model.BaseKGE`*

Contextualized Knowledge Graph Embedding (CoKE) model. Based on: <https://arxiv.org/pdf/1911.02168>.

CoKE uses a transformer encoder to learn contextualized representations of entities and relations. For link prediction, it predicts masked elements in (head, relation, tail) triples using bidirectional attention, similar to BERT's masked language modeling approach.

The model creates a sequence [head_emb, relation_emb, mask_emb], adds positional embeddings, and processes it through transformer layers to predict the tail entity.

name = 'CoKE'

config

pos_emb

mask_emb

blocks

ln_f

coke_dropout

forward_k_vs_all (*x: torch.Tensor*)

score (*emb_h, emb_r, emb_t*)

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

dicee.models.static_funcs

Functions

```
quaternion_mul(→ Tuple[torch.Tensor, torch.Tensor, ...])
```

Module Contents

```
dicee.models.static_funcs.quaternion_mul(*, Q_1, Q_2)
→ Tuple[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

<i>ByteE</i>	Base class for all neural network modules.
<i>LayerNorm</i>	LayerNorm but with an optional bias. PyTorch doesn't support simply bias=False
<i>SelfAttention</i>	Base class for all neural network modules.
<i>MLP</i>	Base class for all neural network modules.
<i>Block</i>	Base class for all neural network modules.
<i>GPTConfig</i>	
<i>GPT</i>	Base class for all neural network modules.

Module Contents

```
class dicee.models.transformers.ByteE(*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 sub-modules 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 = '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.SelfAttention(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 sub-modules 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.

`c_attn`

`c_proj`

`attn_dropout`

`resid_dropout`

`n_head`

`n_embd`

`dropout`

`causal`

`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 sub-modules 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.

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 sub-modules 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`

`causal: bool = True`

```
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 sub-modules 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

<i>ADOPT</i>	ADOPT Optimizer.
<i>BaseKGELightning</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>IdentityClass</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>Block</i>	Base class for all neural network modules.
<i>DistMult</i>	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
<i>TransE</i>	Translating Embeddings for Modeling
<i>Shallom</i>	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
<i>Pyke</i>	A Physical Embedding Model for Knowledge Graphs
<i>CoKEConfig</i>	Configuration for the CoKE (Contextualized Knowledge Graph Embedding) model.
<i>CoKE</i>	Contextualized Knowledge Graph Embedding (CoKE) model.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>ConEx</i>	Convolutional ComplEx Knowledge Graph Embeddings
<i>AConEx</i>	Additive Convolutional ComplEx Knowledge Graph Embeddings
<i>Complex</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>IdentityClass</i>	Base class for all neural network modules.
<i>QMult</i>	Base class for all neural network modules.
<i>ConvQ</i>	Convolutional Quaternion Knowledge Graph Embeddings
<i>AConvQ</i>	Additive Convolutional Quaternion Knowledge Graph Embeddings
<i>BaseKGE</i>	Base class for all neural network modules.
<i>IdentityClass</i>	Base class for all neural network modules.
<i>OMult</i>	Base class for all neural network modules.
<i>ConvO</i>	Base class for all neural network modules.
<i>AConvO</i>	Additive Convolutional Octonion Knowledge Graph Embeddings
<i>Keci</i>	Base class for all neural network modules.
<i>CKeci</i>	Without learning dimension scaling
<i>DeCaL</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>PykeenKGE</i>	A class for using knowledge graph embedding models implemented in Pykeen
<i>BaseKGE</i>	Base class for all neural network modules.
<i>FMult</i>	Learning Knowledge Neural Graphs
<i>GFMult</i>	Learning Knowledge Neural Graphs
<i>FMult2</i>	Learning Knowledge Neural Graphs
<i>LFMult1</i>	Embedding with trigonometric functions. We represent all entities and relations in the complex number space as:
<i>LFMult</i>	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:

continues on next page

Table 41 – continued from previous page

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

Functions

<i>quaternion_mul</i> (\rightarrow Tuple[torch.Tensor, torch.Tensor, ...])	Perform quaternion multiplication
<i>quaternion_mul_with_unit_norm</i> (*, Q_1, Q_2)	
<i>octonion_mul</i> (*, O_1, O_2)	
<i>octonion_mul_norm</i> (*, 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`

ADOPT Optimizer.

ADOPT is an adaptive learning rate optimization algorithm that combines momentum-based updates with adaptive per-parameter learning rates. It uses exponential moving averages of gradients and squared gradients, with gradient clipping for stability.

The algorithm performs the following key operations: 1. Normalizes gradients by the square root of the second moment estimate 2. Applies optional gradient clipping based on the training step 3. Updates parameters using momentum-smoothed normalized gradients 4. Supports decoupled weight decay (AdamW-style) or L2 regularization

Mathematical formulation:

$$m_t = \beta_1 * m_{\{t-1\}} + (1 - \beta_1) * \text{clip}(g_t / \sqrt{v_t}) \quad v_t = \beta_2 * v_{\{t-1\}} + (1 - \beta_2) * g_t^2 \quad \theta_t = \theta_{\{t-1\}} - \alpha * m_t$$

where:

- θ_t : parameter at step t
- g_t : gradient at step t
- m_t : first moment estimate (momentum)
- v_t : second moment estimate (variance)
- α : learning rate
- β_1, β_2 : exponential decay rates
- `clip()`: optional gradient clipping function

Reference:

Original implementation: <https://github.com/iShohei220/adopt>

Parameters

- **params** (*ParamsT*) – Iterable of parameters to optimize or dicts defining parameter groups.
- **lr** (*float or Tensor, optional*) – Learning rate. Can be a float or 1-element Tensor. Default: 1e-3
- **betas** (*Tuple[float, float], optional*) – Coefficients (β_1, β_2) for computing running averages of gradient and its square. β_1 controls momentum, β_2 controls variance. Default: (0.9, 0.9999)
- **eps** (*float, optional*) – Term added to denominator to improve numerical stability. Default: 1e-6
- **clip_lambda** (*Callable[[int], float], optional*) – Function that takes the step number and returns the gradient clipping threshold. Common choices: - lambda step: step**0.25 (default, gradually increases clipping threshold) - lambda step: 1.0 (constant clipping) - None (no clipping) Default: lambda step: step**0.25
- **weight_decay** (*float, optional*) – Weight decay coefficient (L2 penalty). Default: 0.0
- **decouple** (*bool, optional*) – If True, uses decoupled weight decay (AdamW-style), applying weight decay directly to parameters. If False, adds weight decay to gradients (L2 regularization). Default: False
- **foreach** (*bool, optional*) – If True, uses the faster foreach implementation for multi-tensor operations. Default: None (auto-select)
- **maximize** (*bool, optional*) – If True, maximizes parameters instead of minimizing. Useful for reinforcement learning. Default: False
- **capturable** (*bool, optional*) – If True, the optimizer is safe to capture in a CUDA graph. Requires learning rate as Tensor. Default: False
- **differentiable** (*bool, optional*) – If True, the optimization step can be differentiated. Useful for meta-learning. Default: False
- **fused** (*bool, optional*) – If True, uses fused kernel implementation (currently not supported). Default: None

Raises

- **ValueError** – If learning rate, epsilon, betas, or weight_decay are invalid.
- **RuntimeError** – If fused is enabled (not currently supported).
- **RuntimeError** – If lr is a Tensor with foreach=True and capturable=False.

Example

```
>>> # Basic usage
>>> optimizer = ADOPT(model.parameters(), lr=0.001)
>>> optimizer.zero_grad()
>>> loss.backward()
>>> optimizer.step()
```

```
>>> # With decoupled weight decay
>>> optimizer = ADOPT(model.parameters(), lr=0.001, weight_decay=0.01, ↵
↵decouple=True)
```

```
>>> # Custom gradient clipping
>>> optimizer = ADOPT(model.parameters(), clip_lambda=lambda step: max(1.0,
↳ step**0.5))
```

Note

- For most use cases, the default hyperparameters work well
- Consider using decouple=True for better generalization (similar to AdamW)
- The clip_lambda function helps stabilize training in early steps

clip_lambda

__setstate__(state)

Restore optimizer state from a checkpoint.

This method handles backward compatibility when loading optimizer state from older versions. It ensures all required fields are present with default values and properly converts step counters to tensors if needed.

Key responsibilities: 1. Set default values for newly added hyperparameters 2. Convert old-style scalar step counters to tensor format 3. Place step tensors on appropriate devices based on capturable/fused modes

Parameters

state (dict) – Optimizer state dictionary (typically from torch.load()).

Note

- This enables loading checkpoints saved with older ADOPT versions
- Step counters are converted to appropriate device/dtype for compatibility
- Capturable and fused modes require step tensors on parameter devices

step(closure=None)

Perform a single optimization step.

This method executes one iteration of the ADOPT optimization algorithm across all parameter groups. It orchestrates the following workflow:

1. Optionally evaluates a closure to recompute the loss (useful for algorithms like LBFGS or when loss needs multiple evaluations)
2. For each parameter group: - Collects parameters with gradients and their associated state - Extracts hyperparameters (betas, learning rate, etc.) - Calls the functional adopt() API to perform the actual update
3. Returns the loss value if a closure was provided

The functional API (adopt()) handles three execution modes: - Single-tensor: Updates one parameter at a time (default, JIT-compatible) - Multi-tensor (foreach): Batches operations for better performance - Fused: Uses fused CUDA kernels (not yet implemented)

Gradient scaling support: This method is compatible with automatic mixed precision (AMP) training. It can access grad_scale and found_inf attributes for gradient unscaling and inf/nan detection when used with GradScaler.

Parameters

closure (*Callable, optional*) – A callable that reevaluates the model and returns the loss. The closure should: - Enable gradients (`torch.enable_grad()`) - Compute forward pass - Compute loss - Compute backward pass - Return the loss value Example: `lambda: (loss := model(x), loss.backward(), loss)[-1]` Default: None

Returns

The loss value returned by the closure, or None if no closure was provided.

Return type

Optional[Tensor]

Example

```
>>> # Standard usage
>>> loss = criterion(model(input), target)
>>> loss.backward()
>>> optimizer.step()
```

```
>>> # With closure (e.g., for line search)
>>> def closure():
...     optimizer.zero_grad()
...     output = model(input)
...     loss = criterion(output, target)
...     loss.backward()
...     return loss
>>> loss = optimizer.step(closure)
```

Note

- Call `zero_grad()` before computing gradients for the next step
- CUDA graph capture is checked for safety when `capturable=True`
- The method is thread-safe for different parameter groups

```
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 sub-modules as regular attributes:

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

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
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 **:param-ref:~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.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 sub-modules 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 ( $B \times 2 \times T$ )

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 ( $B \times 2 \times T$ )

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 sub-modules 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 sub-modules 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
         $\mathbf{x}$  ( $B \times 2 \times T$ )

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

        •  $\mathbf{x}$ 

        •  $\mathbf{y\_idx}$ 

        • ordered_bpe_entities

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

    Parameters
         $\mathbf{x}$ 

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 ($B \times 2 \times T$)

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

`class dicee.models.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 sub-modules 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.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.CoKEConfig
    Configuration for the CoKE (Contextualized Knowledge Graph Embedding) model.

    block_size
        Sequence length for transformer (3 for triples: head, relation, tail)

    vocab_size
        Total vocabulary size (num_entities + num_relations)

    n_layer
        Number of transformer layers

    n_head
        Number of attention heads per layer

    n_embd
        Embedding dimension (set to match model embedding_dim)

    dropout
        Dropout rate applied throughout the model

    bias
        Whether to use bias in linear layers

    causal
        Whether to use causal masking (False for bidirectional attention)

    block_size: int = 3

    vocab_size: int = None

    n_layer: int = 6

    n_head: int = 8

    n_embd: int = None

    dropout: float = 0.3

    bias: bool = True

    causal: bool = False

```

```
class dicee.models.CoKE(args, config: CoKEConfig = CoKEConfig())
```

Bases: `dicee.models.base_model.BaseKGE`

Contextualized Knowledge Graph Embedding (CoKE) model. Based on: <https://arxiv.org/pdf/1911.02168>.

CoKE uses a transformer encoder to learn contextualized representations of entities and relations. For link prediction, it predicts masked elements in (head, relation, tail) triples using bidirectional attention, similar to BERT's masked language modeling approach.

The model creates a sequence [head_emb, relation_emb, mask_emb], adds positional embeddings, and processes it through transformer layers to predict the tail entity.

name = 'CoKE'

config

pos_emb

mask_emb

blocks

ln_f

coke_dropout

forward_k_vs_all (*x*: torch.Tensor)

score (*emb_h*, *emb_r*, *emb_t*)

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

```
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
         $\mathbf{x}$  ( $B \times 2 \times T$ )

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

    •  $\mathbf{x}$ 

    •  $\mathbf{y\_idx}$ 

    • ordered_bpe_entities

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

    Parameters
     $\mathbf{x}$ 

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

    • ( $\mathbf{b}$  ( $x$  shape))

    • 3

    •  $\mathbf{t}$ )

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

    Parameters
         $\mathbf{x}$  ( $B \times 2 \times T$ )

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 sub-modules 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 sub-modules 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
         $\mathbf{x} (B \times 2 \times T)$ 

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 ($B \times 2 \times T$)

```
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 sub-modules 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.

`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:

```
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 sub-modules 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
         $\mathbf{x} (B \times 2 \times T)$ 

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 ($B \times 2 \times T$)

```
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 sub-modules 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.

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):
    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 sub-modules 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 $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k$

σ_{pp} captures the interactions between along p bases For instance, let $p \in \{e_1, e_2, e_3\}$, 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 $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_{j \ r \ k} - h_{k \ r \ j}) e_j e_k$ σ_{qq} 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

$$h = h_0 + \sum_{i=1}^p h_{i \ r \ i} e_i + \sum_{j=p+1}^{p+q} h_{j \ r \ j} e_j$$

$$r = r_0 + \sum_{i=1}^p r_{i \ r \ i} e_i + \sum_{j=p+1}^{p+q} r_{j \ r \ j} e_j$$

$$e_i^2 = +1 \text{ for } i \leq p, e_j^2 = -1 \text{ for } p < j \leq p+q, e_i e_j = -e_j e_i \text{ for } i$$

e_j

$h \ r = \sigma_0 + \sigma_p + \sigma_q + \sigma_{pp} + \sigma_{qq} + \sigma_{pq}$ where

(1) $\sigma_0 = h_0 \ r_0 + \sum_{i=1}^p (h_0 \ r_{i \ r \ i} e_i - \sum_{j=p+1}^{p+q} (h_{j \ r \ j} e_j$

(2) $\sigma_p = \sum_{i=1}^p (h_0 \ r_{i \ r \ i} + h_{i \ r \ i} e_i$

(3) $\sigma_q = \sum_{j=p+1}^{p+q} (h_0 \ r_{j \ r \ j} + h_{j \ r \ j} e_j$

(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 function are identical Parameter ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, **IEI**) 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 sub-modules 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*) \rightarrow *torch.FloatTensor*

Parameter

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

rtype

torch.FloatTensor with (n) shape

cl_pqr (a : *torch.tensor*) \rightarrow *torch.tensor*

Input: *tensor*(batch_size, emb_dim) \rightarrow 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,

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

and return:

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

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

$$\sigma_p p = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (h_i r_{i'} - h_{i'} r_i) (modelstheinteractionsbetweene_i \text{ and } e_{i'} \text{ for } 1 \leq i, i' \leq p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{j'} r_j)$$

For different base vector interactions, we have

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

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 functions are identical Parameter ——— *x*: torch.LongTensor with (n,) shape :rtype: torch.FloatTensor with (n, **IE**) 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* e₁, e₂, e₃, we compute interactions between e₁ e₂, e₁ e₃, and e₂ e₃ 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., $e_1 e_1, e_1 e_2, e_1 e_3$,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals: $e_1 e_2, e_1 e_3, e_2 e_3$.

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 sub-modules 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

$\mathbf{x} (B \times 2 \times T)$

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* × 2 × *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)

abstractmethod 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 sub-
    modules 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

$\mathbf{x} (B \times 2 \times T)$

```

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 dictee.models.FMult (args)
    Bases: dictee.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: $f(x) = \sum_{k=0}^{d-1} w_k e^{kix}$. and use the three different 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: $f(x) = \sum_{i=0}^{d-1} a_i x^i$ and use the three different 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. $h = \text{sigma}(wh^T x + bh)$, $r = \text{sigma}(wr^T x + br)$,
 $t = \text{sigma}(wt^T x + bt)$

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:

$$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*b_j*c_k\} \{1+(i+j+k)*d\}$$

1. generate the range for i,j and k from [0 d-1]
2. perform $\text{dfrac}\{a_i*b_j*c_k\} \{1+(i+j+k)*d\}$ 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:

$$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*c_j*b_k - b_i*c_j*a_k\} \{(1+(i+j)*d)(1+k)\}$$

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. $\text{score} = \langle h, r, t \rangle$

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 ($\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d$,

$$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d)$$

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 ($\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d$,

$$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d)$$

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: str, 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*)

abstractmethod 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 byte pair encoding and add padding

preprocess_with_pandas () → None
 Preprocess with pandas: add reciprocal triples, construct vocabulary, and index datasets

preprocess_with_polars () → None
 Preprocess with polars: add reciprocal triples and create indexed datasets

`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

<code>polars_dataframe_indexer</code> (\rightarrow polars.DataFrame)	Replaces 'subject', 'relation', and 'object' columns in the input Polars DataFrame with their corresponding index values
<code>pandas_dataframe_indexer</code> (\rightarrow pandas.DataFrame)	Replaces 'subject', 'relation', and 'object' columns in the input Pandas DataFrame with their corresponding index values
<code>apply_reciprocal_or_noise</code> (add_reciprocal, eval_model)	Add reciprocal triples if conditions are met
<code>timeit</code> (func)	
<code>read_with_polars</code> (\rightarrow polars.DataFrame)	Load and Preprocess via Polars
<code>read_with_pandas</code> (data_path[, read_only_few, ...])	Load and Preprocess via Pandas
<code>read_from_disk</code> (\rightarrow Tuple[polars.DataFrame, pandas.DataFrame])	
<code>count_triples</code> (\rightarrow int)	Returns the total number of triples in the triple store.
<code>fetch_worker</code> (endpoint, offsets, chunk_size, ...)	Worker process: fetch assigned chunks and save to disk with per-worker tqdm.
<code>read_from_triple_store_with_polars</code> (endpoint[, ...])	Main function to read all triples in parallel, save as Parquet, and load into Polars dataframe.
<code>read_from_triple_store_with_pandas</code> ([endpoint]	Read triples from triple store into pandas dataframe
<code>get_er_vocab</code> (data[, file_path])	
<code>get_re_vocab</code> (data[, file_path])	
<code>get_ee_vocab</code> (data[, file_path])	
<code>create_constraints</code> (triples[, file_path])	
<code>load_with_pandas</code> (\rightarrow None)	Deserialize data
<code>save_numpy_ndarray</code> (*, data, file_path)	
<code>load_numpy_ndarray</code> (*, file_path)	
<code>save_pickle</code> (*, data[, file_path])	
<code>load_pickle</code> (*[, file_path])	
<code>create_recipriocal_triples</code> (x)	Add inverse triples into dask dataframe
<code>dataset_sanity_checking</code> (\rightarrow 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
 \rightarrow 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_reciprocal_or_noise (add_reciprocal: bool,  
    eval_model: str, df: object = None, info: str = None)
```

Add reciprocal triples if conditions are met

```
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)
```

Load and Preprocess via Pandas

```
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.count_triples (endpoint: str) → int
```

Returns the total number of triples in the triple store.

```
dicee.read_preprocess_save_load_kg.util.fetch_worker (endpoint: str, offsets: list[int],  
    chunk_size: int, output_dir: str, worker_id: int)
```

Worker process: fetch assigned chunks and save to disk with per-worker tqdm.

```
dicee.read_preprocess_save_load_kg.util.read_from_triple_store_with_polars (  
    endpoint: str, chunk_size: int = 500000, output_dir: str = 'triples_parquet')
```

Main function to read all triples in parallel, save as Parquet, and load into Polars dataframe.

```
dicee.read_preprocess_save_load_kg.util.read_from_triple_store_with_pandas (  
    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. Create constrained 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

<i>PreprocessKG</i>	Preprocess the data in memory
<i>LoadSaveToDisk</i>	
<i>ReadFromDisk</i>	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 byte pair encoding and add padding

`preprocess_with_pandas()` → None

Preprocess with pandas: add reciprocal triples, construct vocabulary, and index datasets

`preprocess_with_polars()` → None

Preprocess with polars: add reciprocal triples and create indexed datasets

`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`

Functions

`get_default_arguments()`

`index(args)`

`root()`

`search_embeddings(q)`

`retrieve_embeddings(q)`

`search_embeddings_batch(request)`

continues on next page

Table 52 – continued from previous page

`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``Bases: pydantic.BaseModel``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

continues on next page

Table 53 – continued from previous page

`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`

Static utility functions for DICE embeddings.

This module provides utility functions for model initialization, data loading, serialization, and various helper operations.

