DICE Embeddings

Release 0.1.3.2

Caglar Demir

Oct 25, 2024

Contents:

1	Dicee Manual	2
2	Installation 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 6.1 Learning Embeddings	5 6 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 14.1 Subpackages	12
	14.2 Submodules	
	14.3 Attributes	
	14.4 Classes	
	14.5 Functions	
	14.6 Package Contents	

Index 206

DICE Embeddings¹: Hardware-agnostic Framework for Large-scale Knowledge Graph Embeddings:

1 Dicee Manual

Version: dicee 0.1.3.2

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

Publisher and maintainer: Caglar 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. Pandas³ & Co. to use parallelism at preprocessing a large knowledge graph,
- 2. PyTorch⁴ & Co. to learn knowledge graph embeddings via multi-CPUs, GPUs, TPUs or computing cluster, and
- 3. **Huggingface**⁵ to ease the deployment of pre-trained models.

Why 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 PyTorch⁸ & PytorchLightning⁹. Users can choose the trainer class (e.g., DDP by Pytorch) to train large knowledge graph embedding models with billions of