Attributes

`MODEL_REGISTRY`

Functions

<code>create_recipriocal_triples(→ das.DataFrame)</code>	pan-	Add inverse triples to a DataFrame.
<code>get_er_vocab(→ Dict[Tuple[int, int], List[int]])</code>		Build entity-relation to tail vocabulary.
<code>get_re_vocab(→ Dict[Tuple[int, int], List[int]])</code>		Build relation-entity (tail) to head vocabulary.
<code>get_ee_vocab(→ Dict[Tuple[int, int], List[int]])</code>		Build entity-entity to relation vocabulary.
<code>timeit(→ Callable)</code>		Decorator to measure and print execution time and memory usage.
<code>save_pickle(→ None)</code>		Save data to a pickle file.
<code>load_pickle(→ object)</code>		Load data from a pickle file.
<code>load_term_mapping(→ Union[dict, lars.DataFrame])</code>	po-	Load term-to-index mapping from pickle or CSV file.
<code>select_model(args[, is_continual_training, age_path])</code>	stor-	
<code>load_model(→ Tuple[object, Tuple[dict, dict]])</code>		Load weights and initialize pytorch module from namespace arguments
<code>load_model_ensemble(...)</code>		Construct Ensemble Of weights and initialize pytorch module from namespace arguments
<code>save_numpy_ndarray(*, data, file_path)</code>		
<code>numpy_data_type_changer(→ numpy.ndarray)</code>		Detect most efficient data type for a given triples
<code>save_checkpoint_model(→ None)</code>		Store Pytorch model into disk
<code>store(→ None)</code>		
<code>add_noisy_triples(→ pandas.DataFrame)</code>		Add randomly constructed triples
<code>read_or_load_kg(args, cls)</code>		

continues on next page

Table 55 – continued from previous page

<code>initialize_model(...)</code>	Initialize a knowledge graph embedding model.
<code>load_json(→ Dict)</code>	Load JSON file into a dictionary.
<code>save_embeddings(→ None)</code>	Save embeddings to a CSV file.
<code>vocab_to_parquet(vocab_to_idx, name, ...)</code>	
<code>create_experiment_folder(→ str)</code>	Create a timestamped experiment folder.
<code>continual_training_setup_executor(→ None)</code>	
<code>exponential_function(→ torch.FloatTensor)</code>	
<code>load_numpy(→ numpy.ndarray)</code>	
<code>evaluate(entity_to_idx, scores, easy_answers, # @TODO: CD: Renamed this function hard_answers)</code>	
<code>download_file(url[, destination_folder])</code>	
<code>download_files_from_url(→ None)</code>	
<code>download_pretrained_model(→ str)</code>	
<code>write_csv_from_model_parallel(path)</code>	Create
<code>from_pretrained_model_write_embeddings_int(→ None)</code>	

Module Contents

`dicee.static_funcs.MODEL_REGISTRY: Dict[str, Tuple[Type, str]]`

`dicee.static_funcs.create_recipriocal_triples(df: pandas.DataFrame) → pandas.DataFrame`

Add inverse triples to a DataFrame.

For each triple (s, p, o), creates an inverse triple (o, p_inverse, s).

Parameters

df – DataFrame with ‘subject’, ‘relation’, and ‘object’ columns.

Returns

DataFrame with original and inverse triples concatenated.

`dicee.static_funcs.get_er_vocab(data: numpy.ndarray, file_path: str | None = None)`

→ Dict[Tuple[int, int], List[int]]

Build entity-relation to tail vocabulary.

Parameters

- **data** – Array of triples with shape (n, 3) where columns are (head, relation, tail).
- **file_path** – Optional path to save the vocabulary as pickle.

Returns

Dictionary mapping (head, relation) pairs to list of tail entities.

`dicee.static_funcs.get_re_vocab(data: numpy.ndarray, file_path: str | None = None)`

→ Dict[Tuple[int, int], List[int]]

Build relation-entity (tail) to head vocabulary.

Parameters

- **data** – Array of triples with shape (n, 3) where columns are (head, relation, tail).
- **file_path** – Optional path to save the vocabulary as pickle.

Returns

Dictionary mapping (relation, tail) pairs to list of head entities.

```
dicee.static_funcs.get_ee_vocab(data: numpy.ndarray, file_path: str | None = None)
→ Dict[Tuple[int, int], List[int]]
```

Build entity-entity to relation vocabulary.

Parameters

- **data** – Array of triples with shape (n, 3) where columns are (head, relation, tail).
- **file_path** – Optional path to save the vocabulary as pickle.

Returns

Dictionary mapping (head, tail) pairs to list of relations.

```
dicee.static_funcs.timeit(func: Callable) → Callable
```

Decorator to measure and print execution time and memory usage.

Parameters

func – Function to be timed.

Returns

Wrapped function that prints timing information.

```
dicee.static_funcs.save_pickle(*, data: object | None = None, file_path: str) → None
```

Save data to a pickle file.

Note: Consider using more portable formats (JSON, Parquet) for new code.

Parameters

- **data** – Object to serialize. If None, nothing is saved.
- **file_path** – Path where the pickle file will be saved.

```
dicee.static_funcs.load_pickle(file_path: str) → object
```

Load data from a pickle file.

Note: Consider using more portable formats (JSON, Parquet) for new code.

Parameters

file_path – Path to the pickle file.

Returns

Deserialized object from the pickle file.

```
dicee.static_funcs.load_term_mapping(file_path: str) → dict | polars.DataFrame
```

Load term-to-index mapping from pickle or CSV file.

Attempts to load from pickle first, falls back to CSV if not found.

Parameters

file_path – Base path without extension.

Returns

Dictionary or Polars DataFrame containing the mapping.

```
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.initialize_model (args: Dict, verbose: int = 0)`
→ Tuple[dicee.models.base_model.BaseKGE, str]

Initialize a knowledge graph embedding model.

Parameters

- **args** – Dictionary containing model configuration including ‘model’ key.
- **verbose** – Verbosity level. If > 0, prints initialization message.

Returns

Tuple of (initialized model, form of labelling string).

Raises

ValueError – If the model name is not recognized.

`dicee.static_funcs.load_json (path: str) → Dict`

Load JSON file into a dictionary.

Parameters

path – Path to the JSON file.

Returns

Dictionary containing the JSON data.

Raises

- **FileNotFoundError** – If the file does not exist.

- `json.JSONDecodeError` – If the file contains invalid JSON.

`dicee.static_funcs.save_embeddings(embeddings: numpy.ndarray, indexes: List, path: str) → None`

Save embeddings to a CSV file.

Parameters

- **embeddings** – NumPy array of embeddings with shape (n_items, embedding_dim).
- **indexes** – List of index labels (entity/relation names).
- **path** – Output file path.

`dicee.static_funcs.vocab_to_parquet(vocab_to_idx, name, path_for_serialization, print_into)`

`dicee.static_funcs.create_experiment_folder(folder_name: str = 'Experiments') → str`

Create a timestamped experiment folder.

Parameters

folder_name – Base directory name for experiments.

Returns

Full path to the created experiment folder.

`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`

Training-related static functions.

This module provides backward compatibility by re-exporting evaluation functions from the new `dicee.evaluation` module, along with training utilities.

Deprecated since version Evaluation: functions have moved to `dicee.evaluation`. Use that module for new code. This module will continue to export training utilities.

Functions

<code>evaluate_lp</code> (\rightarrow Dict[str, float])	Evaluate link prediction with batched processing.
<code>evaluate_bpe_lp</code> (\rightarrow Dict[str, float])	Evaluate link prediction with BPE-encoded entities.
<code>make_iterable_verbose</code> (\rightarrow Iterable)	Wrap an iterable with tqdm progress bar if verbose is True.
<code>efficient_zero_grad</code> (\rightarrow None)	Efficiently zero gradients using parameter.grad = None.

Module Contents

`dicee.static_funcs_training.evaluate_lp`(*model*, *triple_idx*, *num_entities*: int, *er_vocab*: Dict[Tuple, List], *re_vocab*: Dict[Tuple, List], *info*: str = 'Eval Starts', *batch_size*: int = 128, *chunk_size*: int = 1000) \rightarrow Dict[str, float]

Evaluate link prediction with batched processing.

Memory-efficient evaluation using chunked entity scoring.

Parameters

- **model** – The KGE model to evaluate.
- **triple_idx** – Integer-indexed triples as numpy array.
- **num_entities** – Total number of entities.
- **er_vocab** – Mapping (head_idx, rel_idx) \rightarrow list of tail indices.
- **re_vocab** – Mapping (rel_idx, tail_idx) \rightarrow list of head indices.
- **info** – Description to print.
- **batch_size** – Batch size for triple processing.
- **chunk_size** – Chunk size for entity scoring.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

`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*: str = 'Eval Starts') \rightarrow Dict[str, float]

Evaluate link prediction with BPE-encoded entities.

Parameters

- **model** – The KGE model to evaluate.
- **triple_idx** – List of BPE-encoded triple tuples.
- **all_bpe_shaped_entities** – All entities with BPE representations.
- **er_vocab** – Mapping for tail filtering.
- **re_vocab** – Mapping for head filtering.
- **info** – Description to print.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

`dicee.static_funcs_training.make_iterable_verbose(iterable_object: Iterable, verbose: bool, desc: str = 'Default', position: int | None = None, leave: bool = True) → Iterable`

Wrap an iterable with tqdm progress bar if verbose is True.

Parameters

- **iterable_object** – The iterable to potentially wrap.
- **verbose** – Whether to show progress bar.
- **desc** – Description for the progress bar.
- **position** – Position of the progress bar.
- **leave** – Whether to leave the progress bar after completion.

Returns

The original iterable or a tqdm-wrapped version.

`dicee.static_funcs_training.efficient_zero_grad(model) → None`

Efficiently zero gradients using `parameter.grad = None`.

This is more efficient than `optimizer.zero_grad()` as it avoids memory operations.

See: https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Parameters

model – PyTorch model to zero gradients for.

`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`

DICE Trainer module for knowledge graph embedding training.

Provides the DICE_Trainer class which supports multiple training backends including PyTorch Lightning, DDP, and custom CPU/GPU trainers.

Classes

<code>DICE_Trainer</code>	DICE_Trainer implement
---------------------------	------------------------

Functions

<code>load_term_mapping(→ polars.DataFrame)</code>	Load term-to-index mapping from CSV file.
<code>initialize_trainer(...)</code>	Initialize the appropriate trainer based on configuration.
<code>get_callbacks(→ List)</code>	Create list of callbacks based on configuration.

Module Contents

`dicee.trainer.dice_trainer.load_term_mapping (file_path: str) → polars.DataFrame`

Load term-to-index mapping from CSV file.

Parameters

file_path – Base path without extension.

Returns

Polars DataFrame containing the mapping.

```
dicee.trainer.dice_trainer.initialize_trainer(args, callbacks: List)
```

→ *dicee.trainer.torch_trainer.TorchTrainer* | *dicee.trainer.model_parallelism.TensorParallel* | *dicee.trainer.torch_trainer_ddp*

Initialize the appropriate trainer based on configuration.

Parameters

- **args** – Configuration arguments containing trainer type.
- **callbacks** – List of training callbacks.

Returns

Initialized trainer instance.

Raises

AssertionError – If trainer is None after initialization.

```
dicee.trainer.dice_trainer.get_callbacks(args) → List
```

Create list of callbacks based on configuration.

Parameters

args – Configuration arguments.

Returns

List of callback instances.

```
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* | *dicее.trainer.model_parallelism.TensorParallel* | *dicее.trainer.torch_trainer.TorchTrainer* | *dicее.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: dicее.knowledge_graph.KG* | *numpy.memmap*)

→ *Tuple[dicее.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[dicее.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

dicее.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_idx

mapping.

relation_idx

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

<i>DICE_Trainer</i>	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` | `dicce.trainer.model_parallelism.TensorParallel` | `dicce.trainer.torch_trainer.TorchTrainer` | `dicce.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: dicce.knowledge_graph.KG* | *numpy.memmap*)

→ `Tuple[dicce.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[dicce.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

dicce.weight_averaging

Classes

<i>ASWA</i>	Adaptive stochastic weight averaging
<i>SWA</i>	Stochastic Weight Averaging callback.
<i>SWAG</i>	Stochastic Weight Averaging - Gaussian (SWAG).
<i>EMA</i>	Exponential Moving Average (EMA) callback.
<i>TWA</i>	Train with Weight Averaging (TWA) using subspace projection + averaging.

Module Contents

class `dicee.weight_averaging.ASWA` (*num_epochs*, *path*)

Bases: `dicee.abstracts.AbstractCallback`

Adaptive stochastic weight averaging ASWE keeps track of the validation performance and updates 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.weight_averaging.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 callback.
    Initialize SWA callback.

        swa_start_epoch: int
            The epoch at which to start SWA.

        swa_c_epochs: int
            The number of epochs to use for SWA.

        lr_init: float
            The initial learning rate.

        swa_lr: float
            The learning rate to use during SWA.

        max_epochs: int
            The maximum number of epochs. args.num_epochs

    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. Math: # SWA update: #  $\theta_{swa} \leftarrow (1 - \alpha) * \theta_{swa} + \alpha * \theta$  #  $\alpha = 1 / (n + 1)$ , where n = number of models already averaged # alpha is tracked via self.swa_n in code

    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.

class dicee.weight_averaging.SWAG(swa_start_epoch, swa_c_epochs: int = 1, lr_init: float = 0.1,
    swa_lr: float = 0.05, max_epochs: int = None, max_num_models: int = 20,
    var_clamp: float = 1e-30)
    Bases: dicee.abstracts.AbstractCallback
    Stochastic Weight Averaging - Gaussian (SWAG). Parameters

        swa_start_epoch
            [int] Epoch at which to start collecting weights.

```

swa_c_epochs
[int] Interval of epochs between updates.

lr_init
[float] Initial LR.

swa_lr
[float] LR in SWA / GSWA phase.

max_epochs
[int] Total number of epochs.

max_num_models
[int] Number of models to keep for low-rank covariance approx.

var_clamp
[float] Clamp low variance for stability.

swa_start_epoch

swa_c_epochs = 1

swa_lr = 0.05

lr_init = 0.1

max_epochs = None

max_num_models = 20

var_clamp = 1e-30

mean = None

sq_mean = None

deviations = []

gswa_n = 0

current_epoch = -1

get_mean_and_var()
Return mean + variance (diagonal part).

sample (*base_model*, *scale=0.5*)
Sample new model from SWAG posterior distribution.

Math: # From SWAG, posterior is approximated as: # $\theta \sim N(\text{mean}, \Sigma)$ # where $\Sigma \approx \text{diag}(\text{var}) + (1/(K-1)) * D D^T$ # - mean = running average of weights # - var = elementwise variance ($\text{sq_mean} - \text{mean}^2$) # - D = [$\text{dev}_1, \text{dev}_2, \dots, \text{dev}_K$], deviations from mean (low-rank approx) # - K = number of collected models

Sampling step: # 1. $\theta_{\text{diag}} = \text{mean} + \text{scale} * \text{std} \odot \varepsilon$, where $\varepsilon \sim N(0, I)$ # 2. $\theta_{\text{lowrank}} = \theta_{\text{diag}} + (D z) / \sqrt{K-1}$, where $z \sim N(0, I_K)$ # Final sample = θ_{lowrank}

on_train_epoch_start (*trainer*, *model*)
Update LR schedule (same as SWA).

on_train_epoch_end (*trainer*, *model*)
Collect Gaussian stats at the end of epochs after swa_start.

on_fit_end (*trainer, model*)

Set model weights to the collected SWAG mean at the end of training.

class dicee.weight_averaging.**EMA** (*ema_start_epoch: int, decay: float = 0.999,*
max_epochs: int = None, ema_c_epochs: int = 1)

Bases: *dicee.abstracts.AbstractCallback*

Exponential Moving Average (EMA) callback.

Parameters

- **ema_start_epoch** (*int*) – Epoch to start EMA.
- **decay** (*float*) – EMA decay rate (typical: 0.99 - 0.9999) Math: $\theta_{\text{ema}} \leftarrow \text{decay} * \theta_{\text{ema}} + (1 - \text{decay}) * \theta$
- **max_epochs** (*int*) – Maximum number of epochs.

ema_start_epoch

decay = 0.999

max_epochs = None

ema_c_epochs = 1

ema_model = None

current_epoch = -1

static ema_update (*ema_model, running_model, decay: float*)

Update EMA model with exponential moving average of current model. Math: # EMA update: $\theta_{\text{ema}} \leftarrow (1 - \alpha) * \theta_{\text{ema}} + \alpha * \theta$ # $\alpha = 1 - \text{decay}$, where decay is the EMA smoothing factor (typical 0.99 - 0.999) # α controls how much of the current model θ contributes to the EMA # decay is fixed in code -> can be extended to scheduled

on_train_epoch_start (*trainer, model*)

Track current epoch.

on_train_epoch_end (*trainer, model*)

Update EMA if past start epoch.

on_fit_end (*trainer, model*)

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

class dicee.weight_averaging.**TWA** (*twa_start_epoch: int, lr_init: float, num_samples: int = 5,*
reg_lambda: float = 0.0, max_epochs: int = None, twa_c_epochs: int = 1)

Bases: *dicee.abstracts.AbstractCallback*

Train with Weight Averaging (TWA) using subspace projection + averaging.

Parameters

twa_start_epoch

[int] Epoch to start TWA.

lr_init

[float] Learning rate used for β updates.

num_samples

[int] Number of sampled weight snapshots to build projection subspace.

reg_lambda
[float] Regularization coefficient for β updates.

max_epochs
[int] Total number of training epochs.

twa_c_epochs
[int] Interval of epochs between TWA updates.

twa_start_epoch

num_samples = 5

reg_lambda = 0.0

max_epochs = None

lr_init

twa_c_epochs = 1

current_epoch = -1

weight_samples = []

twa_model = None

base_weights = None

P = None

beta = None

sample_weights (*model*)
Collect sampled weights from the current model and maintain rolling buffer.

build_projection (*weight_samples*, *k=None*)
Build projection subspace from collected weight samples. :param weight_samples: list of flat weight tensors [(D,), ...] :param k: number of basis vectors to keep. Defaults to min(N, D).

Returns
(D,) base weight vector (average) P: (D, k) projection matrix with top-k basis directions

Return type
mean_w

on_train_epoch_start (*trainer*, *model*)
Track epoch.

on_train_epoch_end (*trainer*, *model*)
Main TWA logic: build subspace and update in β space.

Math: # TWA weight update: # $w_{\text{twa}} = \text{mean_w} + P * \beta$ # $\text{mean_w} = (1/n) * \sum_i w_i$ (SWA baseline)
$\beta \leftarrow (1 - \eta * \lambda) * \beta - \eta * P^T * g$ # g = gradient of training loss w.r.t. full model weights
η = learning rate, λ = ridge regularization # P = orthonormal basis spanning sampled checkpoints
{w_i}

on_fit_end (*trainer*, *model*)
Replace with TWA model at the end.

14.2 Attributes

`__version__`

14.3 Classes

<i>Execute</i>	Executor class for training, retraining and evaluating KGE models.
<i>KGE</i>	Knowledge Graph Embedding Class for interactive usage of pre-trained models
<i>QueryGenerator</i>	
<i>DICE_Trainer</i>	DICE_Trainer implement
<i>Evaluator</i>	Evaluator class for KGE models in various downstream tasks.

14.4 Package Contents

class `dicee.Execute` (*args*, *continuous_training: bool = False*)

Executor class for training, retraining and evaluating KGE models.

Handles the complete workflow: 1. Loading & Preprocessing & Serializing input data 2. Training & Validation & Testing 3. Storing all necessary information

args

Processed input arguments.

distributed

Whether distributed training is enabled.

rank

Process rank in distributed training.

world_size

Total number of processes.

local_rank

Local GPU rank.

trainer

Training handler instance.

trained_model

The trained model after training completes.

knowledge_graph

The loaded knowledge graph.

report

Dictionary storing training metrics and results.

evaluator

Model evaluation handler.

```

distributed

args

is_continual_training = False

trainer: dicce.trainer.DICE_Trainer | None = None

trained_model = None

knowledge_graph: dicce.knowledge_graph.KG | None = None

report: Dict

evaluator: dicce.evaluator.Evaluator | None = None

start_time: float | None = None

is_rank_zero() → bool

cleanup()

setup_executor() → None
    Set up storage directories for the experiment.
    Creates or reuses experiment directories based on configuration. Saves the configuration to a JSON file.

create_and_store_kg() → None
    Create knowledge graph and store as memory-mapped file.
    Only executed on rank 0 in distributed training. Skips if memmap already exists.

load_from_memmap() → None
    Load knowledge graph from memory-mapped file.

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.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.

$\text{argmax}_{\{e \in E\}} f(e, r, t)$, 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.

$\text{argmax}_{\{r \in R\}} f(h,r,t)$, 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

$\text{argmax}_{\{e \in E\}} f(h,r,e)$, 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 :

forall e in E and forall r in R f(e,r,x)

Return (e,r,x)

return 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 and (e,r,x)}

return G

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.**QueryGenerator** (*train_path: str, 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)

abstractmethod 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

```

```
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.trainer.cpu_trainer.CPUTrainer](#)

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.Evaluator(args, is_continual_training: bool = False)`

Evaluator class for KGE models in various downstream tasks.

Orchestrates link prediction evaluation with different scoring techniques including standard evaluation and byte-pair encoding based evaluation.

er_vocab

Entity-relation to tail vocabulary for filtered ranking.

re_vocab

Relation-entity (tail) to head vocabulary.

ee_vocab

Entity-entity to relation vocabulary.

num_entities

Total number of entities in the knowledge graph.

num_relations

Total number of relations in the knowledge graph.

args

Configuration arguments.

report

Dictionary storing evaluation results.

during_training

Whether evaluation is happening during training.

Example

```
>>> from dicee.evaluation import Evaluator
>>> evaluator = Evaluator(args)
>>> results = evaluator.eval(dataset, model, 'EntityPrediction')
>>> print(f"Test MRR: {results['Test']['MRR']:.4f}")
```

```

re_vocab: Dict | None = None
er_vocab: Dict | None = None
ee_vocab: Dict | None = None
func_triple_to_bpe_representation = None
is_continual_training = False
num_entities: int | None = None
num_relations: int | None = None
domain_constraints_per_rel = None
range_constraints_per_rel = None
args
report: Dict
during_training = False
vocab_preparation(dataset) → None
    Prepare vocabularies from the dataset for evaluation.
    Resolves any future objects and saves vocabularies to disk.

```

Parameters

dataset – Knowledge graph dataset with vocabulary attributes.

eval(*dataset*, *trained_model*, *form_of_labelling*: str, *during_training*: bool = False) → Dict | None
 Evaluate the trained model on the dataset.

Parameters

- **dataset** – Knowledge graph dataset (KG instance).
- **trained_model** – The trained KGE model.
- **form_of_labelling** – Type of labelling ('EntityPrediction' or 'RelationPrediction').
- **during_training** – Whether evaluation is during training.

Returns

Dictionary of evaluation metrics, or None if evaluation is skipped.

eval_rank_of_head_and_tail_entity(*, *train_set*, *valid_set*=None, *test_set*=None, *trained_model*)
 → None

Evaluate with negative sampling scoring.

eval_rank_of_head_and_tail_byte_pair_encoded_entity(*, *train_set*=None, *valid_set*=None, *test_set*=None, *ordered_bpe_entities*, *trained_model*) → None

Evaluate with BPE-encoded entities and negative sampling.

eval_with_byte(*, *raw_train_set*, *raw_valid_set*=None, *raw_test_set*=None, *trained_model*, *form_of_labelling*) → None

Evaluate ByteE model with generation.

eval_with_bpe_vs_all (*, raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model, form_of_labelling) → None

Evaluate with BPE and KvsAll scoring.

eval_with_vs_all (*, train_set, valid_set=None, test_set=None, trained_model, form_of_labelling) → None

Evaluate with KvsAll or lvsAll scoring.

evaluate_lp_k_vs_all (model, triple_idx, info: str | None = None, form_of_labelling: str | None = None) → Dict[str, float]

Filtered link prediction evaluation with KvsAll scoring.

Parameters

- **model** – The trained model to evaluate.
- **triple_idx** – Integer-indexed test triples.
- **info** – Description to print.
- **form_of_labelling** – ‘EntityPrediction’ or ‘RelationPrediction’.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

evaluate_lp_with_byte (model, triples: List[List[str]], info: str | None = None) → Dict[str, float]

Evaluate Byte model with text generation.

Parameters

- **model** – Byte model.
- **triples** – String triples.
- **info** – Description to print.

Returns

Dictionary with placeholder metrics (-1 values).

evaluate_lp_bpe_k_vs_all (model, triples: List[List[str]], info: str | None = None, form_of_labelling: str | None = None) → Dict[str, float]

Evaluate BPE model with KvsAll scoring.

Parameters

- **model** – BPE-enabled model.
- **triples** – String triples.
- **info** – Description to print.
- **form_of_labelling** – Type of labelling.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

evaluate_lp (model, triple_idx, info: str) → Dict[str, float]

Evaluate link prediction with negative sampling.

Parameters

- **model** – The model to evaluate.
- **triple_idx** – Integer-indexed triples.
- **info** – Description to print.

Returns

Dictionary with H@1, H@3, **H@10**, and MRR metrics.

dummy_eval (*trained_model*, *form_of_labelling*: *str*) → None

Run evaluation from saved data (for continual training).

Parameters

- **trained_model** – The trained model.
- **form_of_labelling** – Type of labelling.

eval_with_data (*dataset*, *trained_model*, *triple_idx*: *numpy.ndarray*, *form_of_labelling*: *str*)
→ Dict[str, float]

Evaluate a trained model on a given dataset.

Parameters

- **dataset** – Knowledge graph dataset.
- **trained_model** – The trained model.
- **triple_idx** – Integer-indexed triples to evaluate.
- **form_of_labelling** – Type of labelling.

Returns

Dictionary with evaluation metrics.

Raises

ValueError – If scoring technique is invalid.

`dicee.__version__ = '0.2.1'`

Python Module Index

d

- `dicee`, 12
- `dicee.__main__`, 12
- `dicee.abstracts`, 12
- `dicee.analyse_experiments`, 19
- `dicee.callbacks`, 21
- `dicee.config`, 28
- `dicee.dataset_classes`, 31
- `dicee.eval_static_funcs`, 45
- `dicee.evaluation`, 48
 - `dicee.evaluation.ensemble`, 48
 - `dicee.evaluation.evaluator`, 49
 - `dicee.evaluation.link_prediction`, 53
 - `dicee.evaluation.literal_prediction`, 56
 - `dicee.evaluation.utils`, 57
- `dicee.evaluator`, 67
- `dicee.executer`, 71
- `dicee.knowledge_graph`, 74
- `dicee.knowledge_graph_embeddings`, 76
- `dicee.models`, 80
 - `dicee.models.adopt`, 80
 - `dicee.models.base_model`, 89
 - `dicee.models.clifford`, 98
 - `dicee.models.complex`, 105
 - `dicee.models.dualE`, 107
 - `dicee.models.ensemble`, 109
 - `dicee.models.function_space`, 109
 - `dicee.models.literal`, 113
 - `dicee.models.octonion`, 114
 - `dicee.models.pykeen_models`, 118
 - `dicee.models.quaternion`, 119
 - `dicee.models.real`, 122
 - `dicee.models.static_funcs`, 125
 - `dicee.models.transformers`, 125
- `dicee.query_generator`, 184
- `dicee.read_preprocess_save_load_kg`, 186
- `dicee.read_preprocess_save_load_kg.preprocess`, 186
- `dicee.read_preprocess_save_load_kg.read_from_disk`, 187
- `dicee.read_preprocess_save_load_kg.save_load_disk`, 187
- `dicee.read_preprocess_save_load_kg.util`, 188
- `dicee.sanity_checkers`, 192
- `dicee.scripts`, 193
 - `dicee.scripts.index_serve`, 193
 - `dicee.scripts.run`, 194
- `dicee.static_funcs`, 195
- `dicee.static_funcs_training`, 199
- `dicee.static_preprocess_funcs`, 201
- `dicee.trainer`, 202
 - `dicee.trainer.dice_trainer`, 202
 - `dicee.trainer.model_parallelism`, 204
 - `dicee.trainer.torch_trainer`, 205
 - `dicee.trainer.torch_trainer_ddp`, 206
- `dicee.weight_averaging`, 209

Index

Non-alphabetical

`__call__()` (*dicee.models.base_model.IdentityClass method*), 98
`__call__()` (*dicee.models.ensemble.EnsembleKGE method*), 109
`__call__()` (*dicee.models.IdentityClass method*), 145, 159, 165
`__getitem__()` (*dicee.dataset_classes.AllvsAll method*), 35
`__getitem__()` (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 32
`__getitem__()` (*dicee.dataset_classes.KvsAll method*), 35
`__getitem__()` (*dicee.dataset_classes.KvsSampleDataset method*), 38
`__getitem__()` (*dicee.dataset_classes.LiteralDataset method*), 44
`__getitem__()` (*dicee.dataset_classes.MultiClassClassificationDataset method*), 33
`__getitem__()` (*dicee.dataset_classes.MultiLabelDataset method*), 33
`__getitem__()` (*dicee.dataset_classes.NegSampleDataset method*), 38
`__getitem__()` (*dicee.dataset_classes.OnevsAllDataset method*), 34
`__getitem__()` (*dicee.dataset_classes.OnevsSample method*), 37
`__getitem__()` (*dicee.dataset_classes.TriplePredictionDataset method*), 39
`__iter__()` (*dicee.config.Namespace method*), 31
`__iter__()` (*dicee.knowledge_graph.KG method*), 76
`__iter__()` (*dicee.models.ensemble.EnsembleKGE method*), 109
`__len__()` (*dicee.dataset_classes.AllvsAll method*), 35
`__len__()` (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 32
`__len__()` (*dicee.dataset_classes.KvsAll method*), 35
`__len__()` (*dicee.dataset_classes.KvsSampleDataset method*), 38
`__len__()` (*dicee.dataset_classes.LiteralDataset method*), 44
`__len__()` (*dicee.dataset_classes.MultiClassClassificationDataset method*), 33
`__len__()` (*dicee.dataset_classes.MultiLabelDataset method*), 33
`__len__()` (*dicee.dataset_classes.NegSampleDataset method*), 38
`__len__()` (*dicee.dataset_classes.OnevsAllDataset method*), 34
`__len__()` (*dicee.dataset_classes.OnevsSample method*), 37
`__len__()` (*dicee.dataset_classes.TriplePredictionDataset method*), 39
`__len__()` (*dicee.knowledge_graph.KG method*), 76
`__len__()` (*dicee.models.ensemble.EnsembleKGE method*), 109
`__setstate__()` (*dicee.models.ADOPT method*), 135
`__setstate__()` (*dicee.models.adopt.ADOPT method*), 83
`__str__()` (*dicee.KGE method*), 217
`__str__()` (*dicee.knowledge_graph_embeddings.KGE method*), 76
`__str__()` (*dicee.models.ensemble.EnsembleKGE method*), 109
`__version__` (in module *dicee*), 226

A

`AbstractCallback` (class in *dicee.abstracts*), 16
`AbstractPPECallback` (class in *dicee.abstracts*), 18
`AbstractTrainer` (class in *dicee.abstracts*), 13
`AccumulateEpochLossCallback` (class in *dicee.callbacks*), 21
`achieve_answer()` (*dicee.query_generator.QueryGenerator method*), 185
`achieve_answer()` (*dicee.QueryGenerator method*), 221
`AConEx` (class in *dicee.models*), 154
`AConEx` (class in *dicee.models.complex*), 106
`AConvO` (class in *dicee.models*), 167
`AConvO` (class in *dicee.models.octonion*), 117
`AConvQ` (class in *dicee.models*), 161
`AConvQ` (class in *dicee.models.quaternion*), 121
`adaptive_lr` (*dicee.config.Namespace attribute*), 31
`adaptive_swa` (*dicee.config.Namespace attribute*), 30
`add_new_entity_embeddings()` (*dicee.abstracts.BaseInteractiveKGE method*), 16
`add_noise_rate` (*dicee.config.Namespace attribute*), 28
`add_noise_rate` (*dicee.knowledge_graph.KG attribute*), 75
`add_noisy_triples()` (in module *dicee.static_funcs*), 198
`add_noisy_triples_into_training()` (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method*), 187
`add_noisy_triples_into_training()` (*dicee.read_preprocess_save_load_kg.ReadFromDisk method*), 192
`add_reciprocal` (*dicee.knowledge_graph.KG attribute*), 75
`ADOPT` (class in *dicee.models*), 133
`ADOPT` (class in *dicee.models.adopt*), 82
`adopt()` (in module *dicee.models.adopt*), 85
`ALL_HITS_RANGE` (in module *dicee.evaluation*), 67
`ALL_HITS_RANGE` (in module *dicee.evaluation.utils*), 57

AllvsAll (class in *dicее.dataset_classes*), 35
 alphas (*dicее.abstracts.AbstractPPECallback* attribute), 18
 alphas (*dicее.weight_averaging.ASWA* attribute), 210
 analyse () (in module *dicее.analyse_experiments*), 21
 answer_multi_hop_query () (*dicее.KGE* method), 219
 answer_multi_hop_query () (*dicее.knowledge_graph_embeddings.KGE* method), 79
 app (in module *dicее.scripts.index_serve*), 194
 apply_coefficients () (*dicее.models.clifford.DeCaL* method), 103
 apply_coefficients () (*dicее.models.clifford.Keci* method), 100
 apply_coefficients () (*dicее.models.DeCaL* method), 173
 apply_coefficients () (*dicее.models.Keci* method), 169
 apply_reciprocal_or_noise () (in module *dicее.read_preprocess_save_load_kg.util*), 190
 apply_semantic_constraint (*dicее.abstracts.BaseInteractiveKGE* attribute), 15
 apply_unit_norm (*dicее.models.base_model.BaseKGE* attribute), 95
 apply_unit_norm (*dicее.models.BaseKGE* attribute), 143, 146, 152, 156, 162, 175, 179
 args (*dicее.DICE_Trainer* attribute), 222
 args (*dicее.evaluation.Evaluator* attribute), 60
 args (*dicее.evaluation.evaluator.Evaluator* attribute), 50, 51
 args (*dicее.Evaluator* attribute), 223, 224
 args (*dicее.evaluator.Evaluator* attribute), 68, 69
 args (*dicее.Execute* attribute), 215, 216
 args (*dicее.executer.Execute* attribute), 71, 72
 args (*dicее.models.base_model.BaseKGE* attribute), 95
 args (*dicее.models.base_model.IdentityClass* attribute), 98
 args (*dicее.models.BaseKGE* attribute), 143, 146, 152, 156, 162, 175, 178
 args (*dicее.models.ensemble.EnsembleKGE* attribute), 109
 args (*dicее.models.IdentityClass* attribute), 145, 159, 165
 args (*dicее.models.pykeen_models.PykeenKGE* attribute), 118
 args (*dicее.models.PykeenKGE* attribute), 177
 args (*dicее.trainer.DICE_Trainer* attribute), 208
 args (*dicее.trainer.dice_trainer.DICE_Trainer* attribute), 203
 ASWA (class in *dicее.weight_averaging*), 210
 aswa (*dicее.analyse_experiments.Experiment* attribute), 20
 attn (*dicее.models.Block* attribute), 148
 attn (*dicее.models.transformers.Block* attribute), 130
 attn_dropout (*dicее.models.transformers.SelfAttention* attribute), 128
 attributes (*dicее.abstracts.AbstractTrainer* attribute), 13
 auto_batch_finding (*dicее.config.Namespace* attribute), 30

B

backend (*dicее.config.Namespace* attribute), 29
 backend (*dicее.knowledge_graph.KG* attribute), 75
 base_weights (*dicее.weight_averaging.TWA* attribute), 214
 BaseInteractiveKGE (class in *dicее.abstracts*), 14
 BaseInteractiveTrainKGE (class in *dicее.abstracts*), 18
 BaseKGE (class in *dicее.models*), 142, 145, 151, 156, 162, 174, 178
 BaseKGE (class in *dicее.models.base_model*), 94
 BaseKGELightning (class in *dicее.models*), 136
 BaseKGELightning (class in *dicее.models.base_model*), 89
 batch_kronecker_product () (*dicее.callbacks.KronE* static method), 25
 batch_size (*dicее.analyse_experiments.Experiment* attribute), 20
 batch_size (*dicее.callbacks.PseudoLabellingCallback* attribute), 24
 batch_size (*dicее.config.Namespace* attribute), 28
 batch_size (*dicее.dataset_classes.CVDataModule* attribute), 40
 batches_per_epoch (*dicее.callbacks.LRScheduler* attribute), 27
 beta (*dicее.weight_averaging.TWA* attribute), 214
 bias (*dicее.models.CoKEConfig* attribute), 150
 bias (*dicее.models.real.CoKEConfig* attribute), 124
 bias (*dicее.models.transformers.GPTConfig* attribute), 130
 bias (*dicее.models.transformers.LayerNorm* attribute), 127
 Block (class in *dicее.models*), 148
 Block (class in *dicее.models.transformers*), 129
 block_size (*dicее.config.Namespace* attribute), 30
 block_size (*dicее.dataset_classes.MultiClassClassificationDataset* attribute), 33
 block_size (*dicее.models.base_model.BaseKGE* attribute), 96
 block_size (*dicее.models.BaseKGE* attribute), 144, 147, 153, 157, 163, 176, 179
 block_size (*dicее.models.CoKEConfig* attribute), 150

block_size (*dicee.models.real.CoKEConfig* attribute), 123, 124
 block_size (*dicee.models.transformers.GPTConfig* attribute), 130
 blocks (*dicee.models.CoKE* attribute), 151
 blocks (*dicee.models.real.CoKE* attribute), 124
 bn_conv1 (*dicee.models.AConvQ* attribute), 161
 bn_conv1 (*dicee.models.ConvQ* attribute), 161
 bn_conv1 (*dicee.models.quaternion.AConvQ* attribute), 121
 bn_conv1 (*dicee.models.quaternion.ConvQ* attribute), 121
 bn_conv2 (*dicee.models.AConvQ* attribute), 161
 bn_conv2 (*dicee.models.ConvQ* attribute), 161
 bn_conv2 (*dicee.models.quaternion.AConvQ* attribute), 121
 bn_conv2 (*dicee.models.quaternion.ConvQ* attribute), 121
 bn_conv2d (*dicee.models.AConEx* attribute), 154
 bn_conv2d (*dicee.models.AConvO* attribute), 167
 bn_conv2d (*dicee.models.complex.AConEx* attribute), 106
 bn_conv2d (*dicee.models.complex.ConEx* attribute), 105
 bn_conv2d (*dicee.models.ConEx* attribute), 154
 bn_conv2d (*dicee.models.ConvO* attribute), 167
 bn_conv2d (*dicee.models.octonion.AConvO* attribute), 117
 bn_conv2d (*dicee.models.octonion.ConvO* attribute), 117
 BPE_NegativeSamplingDataset (*class in dicee.dataset_classes*), 32
 build_chain_funcs () (*dicee.models.FMult2* method), 182
 build_chain_funcs () (*dicee.models.function_space.FMult2* method), 111
 build_func () (*dicee.models.FMult2* method), 182
 build_func () (*dicee.models.function_space.FMult2* method), 111
 build_projection () (*dicee.weight_averaging.TWA* method), 214
 Byte (*class in dicee.models.transformers*), 125
 byte_pair_encoding (*dicee.analyse_experiments.Experiment* attribute), 20
 byte_pair_encoding (*dicee.config.Namespace* attribute), 30
 byte_pair_encoding (*dicee.knowledge_graph.KG* attribute), 75
 byte_pair_encoding (*dicee.models.base_model.BaseKGE* attribute), 96
 byte_pair_encoding (*dicee.models.BaseKGE* attribute), 143, 147, 152, 157, 163, 176, 179

C

c_attn (*dicee.models.transformers.SelfAttention* attribute), 128
 c_fc (*dicee.models.transformers.MLP* attribute), 129
 c_proj (*dicee.models.transformers.MLP* attribute), 129
 c_proj (*dicee.models.transformers.SelfAttention* attribute), 128
 callbacks (*dicee.abstracts.AbstractTrainer* attribute), 13
 callbacks (*dicee.analyse_experiments.Experiment* attribute), 20
 callbacks (*dicee.config.Namespace* attribute), 29
 callbacks (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 207
 causal (*dicee.models.CoKEConfig* attribute), 150
 causal (*dicee.models.real.CoKEConfig* attribute), 124
 causal (*dicee.models.transformers.GPTConfig* attribute), 130
 causal (*dicee.models.transformers.SelfAttention* attribute), 128
 chain_func () (*dicee.models.FMult* method), 181
 chain_func () (*dicee.models.function_space.FMult* method), 110
 chain_func () (*dicee.models.function_space.GFMult* method), 111
 chain_func () (*dicee.models.GFMult* method), 181
 CKeci (*class in dicee.models*), 171
 CKeci (*class in dicee.models.clifford*), 101
 cl_pqr () (*dicee.models.clifford.DeCaL* method), 102
 cl_pqr () (*dicee.models.DeCaL* method), 172
 cleanup () (*dicee.Execute* method), 216
 cleanup () (*dicee.executer.Execute* method), 72
 clifford_multiplication () (*dicee.models.clifford.Keci* method), 100
 clifford_multiplication () (*dicee.models.Keci* method), 169
 clip_lambda (*dicee.models.ADOPT* attribute), 135
 clip_lambda (*dicee.models.adopt.ADOPT* attribute), 83
 CoKE (*class in dicee.models*), 150
 CoKE (*class in dicee.models.real*), 124
 coke_dropout (*dicee.models.CoKE* attribute), 151
 coke_dropout (*dicee.models.real.CoKE* attribute), 124
 CoKEConfig (*class in dicee.models*), 150
 CoKEConfig (*class in dicee.models.real*), 123
 collate_fn (*dicee.dataset_classes.AllvsAll* attribute), 35

`collate_fn` (*dicee.dataset_classes.KvsAll* attribute), 35
`collate_fn` (*dicee.dataset_classes.KvsSampleDataset* attribute), 38
`collate_fn` (*dicee.dataset_classes.MultiClassClassificationDataset* attribute), 33
`collate_fn` (*dicee.dataset_classes.MultiLabelDataset* attribute), 33
`collate_fn` (*dicee.dataset_classes.OnevsAllDataset* attribute), 34
`collate_fn` (*dicee.dataset_classes.OnevsSample* attribute), 36, 37
`collate_fn` () (*dicee.dataset_classes.BPE_NegativeSamplingDataset* method), 32
`collate_fn` () (*dicee.dataset_classes.TriplePredictionDataset* method), 39
`collection_name` (*dicee.scripts.index_serve.NeuralSearcher* attribute), 194
`comp_func` () (*dicee.models.function_space.LFMult* method), 113
`comp_func` () (*dicee.models.LFMult* method), 183
`ComplEx` (class in *dicee.models*), 155
`ComplEx` (class in *dicee.models.complex*), 106
`compute_convergence` () (in module *dicee.callbacks*), 24
`compute_func` () (*dicee.models.FMult* method), 181
`compute_func` () (*dicee.models.FMult2* method), 182
`compute_func` () (*dicee.models.function_space.FMult* method), 110
`compute_func` () (*dicee.models.function_space.FMult2* method), 111
`compute_func` () (*dicee.models.function_space.GFMult* method), 110
`compute_func` () (*dicee.models.GFMult* method), 181
`compute_metrics_from_ranks` () (in module *dicee.evaluation*), 66
`compute_metrics_from_ranks` () (in module *dicee.evaluation.utils*), 58
`compute_metrics_from_ranks_simple` () (in module *dicee.evaluation*), 66
`compute_metrics_from_ranks_simple` () (in module *dicee.evaluation.utils*), 58
`compute_mrr` () (*dicee.weight_averaging.ASWA* static method), 210
`compute_sigma_pp` () (*dicee.models.clifford.DeCaL* method), 103
`compute_sigma_pp` () (*dicee.models.clifford.Keci* method), 99
`compute_sigma_pp` () (*dicee.models.DeCaL* method), 173
`compute_sigma_pp` () (*dicee.models.Keci* method), 168
`compute_sigma_pq` () (*dicee.models.clifford.DeCaL* method), 104
`compute_sigma_pq` () (*dicee.models.clifford.Keci* method), 99
`compute_sigma_pq` () (*dicee.models.DeCaL* method), 174
`compute_sigma_pq` () (*dicee.models.Keci* method), 169
`compute_sigma_pr` () (*dicee.models.clifford.DeCaL* method), 105
`compute_sigma_pr` () (*dicee.models.DeCaL* method), 174
`compute_sigma_qq` () (*dicee.models.clifford.DeCaL* method), 104
`compute_sigma_qq` () (*dicee.models.clifford.Keci* method), 99
`compute_sigma_qq` () (*dicee.models.DeCaL* method), 173
`compute_sigma_qq` () (*dicee.models.Keci* method), 169
`compute_sigma_qr` () (*dicee.models.clifford.DeCaL* method), 105
`compute_sigma_qr` () (*dicee.models.DeCaL* method), 174
`compute_sigma_rr` () (*dicee.models.clifford.DeCaL* method), 104
`compute_sigma_rr` () (*dicee.models.DeCaL* method), 174
`compute_sigmas_multivect` () (*dicee.models.clifford.DeCaL* method), 103
`compute_sigmas_multivect` () (*dicee.models.DeCaL* method), 172
`compute_sigmas_single` () (*dicee.models.clifford.DeCaL* method), 102
`compute_sigmas_single` () (*dicee.models.DeCaL* method), 172
`ConEx` (class in *dicee.models*), 153
`ConEx` (class in *dicee.models.complex*), 105
`config` (*dicee.models.CoKE* attribute), 151
`config` (*dicee.models.real.CoKE* attribute), 124
`config` (*dicee.models.transformers.BytE* attribute), 126
`config` (*dicee.models.transformers.GPT* attribute), 131
`configs` (*dicee.abstracts.BaseInteractiveKGE* attribute), 15
`configure_optimizers` () (*dicee.models.base_model.BaseKGELightning* method), 93
`configure_optimizers` () (*dicee.models.BaseKGELightning* method), 141
`configure_optimizers` () (*dicee.models.transformers.GPT* method), 131
`construct_batch_selected_cl_multivector` () (*dicee.models.clifford.Keci* method), 100
`construct_batch_selected_cl_multivector` () (*dicee.models.Keci* method), 170
`construct_cl_multivector` () (*dicee.models.clifford.DeCaL* method), 103
`construct_cl_multivector` () (*dicee.models.clifford.Keci* method), 100
`construct_cl_multivector` () (*dicee.models.DeCaL* method), 173
`construct_cl_multivector` () (*dicee.models.Keci* method), 170
`construct_dataset` () (in module *dicee.dataset_classes*), 32
`construct_ensemble` (*dicee.abstracts.BaseInteractiveKGE* attribute), 14
`construct_graph` () (*dicee.query_generator.QueryGenerator* method), 185
`construct_graph` () (*dicee.QueryGenerator* method), 221
`construct_input_and_output` () (*dicee.abstracts.BaseInteractiveKGE* method), 16

`construct_multi_coeff()` (*dicee.models.function_space.LFMMult method*), 112
`construct_multi_coeff()` (*dicee.models.LFMMult method*), 183
`continual_learning` (*dicee.config.Namespace attribute*), 30
`continual_start()` (*dicee.DICE_Trainer method*), 222
`continual_start()` (*dicee.executer.ContinuousExecute method*), 73
`continual_start()` (*dicee.trainer.DICE_Trainer method*), 208
`continual_start()` (*dicee.trainer.dice_trainer.DICE_Trainer method*), 203
`continual_training_setup_executor()` (*in module dicee.static_funcs*), 199
`ContinuousExecute` (*class in dicee.executer*), 73
`conv2d` (*dicee.models.AConEx attribute*), 154
`conv2d` (*dicee.models.AConvO attribute*), 167
`conv2d` (*dicee.models.AConvQ attribute*), 161
`conv2d` (*dicee.models.complex.AConEx attribute*), 106
`conv2d` (*dicee.models.complex.ConEx attribute*), 105
`conv2d` (*dicee.models.ConEx attribute*), 154
`conv2d` (*dicee.models.ConvO attribute*), 167
`conv2d` (*dicee.models.ConvQ attribute*), 161
`conv2d` (*dicee.models.octonion.AConvO attribute*), 117
`conv2d` (*dicee.models.octonion.ConvO attribute*), 116
`conv2d` (*dicee.models.quaternion.AConvQ attribute*), 121
`conv2d` (*dicee.models.quaternion.ConvQ attribute*), 121
`ConvO` (*class in dicee.models*), 166
`ConvO` (*class in dicee.models.octonion*), 116
`ConvQ` (*class in dicee.models*), 160
`ConvQ` (*class in dicee.models.quaternion*), 121
`count_triples()` (*in module dicee.read_preprocess_save_load_kg.util*), 190
`create_and_store_kg()` (*dicee.Execute method*), 216
`create_and_store_kg()` (*dicee.executer.Execute method*), 72
`create_constraints()` (*in module dicee.read_preprocess_save_load_kg.util*), 190
`create_constraints()` (*in module dicee.static_preprocess_funcs*), 202
`create_experiment_folder()` (*in module dicee.static_funcs*), 199
`create_hits_dict()` (*in module dicee.evaluation*), 67
`create_hits_dict()` (*in module dicee.evaluation.utils*), 58
`create_random_data()` (*dicee.callbacks.PseudoLabellingCallback method*), 24
`create_recipriocal_triples()` (*in module dicee.read_preprocess_save_load_kg.util*), 191
`create_recipriocal_triples()` (*in module dicee.static_funcs*), 196
`create_vector_database()` (*dicee.KGE method*), 217
`create_vector_database()` (*dicee.knowledge_graph_embeddings.KGE method*), 77
`crop_block_size()` (*dicee.models.transformers.GPT method*), 131
`ctx` (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 207
`current_epoch` (*dicee.weight_averaging.EMA attribute*), 213
`current_epoch` (*dicee.weight_averaging.SWA attribute*), 211
`current_epoch` (*dicee.weight_averaging.SWAG attribute*), 212
`current_epoch` (*dicee.weight_averaging.TWA attribute*), 214
`CVDataModule` (*class in dicee.dataset_classes*), 39
`cycle_length` (*dicee.callbacks.LRScheduler attribute*), 27

D

`data_module` (*dicee.callbacks.PseudoLabellingCallback attribute*), 24
`data_property_embeddings` (*dicee.models.literal.LiteralEmbeddings attribute*), 114
`data_property_to_idx` (*dicee.dataset_classes.LiteralDataset attribute*), 44
`dataset_dir` (*dicee.config.Namespace attribute*), 28
`dataset_dir` (*dicee.knowledge_graph.KG attribute*), 74
`dataset_sanity_checking()` (*in module dicee.read_preprocess_save_load_kg.util*), 191
`DeCaL` (*class in dicee.models*), 171
`DeCaL` (*class in dicee.models.clifford*), 101
`decay` (*dicee.weight_averaging.EMA attribute*), 213
`decide()` (*dicee.weight_averaging.ASWA method*), 210
`default_eval_model` (*dicee.callbacks.PeriodicEvalCallback attribute*), 27
`DEFAULT_HITS_RANGE` (*in module dicee.evaluation*), 67
`DEFAULT_HITS_RANGE` (*in module dicee.evaluation.utils*), 57
`degree` (*dicee.models.function_space.LFMMult attribute*), 112
`degree` (*dicee.models.LFMMult attribute*), 182
`denormalize()` (*dicee.dataset_classes.LiteralDataset static method*), 44
`describe()` (*dicee.knowledge_graph.KG method*), 76
`description_of_input` (*dicee.knowledge_graph.KG attribute*), 76
`deviations` (*dicee.weight_averaging.SWAG attribute*), 212

- device (*dicee.models.literal.LiteralEmbeddings* property), 114
- DICE_Trainer (class in *dicee*), 221
- DICE_Trainer (class in *dicee.trainer*), 208
- DICE_Trainer (class in *dicee.trainer.dice_trainer*), 203
- dicee
 - module, 12
- dicee.__main__
 - module, 12
- dicee.abstracts
 - module, 12
- dicee.analyse_experiments
 - module, 19
- dicee.callbacks
 - module, 21
- dicee.config
 - module, 28
- dicee.dataset_classes
 - module, 31
- dicee.eval_static_funcs
 - module, 45
- dicee.evaluation
 - module, 48
- dicee.evaluation.ensemble
 - module, 48
- dicee.evaluation.evaluator
 - module, 49
- dicee.evaluation.link_prediction
 - module, 53
- dicee.evaluation.literal_prediction
 - module, 56
- dicee.evaluation.utils
 - module, 57
- dicee.evaluator
 - module, 67
- dicee.executer
 - module, 71
- dicee.knowledge_graph
 - module, 74
- dicee.knowledge_graph_embeddings
 - module, 76
- dicee.models
 - module, 80
- dicee.models.adopt
 - module, 80
- dicee.models.base_model
 - module, 89
- dicee.models.clifford
 - module, 98
- dicee.models.complex
 - module, 105
- dicee.models.dualE
 - module, 107
- dicee.models.ensemble
 - module, 109
- dicee.models.function_space
 - module, 109
- dicee.models.literal
 - module, 113
- dicee.models.octonion
 - module, 114
- dicee.models.pykeen_models
 - module, 118
- dicee.models.quaternion
 - module, 119
- dicee.models.real
 - module, 122
- dicee.models.static_funcs
 - module, 125

- `dicee.models.transformers`
 - module, 125
- `dicee.query_generator`
 - module, 184
- `dicee.read_preprocess_save_load_kg`
 - module, 186
- `dicee.read_preprocess_save_load_kg.preprocess`
 - module, 186
- `dicee.read_preprocess_save_load_kg.read_from_disk`
 - module, 187
- `dicee.read_preprocess_save_load_kg.save_load_disk`
 - module, 187
- `dicee.read_preprocess_save_load_kg.util`
 - module, 188
- `dicee.sanity_checkers`
 - module, 192
- `dicee.scripts`
 - module, 193
- `dicee.scripts.index_serve`
 - module, 193
- `dicee.scripts.run`
 - module, 194
- `dicee.static_funcs`
 - module, 195
- `dicee.static_funcs_training`
 - module, 199
- `dicee.static_preprocess_funcs`
 - module, 201
- `dicee.trainer`
 - module, 202
- `dicee.trainer.dice_trainer`
 - module, 202
- `dicee.trainer.model_parallelism`
 - module, 204
- `dicee.trainer.torch_trainer`
 - module, 205
- `dicee.trainer.torch_trainer_ddp`
 - module, 206
- `dicee.weight_averaging`
 - module, 209
- `discrete_points` (*dicee.models.FMult2 attribute*), 181
- `discrete_points` (*dicee.models.function_space.FMult2 attribute*), 111
- `dist_func` (*dicee.models.Pyke attribute*), 150
- `dist_func` (*dicee.models.real.Pyke attribute*), 123
- `DistMult` (*class in dicee.models*), 149
- `DistMult` (*class in dicee.models.real*), 122
- `distributed` (*dicee.Execute attribute*), 215
- `distributed` (*dicee.executer.Execute attribute*), 71, 72
- `domain_constraints_per_rel` (*dicee.evaluation.Evaluator attribute*), 60
- `domain_constraints_per_rel` (*dicee.evaluation.evaluator.Evaluator attribute*), 51
- `domain_constraints_per_rel` (*dicee.Evaluator attribute*), 224
- `domain_constraints_per_rel` (*dicee.evaluator.Evaluator attribute*), 69
- `download_file()` (*in module dicee.static_funcs*), 199
- `download_files_from_url()` (*in module dicee.static_funcs*), 199
- `download_pretrained_model()` (*in module dicee.static_funcs*), 199
- `dropout` (*dicee.models.CoKEConfig attribute*), 150
- `dropout` (*dicee.models.literal.LiteralEmbeddings attribute*), 113, 114
- `dropout` (*dicee.models.real.CoKEConfig attribute*), 124
- `dropout` (*dicee.models.transformers.GPTConfig attribute*), 130
- `dropout` (*dicee.models.transformers.MLP attribute*), 129
- `dropout` (*dicee.models.transformers.SelfAttention attribute*), 128
- `DualE` (*class in dicee.models*), 183
- `DualE` (*class in dicee.models.dualE*), 108
- `dummy_eval()` (*dicee.evaluation.Evaluator method*), 62
- `dummy_eval()` (*dicee.evaluation.evaluator.Evaluator method*), 52
- `dummy_eval()` (*dicee.Evaluator method*), 226
- `dummy_eval()` (*dicee.evaluator.Evaluator method*), 70
- `dummy_id` (*dicee.knowledge_graph.KG attribute*), 75

during_training (*dicee.evaluation.Evaluator* attribute), 60
 during_training (*dicee.evaluation.evaluator.Evaluator* attribute), 50, 51
 during_training (*dicee.Evaluator* attribute), 223, 224
 during_training (*dicee.evaluator.Evaluator* attribute), 68, 69

E

ee_vocab (*dicee.evaluation.Evaluator* attribute), 60
 ee_vocab (*dicee.evaluation.evaluator.Evaluator* attribute), 50, 51
 ee_vocab (*dicee.Evaluator* attribute), 223, 224
 ee_vocab (*dicee.evaluator.Evaluator* attribute), 68
 efficient_zero_grad() (in module *dicee.evaluation.utils*), 58
 efficient_zero_grad() (in module *dicee.static_funcs_training*), 201
 EMA (class in *dicee.weight_averaging*), 213
 ema (*dicee.config.Namespace* attribute), 30
 ema_c_epochs (*dicee.weight_averaging.EMA* attribute), 213
 ema_model (*dicee.weight_averaging.EMA* attribute), 213
 ema_start_epoch (*dicee.weight_averaging.EMA* attribute), 213
 ema_update() (*dicee.weight_averaging.EMA* static method), 213
 embedding_dim (*dicee.analyse_experiments.Experiment* attribute), 20
 embedding_dim (*dicee.config.Namespace* attribute), 28
 embedding_dim (*dicee.models.base_model.BaseKGE* attribute), 95
 embedding_dim (*dicee.models.BaseKGE* attribute), 143, 146, 152, 156, 162, 175, 179
 embedding_dim (*dicee.models.literal.LiteralEmbeddings* attribute), 114
 embedding_dims (*dicee.models.literal.LiteralEmbeddings* attribute), 113
 enable_log (in module *dicee.static_preprocess_funcs*), 202
 enc (*dicee.knowledge_graph.KG* attribute), 75
 end() (*dicee.Execute* method), 216
 end() (*dicee.executer.Execute* method), 73
 EnsembleKGE (class in *dicee.models.ensemble*), 109
 ent2id (*dicee.query_generator.QueryGenerator* attribute), 185
 ent2id (*dicee.QueryGenerator* attribute), 221
 ent_in (*dicee.query_generator.QueryGenerator* attribute), 185
 ent_in (*dicee.QueryGenerator* attribute), 221
 ent_out (*dicee.query_generator.QueryGenerator* attribute), 185
 ent_out (*dicee.QueryGenerator* attribute), 221
 entities_str (*dicee.knowledge_graph.KG* property), 76
 entity_embeddings (*dicee.models.AConvQ* attribute), 161
 entity_embeddings (*dicee.models.clifford.DeCaL* attribute), 102
 entity_embeddings (*dicee.models.ConvQ* attribute), 161
 entity_embeddings (*dicee.models.DeCaL* attribute), 171
 entity_embeddings (*dicee.models.DualE* attribute), 184
 entity_embeddings (*dicee.models.dualE.DualE* attribute), 108
 entity_embeddings (*dicee.models.FMult* attribute), 180
 entity_embeddings (*dicee.models.FMult2* attribute), 181
 entity_embeddings (*dicee.models.function_space.FMult* attribute), 110
 entity_embeddings (*dicee.models.function_space.FMult2* attribute), 111
 entity_embeddings (*dicee.models.function_space.GFMult* attribute), 110
 entity_embeddings (*dicee.models.function_space.LFMult* attribute), 112
 entity_embeddings (*dicee.models.function_space.LFMult1* attribute), 111
 entity_embeddings (*dicee.models.GFMult* attribute), 181
 entity_embeddings (*dicee.models.LFMult* attribute), 182
 entity_embeddings (*dicee.models.LFMult1* attribute), 182
 entity_embeddings (*dicee.models.literal.LiteralEmbeddings* attribute), 113, 114
 entity_embeddings (*dicee.models.pykeen_models.PykeenKGE* attribute), 118
 entity_embeddings (*dicee.models.PykeenKGE* attribute), 177
 entity_embeddings (*dicee.models.quaternion.AConvQ* attribute), 121
 entity_embeddings (*dicee.models.quaternion.ConvQ* attribute), 121
 entity_to_idx (*dicee.dataset_classes.LiteralDataset* attribute), 43, 44
 entity_to_idx (*dicee.knowledge_graph.KG* attribute), 74, 75
 entity_to_idx (*dicee.scripts.index_serve.NeuralSearcher* attribute), 194
 epoch_count (*dicee.abstracts.AbstractPPECallback* attribute), 18
 epoch_count (*dicee.weight_averaging.ASWA* attribute), 210
 epoch_counter (*dicee.callbacks.Eval* attribute), 24
 epoch_counter (*dicee.callbacks.KGESaveCallback* attribute), 23
 epoch_counter (*dicee.callbacks.PeriodicEvalCallback* attribute), 26
 epoch_ratio (*dicee.callbacks.Eval* attribute), 24
 er_vocab (*dicee.evaluation.Evaluator* attribute), 59, 60

er_vocab (*dictee.evaluation.evaluator.Evaluator attribute*), 50
 er_vocab (*dictee.Evaluator attribute*), 223, 224
 er_vocab (*dictee.evaluator.Evaluator attribute*), 68
 estimate_mfu () (*dictee.models.transformers.GPT method*), 131
 estimate_q () (*in module dictee.callbacks*), 24
 Eval (*class in dictee.callbacks*), 24
 eval () (*dictee.evaluation.Evaluator method*), 61
 eval () (*dictee.evaluation.evaluator.Evaluator method*), 51
 eval () (*dictee.Evaluator method*), 224
 eval () (*dictee.evaluator.Evaluator method*), 69
 eval () (*dictee.models.ensemble.EnsembleKGE method*), 109
 eval_at_epochs (*dictee.config.Namespace attribute*), 31
 eval_epochs (*dictee.callbacks.PeriodicEvalCallback attribute*), 27
 eval_every_n_epochs (*dictee.config.Namespace attribute*), 31
 eval_lp_performance () (*dictee.KGE method*), 217
 eval_lp_performance () (*dictee.knowledge_graph_embeddings.KGE method*), 77
 eval_model (*dictee.config.Namespace attribute*), 29
 eval_model (*dictee.knowledge_graph.KG attribute*), 75
 eval_rank_of_head_and_tail_byte_pair_encoded_entity () (*dictee.evaluation.Evaluator method*), 61
 eval_rank_of_head_and_tail_byte_pair_encoded_entity () (*dictee.evaluation.evaluator.Evaluator method*), 51
 eval_rank_of_head_and_tail_byte_pair_encoded_entity () (*dictee.Evaluator method*), 224
 eval_rank_of_head_and_tail_byte_pair_encoded_entity () (*dictee.evaluator.Evaluator method*), 69
 eval_rank_of_head_and_tail_entity () (*dictee.evaluation.Evaluator method*), 61
 eval_rank_of_head_and_tail_entity () (*dictee.evaluation.evaluator.Evaluator method*), 51
 eval_rank_of_head_and_tail_entity () (*dictee.Evaluator method*), 224
 eval_rank_of_head_and_tail_entity () (*dictee.evaluator.Evaluator method*), 69
 eval_with_bpe_vs_all () (*dictee.evaluation.Evaluator method*), 61
 eval_with_bpe_vs_all () (*dictee.evaluation.evaluator.Evaluator method*), 51
 eval_with_bpe_vs_all () (*dictee.Evaluator method*), 224
 eval_with_bpe_vs_all () (*dictee.evaluator.Evaluator method*), 69
 eval_with_byte () (*dictee.evaluation.Evaluator method*), 61
 eval_with_byte () (*dictee.evaluation.evaluator.Evaluator method*), 51
 eval_with_byte () (*dictee.Evaluator method*), 224
 eval_with_byte () (*dictee.evaluator.Evaluator method*), 69
 eval_with_data () (*dictee.evaluation.Evaluator method*), 62
 eval_with_data () (*dictee.evaluation.evaluator.Evaluator method*), 53
 eval_with_data () (*dictee.Evaluator method*), 226
 eval_with_data () (*dictee.evaluator.Evaluator method*), 71
 eval_with_vs_all () (*dictee.evaluation.Evaluator method*), 61
 eval_with_vs_all () (*dictee.evaluation.evaluator.Evaluator method*), 51
 eval_with_vs_all () (*dictee.Evaluator method*), 225
 eval_with_vs_all () (*dictee.evaluator.Evaluator method*), 69
 evaluate () (*in module dictee.static_funcs*), 199
 evaluate_bpe_lp () (*in module dictee.evaluation*), 64
 evaluate_bpe_lp () (*in module dictee.evaluation.link_prediction*), 55
 evaluate_bpe_lp () (*in module dictee.static_funcs_training*), 200
 evaluate_ensemble_link_prediction_performance () (*in module dictee.eval_static_funcs*), 47
 evaluate_ensemble_link_prediction_performance () (*in module dictee.evaluation*), 65
 evaluate_ensemble_link_prediction_performance () (*in module dictee.evaluation.ensemble*), 49
 evaluate_link_prediction_performance () (*in module dictee.eval_static_funcs*), 45
 evaluate_link_prediction_performance () (*in module dictee.evaluation*), 63
 evaluate_link_prediction_performance () (*in module dictee.evaluation.link_prediction*), 53
 evaluate_link_prediction_performance_with_bpe () (*in module dictee.eval_static_funcs*), 46
 evaluate_link_prediction_performance_with_bpe () (*in module dictee.evaluation*), 63
 evaluate_link_prediction_performance_with_bpe () (*in module dictee.evaluation.link_prediction*), 54
 evaluate_link_prediction_performance_with_bpe_reciprocals () (*in module dictee.eval_static_funcs*), 46
 evaluate_link_prediction_performance_with_bpe_reciprocals () (*in module dictee.evaluation*), 63
 evaluate_link_prediction_performance_with_bpe_reciprocals () (*in module dictee.evaluation.link_prediction*), 54
 evaluate_link_prediction_performance_with_reciprocals () (*in module dictee.eval_static_funcs*), 45
 evaluate_link_prediction_performance_with_reciprocals () (*in module dictee.evaluation*), 63
 evaluate_link_prediction_performance_with_reciprocals () (*in module dictee.evaluation.link_prediction*), 54
 evaluate_literal_prediction () (*in module dictee.eval_static_funcs*), 47
 evaluate_literal_prediction () (*in module dictee.evaluation*), 65
 evaluate_literal_prediction () (*in module dictee.evaluation.literal_prediction*), 56
 evaluate_lp () (*dictee.evaluation.Evaluator method*), 62
 evaluate_lp () (*dictee.evaluation.evaluator.Evaluator method*), 52
 evaluate_lp () (*dictee.Evaluator method*), 225
 evaluate_lp () (*dictee.evaluator.Evaluator method*), 70

evaluate_lp() (in module *dicee.evaluation*), 64
 evaluate_lp() (in module *dicee.evaluation.link_prediction*), 54
 evaluate_lp() (in module *dicee.static_funcs_training*), 200
 evaluate_lp_bpe_k_vs_all() (*dicee.evaluation.Evaluator* method), 62
 evaluate_lp_bpe_k_vs_all() (*dicee.evaluation.evaluator.Evaluator* method), 52
 evaluate_lp_bpe_k_vs_all() (*dicee.Evaluator* method), 225
 evaluate_lp_bpe_k_vs_all() (*dicee.evaluator.Evaluator* method), 70
 evaluate_lp_bpe_k_vs_all() (in module *dicee.eval_static_funcs*), 46
 evaluate_lp_bpe_k_vs_all() (in module *dicee.evaluation*), 64
 evaluate_lp_bpe_k_vs_all() (in module *dicee.evaluation.link_prediction*), 55
 evaluate_lp_k_vs_all() (*dicee.evaluation.Evaluator* method), 61
 evaluate_lp_k_vs_all() (*dicee.evaluation.evaluator.Evaluator* method), 52
 evaluate_lp_k_vs_all() (*dicee.Evaluator* method), 225
 evaluate_lp_k_vs_all() (*dicee.evaluator.Evaluator* method), 69
 evaluate_lp_with_byte() (*dicee.evaluation.Evaluator* method), 61
 evaluate_lp_with_byte() (*dicee.evaluation.evaluator.Evaluator* method), 52
 evaluate_lp_with_byte() (*dicee.Evaluator* method), 225
 evaluate_lp_with_byte() (*dicee.evaluator.Evaluator* method), 70
 Evaluator (class in *dicee*), 223
 Evaluator (class in *dicee.evaluation*), 59
 Evaluator (class in *dicee.evaluation.evaluator*), 50
 Evaluator (class in *dicee.evaluator*), 68
 evaluator (*dicee.DICE_Trainer* attribute), 222
 evaluator (*dicee.Execute* attribute), 215, 216
 evaluator (*dicee.executer.Execute* attribute), 72
 evaluator (*dicee.trainer.DICE_Trainer* attribute), 208
 evaluator (*dicee.trainer.dice_trainer.DICE_Trainer* attribute), 203
 every_x_epoch (*dicee.callbacks.KGESaveCallback* attribute), 23
 example_input_array (*dicee.models.ensemble.EnsembleKGE* property), 109
 Execute (class in *dicee*), 215
 Execute (class in *dicee.executer*), 71
 exists() (*dicee.knowledge_graph.KG* method), 76
 Experiment (class in *dicee.analyse_experiments*), 20
 experiment_dir (*dicee.callbacks.LRScheduler* attribute), 27
 experiment_dir (*dicee.callbacks.PeriodicEvalCallback* attribute), 26
 explicit (*dicee.models.QMult* attribute), 160
 explicit (*dicee.models.quaternion.QMult* attribute), 120
 exponential_function() (in module *dicee.static_funcs*), 199
 extract_input_outputs() (*dicee.trainer.torch_trainer_ddp.NodeTrainer* method), 208
 extract_input_outputs() (in module *dicee.trainer.model_parallelism*), 205
 extract_input_outputs_set_device() (*dicee.trainer.torch_trainer.TorchTrainer* method), 206

F

f (*dicee.callbacks.KronE* attribute), 25
 fc (*dicee.models.literal.LiteralEmbeddings* attribute), 114
 fc1 (*dicee.models.AConEx* attribute), 154
 fc1 (*dicee.models.AConvO* attribute), 167
 fc1 (*dicee.models.AConvQ* attribute), 161
 fc1 (*dicee.models.complex.AConEx* attribute), 106
 fc1 (*dicee.models.complex.ConEx* attribute), 105
 fc1 (*dicee.models.ConEx* attribute), 154
 fc1 (*dicee.models.ConvO* attribute), 167
 fc1 (*dicee.models.ConvQ* attribute), 161
 fc1 (*dicee.models.octonion.AConvO* attribute), 117
 fc1 (*dicee.models.octonion.ConvO* attribute), 116
 fc1 (*dicee.models.quaternion.AConvQ* attribute), 121
 fc1 (*dicee.models.quaternion.ConvQ* attribute), 121
 fc_num_input (*dicee.models.AConEx* attribute), 154
 fc_num_input (*dicee.models.AConvO* attribute), 167
 fc_num_input (*dicee.models.AConvQ* attribute), 161
 fc_num_input (*dicee.models.complex.AConEx* attribute), 106
 fc_num_input (*dicee.models.complex.ConEx* attribute), 105
 fc_num_input (*dicee.models.ConEx* attribute), 154
 fc_num_input (*dicee.models.ConvO* attribute), 167
 fc_num_input (*dicee.models.ConvQ* attribute), 161
 fc_num_input (*dicee.models.octonion.AConvO* attribute), 117
 fc_num_input (*dicee.models.octonion.ConvO* attribute), 116

`fc_num_input` (*dicee.models.quaternion.AConvQ attribute*), 121
`fc_num_input` (*dicee.models.quaternion.ConvQ attribute*), 121
`fc_out` (*dicee.models.literal.LiteralEmbeddings attribute*), 114
`feature_map_dropout` (*dicee.models.AConEx attribute*), 154
`feature_map_dropout` (*dicee.models.AConvO attribute*), 167
`feature_map_dropout` (*dicee.models.AConvQ attribute*), 161
`feature_map_dropout` (*dicee.models.complex.AConEx attribute*), 106
`feature_map_dropout` (*dicee.models.complex.ConEx attribute*), 105
`feature_map_dropout` (*dicee.models.ConEx attribute*), 154
`feature_map_dropout` (*dicee.models.ConvO attribute*), 167
`feature_map_dropout` (*dicee.models.ConvQ attribute*), 161
`feature_map_dropout` (*dicee.models.octonion.AConvO attribute*), 117
`feature_map_dropout` (*dicee.models.octonion.ConvO attribute*), 117
`feature_map_dropout` (*dicee.models.quaternion.AConvQ attribute*), 121
`feature_map_dropout` (*dicee.models.quaternion.ConvQ attribute*), 121
`feature_map_dropout_rate` (*dicee.config.Namespace attribute*), 30
`feature_map_dropout_rate` (*dicee.models.base_model.BaseKGE attribute*), 95
`feature_map_dropout_rate` (*dicee.models.BaseKGE attribute*), 143, 146, 152, 157, 163, 175, 179
`fetch_worker()` (in module *dicee.read_preprocess_save_load_kg.util*), 190
`fill_query()` (*dicee.query_generator.QueryGenerator method*), 185
`fill_query()` (*dicee.QueryGenerator method*), 221
`find_good_batch_size()` (in module *dicee.trainer.model_parallelism*), 205
`find_missing_triples()` (*dicee.KGE method*), 220
`find_missing_triples()` (*dicee.knowledge_graph_embeddings.KGE method*), 79
`fit()` (*dicee.trainer.model_parallelism.TensorParallel method*), 205
`fit()` (*dicee.trainer.torch_trainer_ddp.TorchDDPTrainer method*), 207
`fit()` (*dicee.trainer.torch_trainer.TorchTrainer method*), 206
`flash` (*dicee.models.transformers.SelfAttention attribute*), 128
`FMult` (class in *dicee.models*), 180
`FMult` (class in *dicee.models.function_space*), 110
`FMult2` (class in *dicee.models*), 181
`FMult2` (class in *dicee.models.function_space*), 111
`form_of_labelling` (*dicee.DICE_Trainer attribute*), 222
`form_of_labelling` (*dicee.trainer.DICE_Trainer attribute*), 208
`form_of_labelling` (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 203
`forward()` (*dicee.models.base_model.BaseKGE method*), 96
`forward()` (*dicee.models.base_model.IdentityClass static method*), 98
`forward()` (*dicee.models.BaseKGE method*), 144, 147, 153, 157, 163, 176, 180
`forward()` (*dicee.models.Block method*), 149
`forward()` (*dicee.models.IdentityClass static method*), 145, 159, 165
`forward()` (*dicee.models.literal.LiteralEmbeddings method*), 114
`forward()` (*dicee.models.transformers.Block method*), 130
`forward()` (*dicee.models.transformers.Byte method*), 126
`forward()` (*dicee.models.transformers.GPT method*), 131
`forward()` (*dicee.models.transformers.LayerNorm method*), 127
`forward()` (*dicee.models.transformers.MLP method*), 129
`forward()` (*dicee.models.transformers.SelfAttention method*), 128
`forward_backward_update()` (*dicee.trainer.torch_trainer.TorchTrainer method*), 206
`forward_backward_update_loss()` (in module *dicee.trainer.model_parallelism*), 205
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.base_model.BaseKGE method*), 96
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.BaseKGE method*), 144, 147, 153, 157, 163, 176, 179
`forward_byte_pair_encoded_triple()` (*dicee.models.base_model.BaseKGE method*), 96
`forward_byte_pair_encoded_triple()` (*dicee.models.BaseKGE method*), 144, 147, 153, 157, 163, 176, 179
`forward_k_vs_all()` (*dicee.models.AConEx method*), 154
`forward_k_vs_all()` (*dicee.models.AConvO method*), 167
`forward_k_vs_all()` (*dicee.models.AConvQ method*), 162
`forward_k_vs_all()` (*dicee.models.base_model.BaseKGE method*), 97
`forward_k_vs_all()` (*dicee.models.BaseKGE method*), 144, 147, 153, 158, 164, 177, 180
`forward_k_vs_all()` (*dicee.models.clifford.DeCaL method*), 103
`forward_k_vs_all()` (*dicee.models.clifford.Keci method*), 100
`forward_k_vs_all()` (*dicee.models.CoKE method*), 151
`forward_k_vs_all()` (*dicee.models.ComplEx method*), 155
`forward_k_vs_all()` (*dicee.models.complex.AConEx method*), 106
`forward_k_vs_all()` (*dicee.models.complex.ComplEx method*), 107
`forward_k_vs_all()` (*dicee.models.complex.ConEx method*), 106
`forward_k_vs_all()` (*dicee.models.ConEx method*), 154
`forward_k_vs_all()` (*dicee.models.ConvO method*), 167
`forward_k_vs_all()` (*dicee.models.ConvQ method*), 161

`forward_k_vs_all()` (*dicee.models.DeCaL method*), 172
`forward_k_vs_all()` (*dicee.models.DistMult method*), 149
`forward_k_vs_all()` (*dicee.models.DualE method*), 184
`forward_k_vs_all()` (*dicee.models.dualE.DualE method*), 108
`forward_k_vs_all()` (*dicee.models.Keci method*), 170
`forward_k_vs_all()` (*dicee.models.octonion.AConvO method*), 117
`forward_k_vs_all()` (*dicee.models.octonion.ConvO method*), 117
`forward_k_vs_all()` (*dicee.models.octonion.OMult method*), 116
`forward_k_vs_all()` (*dicee.models.OMult method*), 166
`forward_k_vs_all()` (*dicee.models.pykeen_models.PykeenKGE method*), 118
`forward_k_vs_all()` (*dicee.models.PykeenKGE method*), 177
`forward_k_vs_all()` (*dicee.models.QMult method*), 160
`forward_k_vs_all()` (*dicee.models.quaternion.AConvQ method*), 122
`forward_k_vs_all()` (*dicee.models.quaternion.ConvQ method*), 121
`forward_k_vs_all()` (*dicee.models.quaternion.QMult method*), 120
`forward_k_vs_all()` (*dicee.models.real.CoKE method*), 124
`forward_k_vs_all()` (*dicee.models.real.DistMult method*), 122
`forward_k_vs_all()` (*dicee.models.real.Shallom method*), 123
`forward_k_vs_all()` (*dicee.models.real.TransE method*), 123
`forward_k_vs_all()` (*dicee.models.Shallom method*), 149
`forward_k_vs_all()` (*dicee.models.TransE method*), 149
`forward_k_vs_sample()` (*dicee.models.AConEx method*), 155
`forward_k_vs_sample()` (*dicee.models.base_model.BaseKGE method*), 97
`forward_k_vs_sample()` (*dicee.models.BaseKGE method*), 144, 147, 153, 158, 164, 177, 180
`forward_k_vs_sample()` (*dicee.models.clifford.Keci method*), 101
`forward_k_vs_sample()` (*dicee.models.CoKE method*), 151
`forward_k_vs_sample()` (*dicee.models.ComplEx method*), 156
`forward_k_vs_sample()` (*dicee.models.complex.AConEx method*), 106
`forward_k_vs_sample()` (*dicee.models.complex.ComplEx method*), 107
`forward_k_vs_sample()` (*dicee.models.complex.ConEx method*), 106
`forward_k_vs_sample()` (*dicee.models.ConEx method*), 154
`forward_k_vs_sample()` (*dicee.models.DistMult method*), 149
`forward_k_vs_sample()` (*dicee.models.Keci method*), 170
`forward_k_vs_sample()` (*dicee.models.pykeen_models.PykeenKGE method*), 119
`forward_k_vs_sample()` (*dicee.models.PykeenKGE method*), 178
`forward_k_vs_sample()` (*dicee.models.QMult method*), 160
`forward_k_vs_sample()` (*dicee.models.quaternion.QMult method*), 120
`forward_k_vs_sample()` (*dicee.models.real.CoKE method*), 124
`forward_k_vs_sample()` (*dicee.models.real.DistMult method*), 122
`forward_k_vs_with_explicit()` (*dicee.models.clifford.Keci method*), 100
`forward_k_vs_with_explicit()` (*dicee.models.Keci method*), 170
`forward_triples()` (*dicee.models.AConEx method*), 154
`forward_triples()` (*dicee.models.AConvO method*), 167
`forward_triples()` (*dicee.models.AConvQ method*), 161
`forward_triples()` (*dicee.models.base_model.BaseKGE method*), 96
`forward_triples()` (*dicee.models.BaseKGE method*), 144, 147, 153, 158, 164, 176, 180
`forward_triples()` (*dicee.models.clifford.DeCaL method*), 102
`forward_triples()` (*dicee.models.clifford.Keci method*), 101
`forward_triples()` (*dicee.models.complex.AConEx method*), 106
`forward_triples()` (*dicee.models.complex.ConEx method*), 106
`forward_triples()` (*dicee.models.ConEx method*), 154
`forward_triples()` (*dicee.models.ConvO method*), 167
`forward_triples()` (*dicee.models.ConvQ method*), 161
`forward_triples()` (*dicee.models.DeCaL method*), 172
`forward_triples()` (*dicee.models.DualE method*), 184
`forward_triples()` (*dicee.models.dualE.DualE method*), 108
`forward_triples()` (*dicee.models.FMult method*), 181
`forward_triples()` (*dicee.models.FMult2 method*), 182
`forward_triples()` (*dicee.models.function_space.FMult method*), 110
`forward_triples()` (*dicee.models.function_space.FMult2 method*), 111
`forward_triples()` (*dicee.models.function_space.GFMult method*), 111
`forward_triples()` (*dicee.models.function_space.LFMult method*), 112
`forward_triples()` (*dicee.models.function_space.LFMult1 method*), 111
`forward_triples()` (*dicee.models.GFMult method*), 181
`forward_triples()` (*dicee.models.Keci method*), 171
`forward_triples()` (*dicee.models.LFMult method*), 182
`forward_triples()` (*dicee.models.LFMult1 method*), 182
`forward_triples()` (*dicee.models.octonion.AConvO method*), 117

`forward_triples()` (*dicdee.models.octonion.ConvO method*), 117
`forward_triples()` (*dicdee.models.Pyke method*), 150
`forward_triples()` (*dicdee.models.pykeen_models.PykeenKGE method*), 118
`forward_triples()` (*dicdee.models.PykeenKGE method*), 178
`forward_triples()` (*dicdee.models.quaternion.AConvQ method*), 122
`forward_triples()` (*dicdee.models.quaternion.ConvQ method*), 121
`forward_triples()` (*dicdee.models.real.Pyke method*), 123
`forward_triples()` (*dicdee.models.real.Shallom method*), 123
`forward_triples()` (*dicdee.models.Shallom method*), 149
`freeze_entity_embeddings` (*dicdee.models.literal.LiteralEmbeddings attribute*), 114
`frequency` (*dicdee.callbacks.Perturb attribute*), 26
`from_pretrained()` (*dicdee.models.transformers.GPT class method*), 131
`from_pretrained_model_write_embeddings_into_csv()` (*in module dicdee.static_funcs*), 199
`full_storage_path` (*dicdee.analyse_experiments.Experiment attribute*), 20
`func_triple_to_bpe_representation` (*dicdee.evaluation.Evaluator attribute*), 60
`func_triple_to_bpe_representation` (*dicdee.evaluation.evaluator.Evaluator attribute*), 51
`func_triple_to_bpe_representation` (*dicdee.Evaluator attribute*), 224
`func_triple_to_bpe_representation` (*dicdee.evaluator.Evaluator attribute*), 68
`func_triple_to_bpe_representation()` (*dicdee.knowledge_graph.KG method*), 76
`function()` (*dicdee.models.FMult2 method*), 182
`function()` (*dicdee.models.function_space.FMult2 method*), 111

G

`gamma` (*dicdee.models.FMult attribute*), 181
`gamma` (*dicdee.models.function_space.FMult attribute*), 110
`gate_residual` (*dicdee.models.literal.LiteralEmbeddings attribute*), 113, 114
`gated_residual_proj` (*dicdee.models.literal.LiteralEmbeddings attribute*), 114
`gelu` (*dicdee.models.transformers.MLP attribute*), 129
`gen_test` (*dicdee.query_generator.QueryGenerator attribute*), 185
`gen_test` (*dicdee.QueryGenerator attribute*), 220
`gen_valid` (*dicdee.query_generator.QueryGenerator attribute*), 185
`gen_valid` (*dicdee.QueryGenerator attribute*), 220
`generate()` (*dicdee.KGE method*), 217
`generate()` (*dicdee.knowledge_graph_embeddings.KGE method*), 77
`generate()` (*dicdee.models.transformers.BytE method*), 126
`generate_queries()` (*dicdee.query_generator.QueryGenerator method*), 185
`generate_queries()` (*dicdee.QueryGenerator method*), 221
`get_aswa_state_dict()` (*dicdee.weight_averaging.ASWA method*), 210
`get_bpe_head_and_relation_representation()` (*dicdee.models.base_model.BaseKGE method*), 97
`get_bpe_head_and_relation_representation()` (*dicdee.models.BaseKGE method*), 144, 148, 153, 158, 164, 177, 180
`get_bpe_token_representation()` (*dicdee.abstracts.BaseInteractiveKGE method*), 15
`get_callbacks()` (*in module dicdee.trainer.dice_trainer*), 203
`get_default_arguments()` (*in module dicdee.analyse_experiments*), 20
`get_default_arguments()` (*in module dicdee.scripts.index_serve*), 194
`get_default_arguments()` (*in module dicdee.scripts.run*), 195
`get_ee_vocab()` (*in module dicdee.read_preprocess_save_load_kg.util*), 190
`get_ee_vocab()` (*in module dicdee.static_funcs*), 197
`get_ee_vocab()` (*in module dicdee.static_preprocess_funcs*), 202
`get_embeddings()` (*dicdee.models.base_model.BaseKGE method*), 97
`get_embeddings()` (*dicdee.models.BaseKGE method*), 144, 148, 153, 158, 164, 177, 180
`get_embeddings()` (*dicdee.models.ensemble.EnsembleKGE method*), 109
`get_embeddings()` (*dicdee.models.real.Shallom method*), 123
`get_embeddings()` (*dicdee.models.Shallom method*), 149
`get_entity_embeddings()` (*dicdee.abstracts.BaseInteractiveKGE method*), 16
`get_entity_index()` (*dicdee.abstracts.BaseInteractiveKGE method*), 15
`get_er_vocab()` (*in module dicdee.read_preprocess_save_load_kg.util*), 190
`get_er_vocab()` (*in module dicdee.static_funcs*), 196
`get_er_vocab()` (*in module dicdee.static_preprocess_funcs*), 202
`get_eval_report()` (*dicdee.abstracts.BaseInteractiveKGE method*), 15
`get_head_relation_representation()` (*dicdee.models.base_model.BaseKGE method*), 97
`get_head_relation_representation()` (*dicdee.models.BaseKGE method*), 144, 147, 153, 158, 164, 177, 180
`get_kronecker_triple_representation()` (*dicdee.callbacks.KronE method*), 25
`get_mean_and_var()` (*dicdee.weight_averaging.SWAG method*), 212
`get_num_params()` (*dicdee.models.transformers.GPT method*), 131
`get_padded_bpe_triple_representation()` (*dicdee.abstracts.BaseInteractiveKGE method*), 15
`get_queries()` (*dicdee.query_generator.QueryGenerator method*), 186
`get_queries()` (*dicdee.QueryGenerator method*), 221

`get_re_vocab()` (in module *dicee.read_preprocess_save_load_kg.util*), 190
`get_re_vocab()` (in module *dicee.static_funcs*), 196
`get_re_vocab()` (in module *dicee.static_preprocess_funcs*), 202
`get_relation_embeddings()` (*dicee.abstracts.BaseInteractiveKGE method*), 16
`get_relation_index()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`get_sentence_representation()` (*dicee.models.base_model.BaseKGE method*), 97
`get_sentence_representation()` (*dicee.models.BaseKGE method*), 144, 147, 153, 158, 164, 177, 180
`get_transductive_entity_embeddings()` (*dicee.KGE method*), 217
`get_transductive_entity_embeddings()` (*dicee.knowledge_graph_embeddings.KGE method*), 77
`get_triple_representation()` (*dicee.models.base_model.BaseKGE method*), 97
`get_triple_representation()` (*dicee.models.BaseKGE method*), 144, 147, 153, 158, 164, 177, 180
`GFMult` (class in *dicee.models*), 181
`GFMult` (class in *dicee.models.function_space*), 110
`global_rank` (*dicee.abstracts.AbstractTrainer attribute*), 13
`global_rank` (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 207
`GPT` (class in *dicee.models.transformers*), 130
`GPTConfig` (class in *dicee.models.transformers*), 130
`gpus` (*dicee.config.Namespace attribute*), 29
`gradient_accumulation_steps` (*dicee.config.Namespace attribute*), 29
`ground_queries()` (*dicee.query_generator.QueryGenerator method*), 185
`ground_queries()` (*dicee.QueryGenerator method*), 221
`gswa_n` (*dicee.weight_averaging.SWAG attribute*), 212

H

`hidden_dim` (*dicee.models.literal.LiteralEmbeddings attribute*), 114
`hidden_dropout` (*dicee.models.base_model.BaseKGE attribute*), 96
`hidden_dropout` (*dicee.models.BaseKGE attribute*), 143, 147, 152, 157, 163, 176, 179
`hidden_dropout_rate` (*dicee.config.Namespace attribute*), 30
`hidden_dropout_rate` (*dicee.models.base_model.BaseKGE attribute*), 95
`hidden_dropout_rate` (*dicee.models.BaseKGE attribute*), 143, 146, 152, 157, 163, 175, 179
`hidden_normalizer` (*dicee.models.base_model.BaseKGE attribute*), 96
`hidden_normalizer` (*dicee.models.BaseKGE attribute*), 143, 147, 152, 157, 163, 176, 179

I

`IdentityClass` (class in *dicee.models*), 144, 158, 164
`IdentityClass` (class in *dicee.models.base_model*), 97
`idx_entity_to_bpe_shaped` (*dicee.knowledge_graph.KG attribute*), 75
`index()` (in module *dicee.scripts.index_serve*), 194
`index_triple()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`init_dataloader()` (*dicee.DICE_Trainer method*), 222
`init_dataloader()` (*dicee.trainer.DICE_Trainer method*), 209
`init_dataloader()` (*dicee.trainer.dice_trainer.DICE_Trainer method*), 204
`init_dataset()` (*dicee.DICE_Trainer method*), 222
`init_dataset()` (*dicee.trainer.DICE_Trainer method*), 209
`init_dataset()` (*dicee.trainer.dice_trainer.DICE_Trainer method*), 204
`init_param` (*dicee.config.Namespace attribute*), 29
`init_params_with_sanity_checking()` (*dicee.models.base_model.BaseKGE method*), 96
`init_params_with_sanity_checking()` (*dicee.models.BaseKGE method*), 144, 147, 153, 157, 163, 176, 180
`initial_eval_setting` (*dicee.weight_averaging.ASWA attribute*), 210
`initialize_or_load_model()` (*dicee.DICE_Trainer method*), 222
`initialize_or_load_model()` (*dicee.trainer.DICE_Trainer method*), 209
`initialize_or_load_model()` (*dicee.trainer.dice_trainer.DICE_Trainer method*), 204
`initialize_trainer()` (*dicee.DICE_Trainer method*), 222
`initialize_trainer()` (*dicee.trainer.DICE_Trainer method*), 209
`initialize_trainer()` (*dicee.trainer.dice_trainer.DICE_Trainer method*), 204
`initialize_trainer()` (in module *dicee.trainer.dice_trainer*), 203
`input_dp_ent_real` (*dicee.models.base_model.BaseKGE attribute*), 96
`input_dp_ent_real` (*dicee.models.BaseKGE attribute*), 143, 147, 152, 157, 163, 176, 179
`input_dp_rel_real` (*dicee.models.base_model.BaseKGE attribute*), 96
`input_dp_rel_real` (*dicee.models.BaseKGE attribute*), 143, 147, 152, 157, 163, 176, 179
`input_dropout_rate` (*dicee.config.Namespace attribute*), 30
`input_dropout_rate` (*dicee.models.base_model.BaseKGE attribute*), 95
`input_dropout_rate` (*dicee.models.BaseKGE attribute*), 143, 146, 152, 156, 163, 175, 179
`InteractiveQueryDecomposition` (class in *dicee.abstracts*), 16
`intialize_model()` (in module *dicee.static_funcs*), 198
`is_continual_training` (*dicee.DICE_Trainer attribute*), 222
`is_continual_training` (*dicee.evaluation.Evaluator attribute*), 60

is_continual_training (*dicee.evaluation.evaluator.Evaluator attribute*), 51
 is_continual_training (*dicee.Evaluator attribute*), 224
 is_continual_training (*dicee.evaluator.Evaluator attribute*), 68
 is_continual_training (*dicee.Execute attribute*), 216
 is_continual_training (*dicee.executer.Execute attribute*), 72
 is_continual_training (*dicee.trainer.DICE_Trainer attribute*), 208
 is_continual_training (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 203
 is_global_zero (*dicee.abstracts.AbstractTrainer attribute*), 13
 is_rank_zero () (*dicee.Execute method*), 216
 is_rank_zero () (*dicee.executer.Execute method*), 72
 is_seen () (*dicee.abstracts.BaseInteractiveKGE method*), 15
 is_sparql_endpoint_alive () (*in module dicee.sanity_checkers*), 193

K

k (*dicee.models.FMult attribute*), 180
 k (*dicee.models.FMult2 attribute*), 181
 k (*dicee.models.function_space.FMult attribute*), 110
 k (*dicee.models.function_space.FMult2 attribute*), 111
 k (*dicee.models.function_space.GFMult attribute*), 110
 k (*dicee.models.GFMult attribute*), 181
 k_fold_cross_validation () (*dicee.DICE_Trainer method*), 223
 k_fold_cross_validation () (*dicee.trainer.DICE_Trainer method*), 209
 k_fold_cross_validation () (*dicee.trainer.dice_trainer.DICE_Trainer method*), 204
 k_vs_all_score () (*dicee.models.clifford.Keci method*), 100
 k_vs_all_score () (*dicee.models.ComplEx static method*), 155
 k_vs_all_score () (*dicee.models.complex.ComplEx static method*), 107
 k_vs_all_score () (*dicee.models.DistMult method*), 149
 k_vs_all_score () (*dicee.models.Keci method*), 170
 k_vs_all_score () (*dicee.models.octonion.OMult method*), 116
 k_vs_all_score () (*dicee.models.OMult method*), 166
 k_vs_all_score () (*dicee.models.QMult method*), 160
 k_vs_all_score () (*dicee.models.quaternion.QMult method*), 120
 k_vs_all_score () (*dicee.models.real.DistMult method*), 122
 Keci (*class in dicee.models*), 168
 Keci (*class in dicee.models.clifford*), 98
 kernel_size (*dicee.config.Namespace attribute*), 30
 kernel_size (*dicee.models.base_model.BaseKGE attribute*), 95
 kernel_size (*dicee.models.BaseKGE attribute*), 143, 146, 152, 157, 163, 175, 179
 KG (*class in dicee.knowledge_graph*), 74
 kg (*dicee.callbacks.PseudoLabellingCallback attribute*), 24
 kg (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk attribute*), 192
 kg (*dicee.read_preprocess_save_load_kg.PreprocessKG attribute*), 191
 kg (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG attribute*), 186
 kg (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk attribute*), 187
 kg (*dicee.read_preprocess_save_load_kg.ReadFromDisk attribute*), 192
 kg (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk attribute*), 187
 KGE (*class in dicee*), 217
 KGE (*class in dicee.knowledge_graph_embeddings*), 76
 KGESaveCallback (*class in dicee.callbacks*), 22
 knowledge_graph (*dicee.Execute attribute*), 215, 216
 knowledge_graph (*dicee.executer.Execute attribute*), 72
 KronE (*class in dicee.callbacks*), 25
 KvsAll (*class in dicee.dataset_classes*), 34
 kvsall_score () (*dicee.models.DualE method*), 184
 kvsall_score () (*dicee.models.dualE.DualE method*), 108
 KvsSampleDataset (*class in dicee.dataset_classes*), 37

L

label_smoothing_rate (*dicee.config.Namespace attribute*), 29
 label_smoothing_rate (*dicee.dataset_classes.AllvsAll attribute*), 35
 label_smoothing_rate (*dicee.dataset_classes.KvsAll attribute*), 35
 label_smoothing_rate (*dicee.dataset_classes.KvsSampleDataset attribute*), 38
 label_smoothing_rate (*dicee.dataset_classes.OnevsSample attribute*), 36, 37
 label_smoothing_rate (*dicee.dataset_classes.TriplePredictionDataset attribute*), 39
 labels (*dicee.dataset_classes.NegSampleDataset attribute*), 38
 layer_norm (*dicee.models.literal.LiteralEmbeddings attribute*), 114
 LayerNorm (*class in dicee.models.transformers*), 127

learning_rate (*dicee.models.base_model.BaseKGE* attribute), 95
learning_rate (*dicee.models.BaseKGE* attribute), 143, 146, 152, 156, 162, 175, 179
length (*dicee.dataset_classes.NegSampleDataset* attribute), 38
length (*dicee.dataset_classes.TriplePredictionDataset* attribute), 39
level (*dicee.callbacks.Perturb* attribute), 26
LFMult (class in *dicee.models*), 182
LFMult (class in *dicee.models.function_space*), 112
LFMult1 (class in *dicee.models*), 182
LFMult1 (class in *dicee.models.function_space*), 111
linear() (*dicee.models.function_space.LFMult* method), 112
linear() (*dicee.models.LFMult* method), 183
list2tuple() (*dicee.query_generator.QueryGenerator* method), 185
list2tuple() (*dicee.QueryGenerator* method), 221
LiteralDataset (class in *dicee.dataset_classes*), 43
LiteralEmbeddings (class in *dicee.models.literal*), 113
lm_head (*dicee.models.transformers.BytE* attribute), 126
lm_head (*dicee.models.transformers.GPT* attribute), 131
ln_1 (*dicee.models.Block* attribute), 148
ln_1 (*dicee.models.transformers.Block* attribute), 130
ln_2 (*dicee.models.Block* attribute), 148
ln_2 (*dicee.models.transformers.Block* attribute), 130
ln_f (*dicee.models.CoKE* attribute), 151
ln_f (*dicee.models.real.CoKE* attribute), 124
load() (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk* method), 192
load() (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk* method), 187
load_and_validate_literal_data() (*dicee.dataset_classes.LiteralDataset* static method), 44
load_from_mmap() (*dicee.Execute* method), 216
load_from_mmap() (*dicee.executer.Execute* method), 72
load_json() (in module *dicee.static_funcs*), 198
load_model() (in module *dicee.static_funcs*), 197
load_model_ensemble() (in module *dicee.static_funcs*), 198
load_numpy() (in module *dicee.static_funcs*), 199
load_numpy_ndarray() (in module *dicee.read_preprocess_save_load_kg.util*), 191
load_pickle() (in module *dicee.read_preprocess_save_load_kg.util*), 191
load_pickle() (in module *dicee.static_funcs*), 197
load_queries() (*dicee.query_generator.QueryGenerator* method), 186
load_queries() (*dicee.QueryGenerator* method), 221
load_queries_and_answers() (*dicee.query_generator.QueryGenerator* static method), 186
load_queries_and_answers() (*dicee.QueryGenerator* static method), 221
load_state_dict() (*dicee.models.ensemble.EnsembleKGE* method), 109
load_term_mapping() (in module *dicee.static_funcs*), 197
load_term_mapping() (in module *dicee.trainer.dice_trainer*), 202
load_with_pandas() (in module *dicee.read_preprocess_save_load_kg.util*), 191
loader_backend (*dicee.dataset_classes.LiteralDataset* attribute), 44
LoadSaveToDisk (class in *dicee.read_preprocess_save_load_kg*), 192
LoadSaveToDisk (class in *dicee.read_preprocess_save_load_kg.save_load_disk*), 187
local_rank (*dicee.abstracts.AbstractTrainer* attribute), 13
local_rank (*dicee.Execute* attribute), 215
local_rank (*dicee.executer.Execute* attribute), 71
local_rank (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 207
loss (*dicee.models.base_model.BaseKGE* attribute), 96
loss (*dicee.models.BaseKGE* attribute), 143, 146, 152, 157, 163, 176, 179
loss_func (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 207
loss_function (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 206
loss_function() (*dicee.models.base_model.BaseKGELightning* method), 91
loss_function() (*dicee.models.BaseKGELightning* method), 138
loss_function() (*dicee.models.transformers.BytE* method), 126
loss_history (*dicee.models.base_model.BaseKGE* attribute), 96
loss_history (*dicee.models.BaseKGE* attribute), 143, 147, 152, 157, 163, 176, 179
loss_history (*dicee.models.pykeen_models.PykeenKGE* attribute), 118
loss_history (*dicee.models.PykeenKGE* attribute), 177
loss_history (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 207
lr (*dicee.analyse_experiments.Experiment* attribute), 20
lr (*dicee.config.Namespace* attribute), 28
lr_init (*dicee.weight_averaging.SWA* attribute), 211
lr_init (*dicee.weight_averaging.SWAG* attribute), 212
lr_init (*dicee.weight_averaging.TWA* attribute), 214
lr_lambda (*dicee.callbacks.LRScheduler* attribute), 27

LRScheduler (class in *dicее.callbacks*), 27

M

m (*dicее.models.function_space.LFMult* attribute), 112
m (*dicее.models.LFMult* attribute), 182
main() (in module *dicее.scripts.index_serve*), 194
main() (in module *dicее.scripts.run*), 195
make_iterable_verbose() (in module *dicее.evaluation*), 67
make_iterable_verbose() (in module *dicее.evaluation.utils*), 57
make_iterable_verbose() (in module *dicее.static_funcs_training*), 200
make_iterable_verbose() (in module *dicее.trainer.torch_trainer_ddp*), 207
mapping_from_first_two_cols_to_third() (in module *dicее.static_preprocess_funcs*), 202
margin (*dicее.models.Pyke* attribute), 150
margin (*dicее.models.real.Pyke* attribute), 123
margin (*dicее.models.real.TransE* attribute), 123
margin (*dicее.models.TransE* attribute), 149
mask_emb (*dicее.models.CoKE* attribute), 151
mask_emb (*dicее.models.real.CoKE* attribute), 124
max_ans_num (*dicее.query_generator.QueryGenerator* attribute), 185
max_ans_num (*dicее.QueryGenerator* attribute), 221
max_epochs (*dicее.callbacks.KGESaveCallback* attribute), 23
max_epochs (*dicее.callbacks.PeriodicEvalCallback* attribute), 26
max_epochs (*dicее.weight_averaging.EMA* attribute), 213
max_epochs (*dicее.weight_averaging.SWA* attribute), 211
max_epochs (*dicее.weight_averaging.SWAG* attribute), 212
max_epochs (*dicее.weight_averaging.TWA* attribute), 214
max_length_subword_tokens (*dicее.knowledge_graph.KG* attribute), 75
max_length_subword_tokens (*dicее.models.base_model.BaseKGE* attribute), 96
max_length_subword_tokens (*dicее.models.BaseKGE* attribute), 143, 147, 153, 157, 163, 176, 179
max_num_models (*dicее.weight_averaging.SWAG* attribute), 212
max_num_of_classes (*dicее.dataset_classes.KvsSampleDataset* attribute), 38
mean (*dicее.weight_averaging.SWAG* attribute), 212
mem_of_model() (*dicее.models.base_model.BaseKGELightning* method), 89
mem_of_model() (*dicее.models.BaseKGELightning* method), 137
mem_of_model() (*dicее.models.ensemble.EnsembleKGE* method), 109
method (*dicее.callbacks.Perturb* attribute), 26
MLP (class in *dicее.models.transformers*), 128
mlp (*dicее.models.Block* attribute), 149
mlp (*dicее.models.transformers.Block* attribute), 130
mode (*dicее.query_generator.QueryGenerator* attribute), 185
mode (*dicее.QueryGenerator* attribute), 221
model (*dicее.config.Namespace* attribute), 28
model (*dicее.models.pykeen_models.PykeenKGE* attribute), 118
model (*dicее.models.PykeenKGE* attribute), 177
model (*dicее.trainer.torch_trainer_ddp.NodeTrainer* attribute), 207
model (*dicее.trainer.torch_trainer.TorchTrainer* attribute), 206
model_kwargs (*dicее.models.pykeen_models.PykeenKGE* attribute), 118
model_kwargs (*dicее.models.PykeenKGE* attribute), 177
model_name (*dicее.analyse_experiments.Experiment* attribute), 20
MODEL_REGISTRY (in module *dicее.static_funcs*), 196
module
 dicее, 12
 dicее.__main__, 12
 dicее.abstracts, 12
 dicее.analyse_experiments, 19
 dicее.callbacks, 21
 dicее.config, 28
 dicее.dataset_classes, 31
 dicее.eval_static_funcs, 45
 dicее.evaluation, 48
 dicее.evaluation.ensemble, 48
 dicее.evaluation.evaluator, 49
 dicее.evaluation.link_prediction, 53
 dicее.evaluation.literal_prediction, 56
 dicее.evaluation.utils, 57
 dicее.evaluator, 67
 dicее.executer, 71

- dicee.knowledge_graph, 74
- dicee.knowledge_graph_embeddings, 76
- dicee.models, 80
- dicee.models.adopt, 80
- dicee.models.base_model, 89
- dicee.models.clifford, 98
- dicee.models.complex, 105
- dicee.models.dualE, 107
- dicee.models.ensemble, 109
- dicee.models.function_space, 109
- dicee.models.literal, 113
- dicee.models.octonion, 114
- dicee.models.pykeen_models, 118
- dicee.models.quaternion, 119
- dicee.models.real, 122
- dicee.models.static_funcs, 125
- dicee.models.transformers, 125
- dicee.query_generator, 184
- dicee.read_preprocess_save_load_kg, 186
- dicee.read_preprocess_save_load_kg.preprocess, 186
- dicee.read_preprocess_save_load_kg.read_from_disk, 187
- dicee.read_preprocess_save_load_kg.save_load_disk, 187
- dicee.read_preprocess_save_load_kg.util, 188
- dicee.sanity_checkers, 192
- dicee.scripts, 193
- dicee.scripts.index_serve, 193
- dicee.scripts.run, 194
- dicee.static_funcs, 195
- dicee.static_funcs_training, 199
- dicee.static_preprocess_funcs, 201
- dicee.trainer, 202
- dicee.trainer.dice_trainer, 202
- dicee.trainer.model_parallelism, 204
- dicee.trainer.torch_trainer, 205
- dicee.trainer.torch_trainer_ddp, 206
- dicee.weight_averaging, 209
- modules() (*dicee.models.ensemble.EnsembleKGE method*), 109
- moving_average() (*dicee.weight_averaging.SWA static method*), 211
- MultiClassClassificationDataset (*class in dicee.dataset_classes*), 33
- MultiLabelDataset (*class in dicee.dataset_classes*), 32

N

- n (*dicee.models.FMult2 attribute*), 181
- n (*dicee.models.function_space.FMult2 attribute*), 111
- n_embd (*dicee.models.CoKEConfig attribute*), 150
- n_embd (*dicee.models.real.CoKEConfig attribute*), 124
- n_embd (*dicee.models.transformers.GPTConfig attribute*), 130
- n_embd (*dicee.models.transformers.SelfAttention attribute*), 128
- n_epochs_eval_model (*dicee.callbacks.PeriodicEvalCallback attribute*), 26
- n_epochs_eval_model (*dicee.config.Namespace attribute*), 31
- n_head (*dicee.models.CoKEConfig attribute*), 150
- n_head (*dicee.models.real.CoKEConfig attribute*), 123, 124
- n_head (*dicee.models.transformers.GPTConfig attribute*), 130
- n_head (*dicee.models.transformers.SelfAttention attribute*), 128
- n_layer (*dicee.models.CoKEConfig attribute*), 150
- n_layer (*dicee.models.real.CoKEConfig attribute*), 123, 124
- n_layer (*dicee.models.transformers.GPTConfig attribute*), 130
- n_layers (*dicee.models.FMult2 attribute*), 181
- n_layers (*dicee.models.function_space.FMult2 attribute*), 111
- name (*dicee.abstracts.BaseInteractiveKGE property*), 15
- name (*dicee.models.AConEx attribute*), 154
- name (*dicee.models.AConvO attribute*), 167
- name (*dicee.models.AConvQ attribute*), 161
- name (*dicee.models.CKeci attribute*), 171
- name (*dicee.models.clifford.CKeci attribute*), 101
- name (*dicee.models.clifford.DeCaL attribute*), 102
- name (*dicee.models.clifford.Keci attribute*), 99

name (*dicee.models.CoKE* attribute), 151
 name (*dicee.models.ComplEx* attribute), 155
 name (*dicee.models.complex.AConEx* attribute), 106
 name (*dicee.models.complex.ComplEx* attribute), 107
 name (*dicee.models.complex.ConEx* attribute), 105
 name (*dicee.models.ConEx* attribute), 154
 name (*dicee.models.ConvO* attribute), 167
 name (*dicee.models.ConvQ* attribute), 161
 name (*dicee.models.DeCaL* attribute), 171
 name (*dicee.models.DistMult* attribute), 149
 name (*dicee.models.DualE* attribute), 183
 name (*dicee.models.dualE.DualE* attribute), 108
 name (*dicee.models.ensemble.EnsembleKGE* attribute), 109
 name (*dicee.models.FMult* attribute), 180
 name (*dicee.models.FMult2* attribute), 181
 name (*dicee.models.function_space.FMult* attribute), 110
 name (*dicee.models.function_space.FMult2* attribute), 111
 name (*dicee.models.function_space.GFMult* attribute), 110
 name (*dicee.models.function_space.LFMult* attribute), 112
 name (*dicee.models.function_space.LFMult1* attribute), 111
 name (*dicee.models.GFMult* attribute), 181
 name (*dicee.models.Keci* attribute), 168
 name (*dicee.models.LFMult* attribute), 182
 name (*dicee.models.LFMult1* attribute), 182
 name (*dicee.models.octonion.AConvO* attribute), 117
 name (*dicee.models.octonion.ConvO* attribute), 116
 name (*dicee.models.octonion.OMult* attribute), 115
 name (*dicee.models.OMult* attribute), 166
 name (*dicee.models.Pyke* attribute), 150
 name (*dicee.models.pykeen_models.PykeenKGE* attribute), 118
 name (*dicee.models.PykeenKGE* attribute), 177
 name (*dicee.models.QMult* attribute), 160
 name (*dicee.models.quaternion.AConvQ* attribute), 121
 name (*dicee.models.quaternion.ConvQ* attribute), 121
 name (*dicee.models.quaternion.QMult* attribute), 120
 name (*dicee.models.real.CoKE* attribute), 124
 name (*dicee.models.real.DistMult* attribute), 122
 name (*dicee.models.real.Pyke* attribute), 123
 name (*dicee.models.real.Shallom* attribute), 123
 name (*dicee.models.real.TransE* attribute), 123
 name (*dicee.models.Shallom* attribute), 149
 name (*dicee.models.TransE* attribute), 149
 name (*dicee.models.transformers.BytE* attribute), 126
 named_children () (*dicee.models.ensemble.EnsembleKGE* method), 109
 Namespace (class in *dicee.config*), 28
 neg_ratio (*dicee.config.Namespace* attribute), 29
 neg_ratio (*dicee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 32
 neg_ratio (*dicee.dataset_classes.KvsSampleDataset* attribute), 38
 neg_sample_ratio (*dicee.dataset_classes.CVDDataModule* attribute), 40
 neg_sample_ratio (*dicee.dataset_classes.NegSampleDataset* attribute), 38
 neg_sample_ratio (*dicee.dataset_classes.OnevsSample* attribute), 36, 37
 neg_sample_ratio (*dicee.dataset_classes.TriplePredictionDataset* attribute), 39
 negnorm () (*dicee.abstracts.InteractiveQueryDecomposition* method), 16
 NegSampleDataset (class in *dicee.dataset_classes*), 38
 neural_searcher (in module *dicee.scripts.index_serve*), 194
 NeuralSearcher (class in *dicee.scripts.index_serve*), 194
 NodeTrainer (class in *dicee.trainer.torch_trainer_ddp*), 207
 norm_fc1 (*dicee.models.AConEx* attribute), 154
 norm_fc1 (*dicee.models.AConvO* attribute), 167
 norm_fc1 (*dicee.models.complex.AConEx* attribute), 106
 norm_fc1 (*dicee.models.complex.ConEx* attribute), 105
 norm_fc1 (*dicee.models.ConEx* attribute), 154
 norm_fc1 (*dicee.models.ConvO* attribute), 167
 norm_fc1 (*dicee.models.octonion.AConvO* attribute), 117
 norm_fc1 (*dicee.models.octonion.ConvO* attribute), 117
 normalization (*dicee.analyse_experiments.Experiment* attribute), 20
 normalization (*dicee.config.Namespace* attribute), 29
 normalization (*dicee.dataset_classes.LiteralDataset* attribute), 43

normalization_params (*dicdee.dataset_classes.LiteralDataset* attribute), 43, 44
 normalization_type (*dicdee.dataset_classes.LiteralDataset* attribute), 44
 normalize_head_entity_embeddings (*dicdee.models.base_model.BaseKGE* attribute), 96
 normalize_head_entity_embeddings (*dicdee.models.BaseKGE* attribute), 143, 146, 152, 157, 163, 176, 179
 normalize_relation_embeddings (*dicdee.models.base_model.BaseKGE* attribute), 96
 normalize_relation_embeddings (*dicdee.models.BaseKGE* attribute), 143, 146, 152, 157, 163, 176, 179
 normalize_tail_entity_embeddings (*dicdee.models.base_model.BaseKGE* attribute), 96
 normalize_tail_entity_embeddings (*dicdee.models.BaseKGE* attribute), 143, 147, 152, 157, 163, 176, 179
 normalizer_class (*dicdee.models.base_model.BaseKGE* attribute), 96
 normalizer_class (*dicdee.models.BaseKGE* attribute), 143, 146, 152, 157, 163, 176, 179
 num_bpe_entities (*dicdee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 32
 num_bpe_entities (*dicdee.knowledge_graph.KG* attribute), 75
 num_core (*dicdee.config.Namespace* attribute), 29
 num_data_properties (*dicdee.dataset_classes.LiteralDataset* attribute), 44
 num_datapoints (*dicdee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 32
 num_datapoints (*dicdee.dataset_classes.MultiLabelDataset* attribute), 33
 num_ent (*dicdee.models.DualE* attribute), 184
 num_ent (*dicdee.models.dualE.DualE* attribute), 108
 num_entities (*dicdee.dataset_classes.CVDDataModule* attribute), 40
 num_entities (*dicdee.dataset_classes.KvsSampleDataset* attribute), 38
 num_entities (*dicdee.dataset_classes.LiteralDataset* attribute), 43, 44
 num_entities (*dicdee.dataset_classes.NegSampleDataset* attribute), 38
 num_entities (*dicdee.dataset_classes.OnevsSample* attribute), 36
 num_entities (*dicdee.dataset_classes.TriplePredictionDataset* attribute), 39
 num_entities (*dicdee.evaluation.Evaluator* attribute), 60
 num_entities (*dicdee.evaluation.evaluator.Evaluator* attribute), 50, 51
 num_entities (*dicdee.Evaluator* attribute), 223, 224
 num_entities (*dicdee.evaluator.Evaluator* attribute), 68, 69
 num_entities (*dicdee.knowledge_graph.KG* attribute), 74, 75
 num_entities (*dicdee.models.base_model.BaseKGE* attribute), 95
 num_entities (*dicdee.models.BaseKGE* attribute), 143, 146, 152, 156, 162, 175, 179
 num_epochs (*dicdee.abstracts.AbstractPPECallback* attribute), 18
 num_epochs (*dicdee.analyse_experiments.Experiment* attribute), 20
 num_epochs (*dicdee.config.Namespace* attribute), 28
 num_epochs (*dicdee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 207
 num_epochs (*dicdee.weight_averaging.ASWA* attribute), 210
 num_folds_for_cv (*dicdee.config.Namespace* attribute), 29
 num_of_data_points (*dicdee.dataset_classes.MultiClassClassificationDataset* attribute), 33
 num_of_data_properties (*dicdee.models.literal.LiteralEmbeddings* attribute), 113, 114
 num_of_epochs (*dicdee.callbacks.PseudoLabellingCallback* attribute), 24
 num_of_output_channels (*dicdee.config.Namespace* attribute), 30
 num_of_output_channels (*dicdee.models.base_model.BaseKGE* attribute), 96
 num_of_output_channels (*dicdee.models.BaseKGE* attribute), 143, 146, 152, 157, 163, 176, 179
 num_params (*dicdee.analyse_experiments.Experiment* attribute), 20
 num_relations (*dicdee.dataset_classes.CVDDataModule* attribute), 40
 num_relations (*dicdee.dataset_classes.NegSampleDataset* attribute), 38
 num_relations (*dicdee.dataset_classes.OnevsSample* attribute), 36
 num_relations (*dicdee.dataset_classes.TriplePredictionDataset* attribute), 39
 num_relations (*dicdee.evaluation.Evaluator* attribute), 60
 num_relations (*dicdee.evaluation.evaluator.Evaluator* attribute), 50, 51
 num_relations (*dicdee.Evaluator* attribute), 223, 224
 num_relations (*dicdee.evaluator.Evaluator* attribute), 68, 69
 num_relations (*dicdee.knowledge_graph.KG* attribute), 74, 75
 num_relations (*dicdee.models.base_model.BaseKGE* attribute), 95
 num_relations (*dicdee.models.BaseKGE* attribute), 143, 146, 152, 156, 162, 175, 179
 num_sample (*dicdee.models.FMult* attribute), 180
 num_sample (*dicdee.models.function_space.FMult* attribute), 110
 num_sample (*dicdee.models.function_space.GFMult* attribute), 110
 num_sample (*dicdee.models.GFMult* attribute), 181
 num_samples (*dicdee.weight_averaging.TWA* attribute), 214
 num_tokens (*dicdee.knowledge_graph.KG* attribute), 75
 num_tokens (*dicdee.models.base_model.BaseKGE* attribute), 95
 num_tokens (*dicdee.models.BaseKGE* attribute), 143, 146, 152, 156, 162, 175, 179
 num_workers (*dicdee.dataset_classes.CVDDataModule* attribute), 40
 numpy_data_type_changer () (in module *dicdee.static_funcs*), 198

O

`octonion_mul()` (in module `dicee.models`), 165
`octonion_mul()` (in module `dicee.models.octonion`), 115
`octonion_mul_norm()` (in module `dicee.models`), 165
`octonion_mul_norm()` (in module `dicee.models.octonion`), 115
`octonion_normalizer()` (`dicee.models.AConvO` static method), 167
`octonion_normalizer()` (`dicee.models.ConvO` static method), 167
`octonion_normalizer()` (`dicee.models.octonion.AConvO` static method), 117
`octonion_normalizer()` (`dicee.models.octonion.ConvO` static method), 117
`octonion_normalizer()` (`dicee.models.octonion.OMult` static method), 116
`octonion_normalizer()` (`dicee.models.OMult` static method), 166
`OMult` (class in `dicee.models`), 165
`OMult` (class in `dicee.models.octonion`), 115
`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.Eval` method), 25
`on_fit_end()` (`dicee.callbacks.KGESaveCallback` method), 23
`on_fit_end()` (`dicee.callbacks.LRScheduler` method), 27
`on_fit_end()` (`dicee.callbacks.PeriodicEvalCallback` method), 27
`on_fit_end()` (`dicee.callbacks.PrintCallback` method), 22
`on_fit_end()` (`dicee.weight_averaging.ASWA` method), 210
`on_fit_end()` (`dicee.weight_averaging.EMA` method), 213
`on_fit_end()` (`dicee.weight_averaging.SWA` method), 211
`on_fit_end()` (`dicee.weight_averaging.SWAG` method), 212
`on_fit_end()` (`dicee.weight_averaging.TWA` method), 214
`on_fit_start()` (`dicee.abstracts.AbstractCallback` method), 17
`on_fit_start()` (`dicee.abstracts.AbstractPPECallback` method), 18
`on_fit_start()` (`dicee.abstracts.AbstractTrainer` method), 13
`on_fit_start()` (`dicee.callbacks.Eval` method), 24
`on_fit_start()` (`dicee.callbacks.KGESaveCallback` method), 23
`on_fit_start()` (`dicee.callbacks.KronE` method), 26
`on_fit_start()` (`dicee.callbacks.PrintCallback` method), 22
`on_init_end()` (`dicee.abstracts.AbstractCallback` method), 17
`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), 14
`on_train_batch_end()` (`dicee.callbacks.Eval` method), 25
`on_train_batch_end()` (`dicee.callbacks.KGESaveCallback` method), 23
`on_train_batch_end()` (`dicee.callbacks.LRScheduler` method), 27
`on_train_batch_end()` (`dicee.callbacks.PrintCallback` method), 22
`on_train_batch_start()` (`dicee.callbacks.Perturb` method), 26
`on_train_epoch_end()` (`dicee.abstracts.AbstractCallback` method), 17
`on_train_epoch_end()` (`dicee.abstracts.AbstractTrainer` method), 14
`on_train_epoch_end()` (`dicee.callbacks.Eval` method), 25
`on_train_epoch_end()` (`dicee.callbacks.KGESaveCallback` method), 23
`on_train_epoch_end()` (`dicee.callbacks.PeriodicEvalCallback` method), 27
`on_train_epoch_end()` (`dicee.callbacks.PrintCallback` method), 22
`on_train_epoch_end()` (`dicee.models.base_model.BaseKGELightning` method), 91
`on_train_epoch_end()` (`dicee.models.BaseKGELightning` method), 138
`on_train_epoch_end()` (`dicee.weight_averaging.ASWA` method), 210
`on_train_epoch_end()` (`dicee.weight_averaging.EMA` method), 213
`on_train_epoch_end()` (`dicee.weight_averaging.SWA` method), 211
`on_train_epoch_end()` (`dicee.weight_averaging.SWAG` method), 212
`on_train_epoch_end()` (`dicee.weight_averaging.TWA` method), 214
`on_train_epoch_start()` (`dicee.abstracts.AbstractTrainer` method), 13
`on_train_epoch_start()` (`dicee.weight_averaging.EMA` method), 213
`on_train_epoch_start()` (`dicee.weight_averaging.SWA` method), 211
`on_train_epoch_start()` (`dicee.weight_averaging.SWAG` method), 212
`on_train_epoch_start()` (`dicee.weight_averaging.TWA` method), 214
`on_train_start()` (`dicee.callbacks.LRScheduler` method), 27
`OnevsAllDataset` (class in `dicee.dataset_classes`), 33
`OnevsSample` (class in `dicee.dataset_classes`), 35
`optim` (`dicee.config.Namespace` attribute), 28
`optimizer` (`dicee.trainer.torch_trainer_ddp.NodeTrainer` attribute), 207

optimizer (*dicdee.trainer.torch_trainer.TorchTrainer* attribute), 206
optimizer_name (*dicdee.models.base_model.BaseKGE* attribute), 95
optimizer_name (*dicdee.models.BaseKGE* attribute), 143, 146, 152, 157, 163, 175, 179
ordered_bpe_entities (*dicdee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 32
ordered_bpe_entities (*dicdee.knowledge_graph.KG* attribute), 76
ordered_shaped_bpe_tokens (*dicdee.knowledge_graph.KG* attribute), 75

P

p (*dicdee.config.Namespace* attribute), 30
p (*dicdee.models.clifford.DeCaL* attribute), 102
p (*dicdee.models.clifford.Keci* attribute), 99
p (*dicdee.models.DeCaL* attribute), 172
p (*dicdee.models.Keci* attribute), 168
P (*dicdee.weight_averaging.TWA* attribute), 214
padding (*dicdee.knowledge_graph.KG* attribute), 75
pandas_dataframe_indexer() (in module *dicdee.read_preprocess_save_load_kg.util*), 189
param_init (*dicdee.models.base_model.BaseKGE* attribute), 96
param_init (*dicdee.models.BaseKGE* attribute), 143, 147, 152, 157, 163, 176, 179
parameters() (*dicdee.abstracts.BaseInteractiveKGE* method), 16
parameters() (*dicdee.models.ensemble.EnsembleKGE* method), 109
path (*dicdee.abstracts.AbstractPPECallback* attribute), 18
path (*dicdee.callbacks.AccumulateEpochLossCallback* attribute), 21
path (*dicdee.callbacks.Eval* attribute), 24
path (*dicdee.callbacks.KGESaveCallback* attribute), 23
path (*dicdee.weight_averaging.ASWA* attribute), 210
path_dataset_folder (*dicdee.analyse_experiments.Experiment* attribute), 20
path_for_deserialization (*dicdee.knowledge_graph.KG* attribute), 75
path_for_serialization (*dicdee.knowledge_graph.KG* attribute), 75
path_single_kg (*dicdee.config.Namespace* attribute), 28
path_single_kg (*dicdee.knowledge_graph.KG* attribute), 75
path_to_store_single_run (*dicdee.config.Namespace* attribute), 28
PeriodicEvalCallback (class in *dicdee.callbacks*), 26
Perturb (class in *dicdee.callbacks*), 26
polars_dataframe_indexer() (in module *dicdee.read_preprocess_save_load_kg.util*), 188
poly_NN() (*dicdee.models.function_space.LFMMult* method), 112
poly_NN() (*dicdee.models.LFMMult* method), 183
polynomial() (*dicdee.models.function_space.LFMMult* method), 113
polynomial() (*dicdee.models.LFMMult* method), 183
pop() (*dicdee.models.function_space.LFMMult* method), 113
pop() (*dicdee.models.LFMMult* method), 183
pos_emb (*dicdee.models.CoKE* attribute), 151
pos_emb (*dicdee.models.real.CoKE* attribute), 124
pq (*dicdee.analyse_experiments.Experiment* attribute), 20
predict() (*dicdee.KGE* method), 218
predict() (*dicdee.knowledge_graph_embeddings.KGE* method), 78
predict_data_loader() (*dicdee.models.base_model.BaseKGELightning* method), 92
predict_data_loader() (*dicdee.models.BaseKGELightning* method), 140
predict_literals() (*dicdee.KGE* method), 220
predict_literals() (*dicdee.knowledge_graph_embeddings.KGE* method), 80
predict_missing_head_entity() (*dicdee.KGE* method), 217
predict_missing_head_entity() (*dicdee.knowledge_graph_embeddings.KGE* method), 77
predict_missing_relations() (*dicdee.KGE* method), 218
predict_missing_relations() (*dicdee.knowledge_graph_embeddings.KGE* method), 77
predict_missing_tail_entity() (*dicdee.KGE* method), 218
predict_missing_tail_entity() (*dicdee.knowledge_graph_embeddings.KGE* method), 77
predict_topk() (*dicdee.KGE* method), 218
predict_topk() (*dicdee.knowledge_graph_embeddings.KGE* method), 78
prepare_data() (*dicdee.dataset_classes.CVDataModule* method), 42
preprocess_with_byte_pair_encoding() (*dicdee.read_preprocess_save_load_kg.PreprocessKG* method), 191
preprocess_with_byte_pair_encoding() (*dicdee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 186
preprocess_with_byte_pair_encoding_with_padding() (*dicdee.read_preprocess_save_load_kg.PreprocessKG* method), 192
preprocess_with_byte_pair_encoding_with_padding() (*dicdee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 186
preprocess_with_pandas() (*dicdee.read_preprocess_save_load_kg.PreprocessKG* method), 192
preprocess_with_pandas() (*dicdee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 186
preprocess_with_polars() (*dicdee.read_preprocess_save_load_kg.PreprocessKG* method), 192
preprocess_with_polars() (*dicdee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 186
preprocesses_input_args() (in module *dicdee.static_preprocess_funcs*), 202

PreprocessKG (class in *dicee.read_preprocess_save_load_kg*), 191
 PreprocessKG (class in *dicee.read_preprocess_save_load_kg.preprocess*), 186
 PrintCallback (class in *dicee.callbacks*), 22
 process (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 206
 PseudoLabellingCallback (class in *dicee.callbacks*), 24
 Pyke (class in *dicee.models*), 149
 Pyke (class in *dicee.models.real*), 123
 pykeen_model_kwargs (*dicee.config.Namespace* attribute), 30
 PykeenKGE (class in *dicee.models*), 177
 PykeenKGE (class in *dicee.models.pykeen_models*), 118

Q

q (*dicee.config.Namespace* attribute), 30
 q (*dicee.models.clifford.DeCaL* attribute), 102
 q (*dicee.models.clifford.Keci* attribute), 99
 q (*dicee.models.DeCaL* attribute), 172
 q (*dicee.models.Keci* attribute), 168
 qdrant_client (*dicee.scripts.index_serve.NeuralSearcher* attribute), 194
 QMult (class in *dicee.models*), 159
 QMult (class in *dicee.models.quaternion*), 119
 quaternion_mul() (in module *dicee.models*), 156
 quaternion_mul() (in module *dicee.models.static_funcs*), 125
 quaternion_mul_with_unit_norm() (in module *dicee.models*), 159
 quaternion_mul_with_unit_norm() (in module *dicee.models.quaternion*), 119
 quaternion_multiplication_followed_by_inner_product() (*dicee.models.QMult* method), 160
 quaternion_multiplication_followed_by_inner_product() (*dicee.models.quaternion.QMult* method), 120
 quaternion_normalizer() (*dicee.models.QMult* static method), 160
 quaternion_normalizer() (*dicee.models.quaternion.QMult* static method), 120
 queries (*dicee.scripts.index_serve.StringListRequest* attribute), 194
 query_name_to_struct (*dicee.query_generator.QueryGenerator* attribute), 185
 query_name_to_struct (*dicee.QueryGenerator* attribute), 221
 QueryGenerator (class in *dicee*), 220
 QueryGenerator (class in *dicee.query_generator*), 185

R

r (*dicee.models.clifford.DeCaL* attribute), 102
 r (*dicee.models.clifford.Keci* attribute), 99
 r (*dicee.models.DeCaL* attribute), 172
 r (*dicee.models.Keci* attribute), 168
 random_seed (*dicee.config.Namespace* attribute), 29
 range_constraints_per_rel (*dicee.evaluation.Evaluator* attribute), 60
 range_constraints_per_rel (*dicee.evaluation.evaluator.Evaluator* attribute), 51
 range_constraints_per_rel (*dicee.Evaluator* attribute), 224
 range_constraints_per_rel (*dicee.evaluator.Evaluator* attribute), 69
 rank (*dicee.Execute* attribute), 215
 rank (*dicee.executer.Execute* attribute), 71
 ratio (*dicee.callbacks.Perturb* attribute), 26
 raw_test_set (*dicee.knowledge_graph.KG* attribute), 75
 raw_train_set (*dicee.knowledge_graph.KG* attribute), 75
 raw_valid_set (*dicee.knowledge_graph.KG* attribute), 75
 re (*dicee.models.clifford.DeCaL* attribute), 102
 re (*dicee.models.DeCaL* attribute), 172
 re_vocab (*dicee.evaluation.Evaluator* attribute), 60
 re_vocab (*dicee.evaluation.evaluator.Evaluator* attribute), 50
 re_vocab (*dicee.Evaluator* attribute), 223
 re_vocab (*dicee.evaluator.Evaluator* attribute), 68
 read_from_disk() (in module *dicee.read_preprocess_save_load_kg.util*), 190
 read_from_triple_store_with_pandas() (in module *dicee.read_preprocess_save_load_kg.util*), 190
 read_from_triple_store_with_polars() (in module *dicee.read_preprocess_save_load_kg.util*), 190
 read_only_few (*dicee.config.Namespace* attribute), 30
 read_only_few (*dicee.knowledge_graph.KG* attribute), 75
 read_or_load_kg() (in module *dicee.static_funcs*), 198
 read_with_pandas() (in module *dicee.read_preprocess_save_load_kg.util*), 190
 read_with_polars() (in module *dicee.read_preprocess_save_load_kg.util*), 190
 ReadFromDisk (class in *dicee.read_preprocess_save_load_kg*), 192
 ReadFromDisk (class in *dicee.read_preprocess_save_load_kg.read_from_disk*), 187
 reducer (*dicee.scripts.index_serve.StringListRequest* attribute), 194

reg_lambda (*dicee.weight_averaging.TWA attribute*), 214
 rel2id (*dicee.query_generator.QueryGenerator attribute*), 185
 rel2id (*dicee.QueryGenerator attribute*), 221
 relation_embeddings (*dicee.models.AConvQ attribute*), 161
 relation_embeddings (*dicee.models.clifford.DeCaL attribute*), 102
 relation_embeddings (*dicee.models.ConvQ attribute*), 161
 relation_embeddings (*dicee.models.DeCaL attribute*), 172
 relation_embeddings (*dicee.models.DualE attribute*), 184
 relation_embeddings (*dicee.models.dualE.DualE attribute*), 108
 relation_embeddings (*dicee.models.FMult attribute*), 180
 relation_embeddings (*dicee.models.FMult2 attribute*), 182
 relation_embeddings (*dicee.models.function_space.FMult attribute*), 110
 relation_embeddings (*dicee.models.function_space.FMult2 attribute*), 111
 relation_embeddings (*dicee.models.function_space.GFMult attribute*), 110
 relation_embeddings (*dicee.models.function_space.LFMult attribute*), 112
 relation_embeddings (*dicee.models.function_space.LFMult1 attribute*), 111
 relation_embeddings (*dicee.models.GFMult attribute*), 181
 relation_embeddings (*dicee.models.LFMult attribute*), 182
 relation_embeddings (*dicee.models.LFMult1 attribute*), 182
 relation_embeddings (*dicee.models.pykeen_models.PykeenKGE attribute*), 118
 relation_embeddings (*dicee.models.PykeenKGE attribute*), 177
 relation_embeddings (*dicee.models.quaternion.AConvQ attribute*), 121
 relation_embeddings (*dicee.models.quaternion.ConvQ attribute*), 121
 relation_to_idx (*dicee.knowledge_graph.KG attribute*), 74, 75
 relations_str (*dicee.knowledge_graph.KG property*), 76
 reload_dataset () (in module *dicee.dataset_classes*), 32
 report (*dicee.DICE_Trainer attribute*), 222
 report (*dicee.evaluation.Evaluator attribute*), 60
 report (*dicee.evaluation.evaluator.Evaluator attribute*), 50, 51
 report (*dicee.Evaluator attribute*), 223, 224
 report (*dicee.evaluator.Evaluator attribute*), 68, 69
 report (*dicee.Execute attribute*), 215, 216
 report (*dicee.executer.Execute attribute*), 72
 report (*dicee.trainer.DICE_Trainer attribute*), 208
 report (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 203
 reports (*dicee.callbacks.Eval attribute*), 24
 reports (*dicee.callbacks.PeriodicEvalCallback attribute*), 26
 requires_grad_for_interactions (*dicee.models.CKeci attribute*), 171
 requires_grad_for_interactions (*dicee.models.clifford.CKeci attribute*), 101
 requires_grad_for_interactions (*dicee.models.clifford.Keci attribute*), 99
 requires_grad_for_interactions (*dicee.models.Keci attribute*), 168
 resid_dropout (*dicee.models.transformers.SelfAttention attribute*), 128
 residual_convolution () (*dicee.models.AConEx method*), 154
 residual_convolution () (*dicee.models.AConvO method*), 167
 residual_convolution () (*dicee.models.AConvQ method*), 161
 residual_convolution () (*dicee.models.complex.AConEx method*), 106
 residual_convolution () (*dicee.models.complex.ConEx method*), 105
 residual_convolution () (*dicee.models.ConEx method*), 154
 residual_convolution () (*dicee.models.ConvO method*), 167
 residual_convolution () (*dicee.models.ConvQ method*), 161
 residual_convolution () (*dicee.models.octonion.AConvO method*), 117
 residual_convolution () (*dicee.models.octonion.ConvO method*), 117
 residual_convolution () (*dicee.models.quaternion.AConvQ method*), 122
 residual_convolution () (*dicee.models.quaternion.ConvQ method*), 121
 retrieve_embedding () (*dicee.scripts.index_serve.NeuralSearcher method*), 194
 retrieve_embeddings () (in module *dicee.scripts.index_serve*), 194
 return_multi_hop_query_results () (*dicee.KGE method*), 219
 return_multi_hop_query_results () (*dicee.knowledge_graph_embeddings.KGE method*), 79
 root () (in module *dicee.scripts.index_serve*), 194
 roots (*dicee.models.FMult attribute*), 181
 roots (*dicee.models.function_space.FMult attribute*), 110
 roots (*dicee.models.function_space.GFMult attribute*), 110
 roots (*dicee.models.GFMult attribute*), 181
 runtime (*dicee.analyse_experiments.Experiment attribute*), 20

S

sample () (*dicee.weight_averaging.SWAG method*), 212

sample_counter (*dicee.abstracts.AbstractPPECallback* attribute), 18
 sample_entity() (*dicee.abstracts.BaseInteractiveKGE* method), 15
 sample_relation() (*dicee.abstracts.BaseInteractiveKGE* method), 15
 sample_triples_ratio (*dicee.config.Namespace* attribute), 30
 sample_triples_ratio (*dicee.knowledge_graph.KG* attribute), 75
 sample_weights() (*dicee.weight_averaging.TWA* method), 214
 sampling_ratio (*dicee.dataset_classes.LiteralDataset* attribute), 43, 44
 sanity_check_callback_args() (in module *dicee.sanity_checkers*), 193
 sanity_checking_with_arguments() (in module *dicee.sanity_checkers*), 193
 save() (*dicee.abstracts.BaseInteractiveKGE* method), 15
 save() (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk* method), 192
 save() (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk* method), 187
 save_checkpoint() (*dicee.abstracts.AbstractTrainer* static method), 14
 save_checkpoint_model() (in module *dicee.static_funcs*), 198
 save_embeddings() (in module *dicee.static_funcs*), 199
 save_embeddings_as_csv (*dicee.config.Namespace* attribute), 28
 save_every_n_epochs (*dicee.config.Namespace* attribute), 31
 save_experiment() (*dicee.analyse_experiments.Experiment* method), 20
 save_model_at_every_epoch (*dicee.config.Namespace* attribute), 29
 save_model_every_n_epoch (*dicee.callbacks.PeriodicEvalCallback* attribute), 26
 save_numpy_ndarray() (in module *dicee.read_preprocess_save_load_kg.util*), 191
 save_numpy_ndarray() (in module *dicee.static_funcs*), 198
 save_pickle() (in module *dicee.read_preprocess_save_load_kg.util*), 191
 save_pickle() (in module *dicee.static_funcs*), 197
 save_queries() (*dicee.query_generator.QueryGenerator* method), 186
 save_queries() (*dicee.QueryGenerator* method), 221
 save_queries_and_answers() (*dicee.query_generator.QueryGenerator* static method), 186
 save_queries_and_answers() (*dicee.QueryGenerator* static method), 221
 save_trained_model() (*dicee.Execute* method), 216
 save_trained_model() (*dicee.executer.Execute* method), 72
 scalar_batch_NN() (*dicee.models.function_space.LFMult* method), 112
 scalar_batch_NN() (*dicee.models.LFMult* method), 183
 scaler (*dicee.callbacks.Perturb* attribute), 26
 scaler (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 208
 scheduler (*dicee.callbacks.LRScheduler* attribute), 27
 score() (*dicee.models.clifford.Keci* method), 101
 score() (*dicee.models.CoKE* method), 151
 score() (*dicee.models.ComplEx* static method), 155
 score() (*dicee.models.complex.ComplEx* static method), 107
 score() (*dicee.models.DistMult* method), 149
 score() (*dicee.models.Keci* method), 170
 score() (*dicee.models.octonion.OMult* method), 116
 score() (*dicee.models.OMult* method), 166
 score() (*dicee.models.QMult* method), 160
 score() (*dicee.models.quaternion.QMult* method), 120
 score() (*dicee.models.real.CoKE* method), 124
 score() (*dicee.models.real.DistMult* method), 122
 score() (*dicee.models.real.TransE* method), 123
 score() (*dicee.models.TransE* method), 149
 score_func (*dicee.models.FMult2* attribute), 181
 score_func (*dicee.models.function_space.FMult2* attribute), 111
 scoring_technique (*dicee.analyse_experiments.Experiment* attribute), 20
 scoring_technique (*dicee.config.Namespace* attribute), 29
 search() (*dicee.scripts.index_serve.NeuralSearcher* method), 194
 search_embeddings() (in module *dicee.scripts.index_serve*), 194
 search_embeddings_batch() (in module *dicee.scripts.index_serve*), 194
 seed (*dicee.query_generator.QueryGenerator* attribute), 185
 seed (*dicee.QueryGenerator* attribute), 220
 select_model() (in module *dicee.static_funcs*), 197
 selected_optimizer (*dicee.models.base_model.BaseKGE* attribute), 96
 selected_optimizer (*dicee.models.BaseKGE* attribute), 143, 146, 152, 157, 163, 176, 179
 SelfAttention (class in *dicee.models.transformers*), 127
 separator (*dicee.config.Namespace* attribute), 29
 separator (*dicee.knowledge_graph.KG* attribute), 75
 sequential_vocabulary_construction() (*dicee.read_preprocess_save_load_kg.PreprocessKG* method), 192
 sequential_vocabulary_construction() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 186
 serve() (in module *dicee.scripts.index_serve*), 194
 set_global_seed() (*dicee.query_generator.QueryGenerator* method), 185

`set_global_seed()` (*dicee.QueryGenerator method*), 221
`set_model_eval_mode()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`set_model_train_mode()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`setup()` (*dicee.dataset_classes.CVDataModule method*), 40
`setup_executor()` (*dicee.Execute method*), 216
`setup_executor()` (*dicee.executer.Execute method*), 72
`Shallom` (*class in dicee.models*), 149
`Shallom` (*class in dicee.models.real*), 123
`shallom` (*dicee.models.real.Shallom attribute*), 123
`shallom` (*dicee.models.Shallom attribute*), 149
`single_hop_query_answering()` (*dicee.KGE method*), 219
`single_hop_query_answering()` (*dicee.knowledge_graph_embeddings.KGE method*), 79
`snapshot_dir` (*dicee.callbacks.LRScheduler attribute*), 27
`snapshot_loss` (*dicee.callbacks.LRScheduler attribute*), 27
`sparql_endpoint` (*dicee.config.Namespace attribute*), 28
`sparql_endpoint` (*dicee.knowledge_graph.KG attribute*), 74
`sq_mean` (*dicee.weight_averaging.SWAG attribute*), 212
`start()` (*dicee.DICE_Trainer method*), 222
`start()` (*dicee.Execute method*), 217
`start()` (*dicee.executer.Execute method*), 73
`start()` (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 191
`start()` (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 186
`start()` (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method*), 187
`start()` (*dicee.read_preprocess_save_load_kg.ReadFromDisk method*), 192
`start()` (*dicee.trainer.DICE_Trainer method*), 209
`start()` (*dicee.trainer.dice_trainer.DICE_Trainer method*), 204
`start_time` (*dicee.callbacks.PrintCallback attribute*), 22
`start_time` (*dicee.Execute attribute*), 216
`start_time` (*dicee.executer.Execute attribute*), 72
`state_dict()` (*dicee.models.ensemble.EnsembleKGE method*), 109
`step()` (*dicee.models.ADOPT method*), 135
`step()` (*dicee.models.adopt.ADOPT method*), 84
`step()` (*dicee.models.ensemble.EnsembleKGE method*), 109
`step_count` (*dicee.callbacks.LRScheduler attribute*), 27
`storage_path` (*dicee.config.Namespace attribute*), 28
`storage_path` (*dicee.DICE_Trainer attribute*), 222
`storage_path` (*dicee.trainer.DICE_Trainer attribute*), 208
`storage_path` (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 203
`store()` (*in module dicee.static_funcs*), 198
`store_ensemble()` (*dicee.abstracts.AbstractPPECallback method*), 18
`strategy` (*dicee.abstracts.AbstractTrainer attribute*), 13
`StringListRequest` (*class in dicee.scripts.index_serve*), 194
`SWA` (*class in dicee.weight_averaging*), 210
`swa` (*dicee.config.Namespace attribute*), 30
`swa_c_epochs` (*dicee.config.Namespace attribute*), 31
`swa_c_epochs` (*dicee.weight_averaging.SWA attribute*), 211
`swa_c_epochs` (*dicee.weight_averaging.SWAG attribute*), 212
`swa_lr` (*dicee.weight_averaging.SWA attribute*), 211
`swa_lr` (*dicee.weight_averaging.SWAG attribute*), 212
`swa_model` (*dicee.weight_averaging.SWA attribute*), 211
`swa_n` (*dicee.weight_averaging.SWA attribute*), 211
`swa_start_epoch` (*dicee.config.Namespace attribute*), 31
`swa_start_epoch` (*dicee.weight_averaging.SWA attribute*), 211
`swa_start_epoch` (*dicee.weight_averaging.SWAG attribute*), 212
`SWAG` (*class in dicee.weight_averaging*), 211
`swag` (*dicee.config.Namespace attribute*), 30

T

`T()` (*dicee.models.DualE method*), 184
`T()` (*dicee.models.dualE.DualE method*), 108
`t_conorm()` (*dicee.abstracts.InteractiveQueryDecomposition method*), 16
`t_norm()` (*dicee.abstracts.InteractiveQueryDecomposition method*), 16
`target_dim` (*dicee.dataset_classes.AllvsAll attribute*), 35
`target_dim` (*dicee.dataset_classes.MultiLabelDataset attribute*), 33
`target_dim` (*dicee.dataset_classes.OnevsAllDataset attribute*), 34
`target_dim` (*dicee.knowledge_graph.KG attribute*), 76
`temperature` (*dicee.models.transformers.BytE attribute*), 126

`tensor_t_norm()` (*dicee.abstracts.InteractiveQueryDecomposition method*), 16
`TensorParallel` (class in *dicee.trainer.model_parallelism*), 205
`test_data_loader()` (*dicee.models.base_model.BaseKGELightning method*), 91
`test_data_loader()` (*dicee.models.BaseKGELightning method*), 139
`test_epoch_end()` (*dicee.models.base_model.BaseKGELightning method*), 91
`test_epoch_end()` (*dicee.models.BaseKGELightning method*), 138
`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*), 185
`test_path` (*dicee.QueryGenerator attribute*), 220
`test_set` (*dicee.knowledge_graph.KG attribute*), 74, 75
`timeit()` (in module *dicee.read_preprocess_save_load_kg.util*), 190
`timeit()` (in module *dicee.static_funcs*), 197
`timeit()` (in module *dicee.static_preprocess_funcs*), 202
`to()` (*dicee.KGE method*), 217
`to()` (*dicee.knowledge_graph_embeddings.KGE method*), 76
`to()` (*dicee.models.ensemble.EnsembleKGE method*), 109
`to_df()` (*dicee.analyse_experiments.Experiment method*), 20
`topk` (*dicee.models.transformers.ByE attribute*), 126
`topk` (*dicee.scripts.index_serve.NeuralSearcher attribute*), 194
`torch_ordered_shaped_bpe_entities` (*dicee.dataset_classes.MultiLabelDataset attribute*), 33
`TorchDDPTrainer` (class in *dicee.trainer.torch_trainer_ddp*), 207
`TorchTrainer` (class in *dicee.trainer.torch_trainer*), 205
`total_epochs` (*dicee.callbacks.LRScheduler attribute*), 27
`total_steps` (*dicee.callbacks.LRScheduler attribute*), 27
`train()` (*dicee.abstracts.BaseInteractiveTrainKGE method*), 19
`train()` (*dicee.trainer.torch_trainer_ddp.NodeTrainer method*), 208
`train_data` (*dicee.dataset_classes.AllvsAll attribute*), 35
`train_data` (*dicee.dataset_classes.KvsAll attribute*), 34
`train_data` (*dicee.dataset_classes.KvsSampleDataset attribute*), 38
`train_data` (*dicee.dataset_classes.MultiClassClassificationDataset attribute*), 33
`train_data` (*dicee.dataset_classes.OnevsAllDataset attribute*), 34
`train_data` (*dicee.dataset_classes.OnevsSample attribute*), 36
`train_data_loader()` (*dicee.dataset_classes.CVDDataModule method*), 40
`train_data_loader()` (*dicee.models.base_model.BaseKGELightning method*), 93
`train_data_loader()` (*dicee.models.BaseKGELightning method*), 140
`train_data_loaders` (*dicee.trainer.torch_trainer.TorchTrainer attribute*), 206
`train_dataset_loader` (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 207
`train_file_path` (*dicee.dataset_classes.LiteralDataset attribute*), 43, 44
`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*), 33
`train_k_vs_all()` (*dicee.abstracts.BaseInteractiveTrainKGE method*), 18
`train_literals()` (*dicee.abstracts.BaseInteractiveTrainKGE method*), 19
`train_mode` (*dicee.models.ensemble.EnsembleKGE attribute*), 109
`train_mrr` (*dicee.analyse_experiments.Experiment attribute*), 20
`train_path` (*dicee.query_generator.QueryGenerator attribute*), 185
`train_path` (*dicee.QueryGenerator attribute*), 220
`train_set` (*dicee.dataset_classes.BPE_NegativeSamplingDataset attribute*), 32
`train_set` (*dicee.dataset_classes.MultiLabelDataset attribute*), 33
`train_set` (*dicee.dataset_classes.NegSampleDataset attribute*), 38
`train_set` (*dicee.dataset_classes.TriplePredictionDataset attribute*), 39
`train_set` (*dicee.knowledge_graph.KG attribute*), 74, 75
`train_set_idx` (*dicee.dataset_classes.CVDDataModule attribute*), 40
`train_set_target` (*dicee.knowledge_graph.KG attribute*), 75
`train_target` (*dicee.dataset_classes.AllvsAll attribute*), 35
`train_target` (*dicee.dataset_classes.KvsAll attribute*), 35
`train_target` (*dicee.dataset_classes.KvsSampleDataset attribute*), 38
`train_target_indices` (*dicee.knowledge_graph.KG attribute*), 76
`train_triples` (*dicee.dataset_classes.NegSampleDataset attribute*), 38
`train_triples()` (*dicee.abstracts.BaseInteractiveTrainKGE method*), 18
`trained_model` (*dicee.Execute attribute*), 215, 216
`trained_model` (*dicee.executer.Execute attribute*), 72
`trainer` (*dicee.config.Namespace attribute*), 29
`trainer` (*dicee.DICE_Trainer attribute*), 222

trainer (*dicее.Execute attribute*), 215, 216
 trainer (*dicее.executer.Execute attribute*), 71, 72
 trainer (*dicее.trainer.DICE_Trainer attribute*), 208
 trainer (*dicее.trainer.dice_trainer.DICE_Trainer attribute*), 203
 trainer (*dicее.trainer.torch_trainer_ddp.NodeTrainer attribute*), 207
 training_step (*dicее.trainer.torch_trainer.TorchTrainer attribute*), 206
 training_step() (*dicее.models.base_model.BaseKGELightning method*), 90
 training_step() (*dicее.models.BaseKGELightning method*), 137
 training_step() (*dicее.models.transformers.BytE method*), 126
 training_step_outputs (*dicее.models.base_model.BaseKGELightning attribute*), 89
 training_step_outputs (*dicее.models.BaseKGELightning attribute*), 137
 training_technique (*dicее.knowledge_graph.KG attribute*), 75
 TransE (*class in dicее.models*), 149
 TransE (*class in dicее.models.real*), 122
 transfer_batch_to_device() (*dicее.dataset_classes.CVDDataModule method*), 41
 transformer (*dicее.models.transformers.BytE attribute*), 126
 transformer (*dicее.models.transformers.GPT attribute*), 131
 trapezoid() (*dicее.models.FMult2 method*), 182
 trapezoid() (*dicее.models.function_space.FMult2 method*), 111
 tri_score() (*dicее.models.function_space.LFMult method*), 112
 tri_score() (*dicее.models.function_space.LFMult1 method*), 111
 tri_score() (*dicее.models.LFMult method*), 183
 tri_score() (*dicее.models.LFMult1 method*), 182
 triple_score() (*dicее.KGE method*), 219
 triple_score() (*dicее.knowledge_graph_embeddings.KGE method*), 78
 TriplePredictionDataset (*class in dicее.dataset_classes*), 38
 tuple2list() (*dicее.query_generator.QueryGenerator method*), 185
 tuple2list() (*dicее.QueryGenerator method*), 221
 TWA (*class in dicее.weight_averaging*), 213
 twa (*dicее.config.Namespace attribute*), 30
 twa_c_epochs (*dicее.weight_averaging.TWA attribute*), 214
 twa_model (*dicее.weight_averaging.TWA attribute*), 214
 twa_start_epoch (*dicее.weight_averaging.TWA attribute*), 214

U

unlabelled_size (*dicее.callbacks.PseudoLabellingCallback attribute*), 24
 unmap() (*dicее.query_generator.QueryGenerator method*), 185
 unmap() (*dicее.QueryGenerator method*), 221
 unmap_query() (*dicее.query_generator.QueryGenerator method*), 185
 unmap_query() (*dicее.QueryGenerator method*), 221
 update_hits() (*in module dicее.evaluation*), 67
 update_hits() (*in module dicее.evaluation.utils*), 58

V

val_aswa (*dicее.weight_averaging.ASWA attribute*), 210
 val_dataloader() (*dicее.models.base_model.BaseKGELightning method*), 92
 val_dataloader() (*dicее.models.BaseKGELightning method*), 139
 val_h1 (*dicее.analyse_experiments.Experiment attribute*), 20
 val_h3 (*dicее.analyse_experiments.Experiment attribute*), 20
 val_h10 (*dicее.analyse_experiments.Experiment attribute*), 20
 val_mrr (*dicее.analyse_experiments.Experiment attribute*), 20
 val_path (*dicее.query_generator.QueryGenerator attribute*), 185
 val_path (*dicее.QueryGenerator attribute*), 220
 VALID_SCORING_TECHNIQUES (*in module dicее.evaluation.evaluator*), 50
 valid_set (*dicее.knowledge_graph.KG attribute*), 74, 75
 validate_knowledge_graph() (*in module dicее.sanity_checkers*), 193
 var_clamp (*dicее.weight_averaging.SWAG attribute*), 212
 vocab_preparation() (*dicее.evaluation.Evaluator method*), 60
 vocab_preparation() (*dicее.evaluation.evaluator.Evaluator method*), 51
 vocab_preparation() (*dicее.Evaluator method*), 224
 vocab_preparation() (*dicее.evaluator.Evaluator method*), 69
 vocab_size (*dicее.models.CoKEConfig attribute*), 150
 vocab_size (*dicее.models.real.CoKEConfig attribute*), 123, 124
 vocab_size (*dicее.models.transformers.GPTConfig attribute*), 130
 vocab_to_parquet() (*in module dicее.static_funcs*), 199
 vtp_score() (*dicее.models.function_space.LFMult method*), 112
 vtp_score() (*dicее.models.function_space.LFMult1 method*), 112

`vtp_score()` (*dicее.models.LFMult* method), 183
`vtp_score()` (*dicее.models.LFMultI* method), 182

W

`warmup_steps` (*dicее.callbacks.LRScheduler* attribute), 27
`weight` (*dicее.models.transformers.LayerNorm* attribute), 127
`weight_decay` (*dicее.config.Namespace* attribute), 29
`weight_decay` (*dicее.models.base_model.BaseKGE* attribute), 96
`weight_decay` (*dicее.models.BaseKGE* attribute), 143, 146, 152, 157, 163, 176, 179
`weight_samples` (*dicее.weight_averaging.TWA* attribute), 214
`weights` (*dicее.models.FMult* attribute), 181
`weights` (*dicее.models.function_space.FMult* attribute), 110
`weights` (*dicее.models.function_space.GFMult* attribute), 110
`weights` (*dicее.models.GFMult* attribute), 181
`world_size` (*dicее.Execute* attribute), 215
`world_size` (*dicее.executer.Execute* attribute), 71
`write_csv_from_model_parallel()` (in module *dicее.static_funcs*), 199
`write_links()` (*dicее.query_generator.QueryGenerator* method), 185
`write_links()` (*dicее.QueryGenerator* method), 221
`write_report()` (*dicее.Execute* method), 217
`write_report()` (*dicее.executer.Execute* method), 73

X

`x_values` (*dicее.models.function_space.LFMult* attribute), 112
`x_values` (*dicее.models.LFMult* attribute), 182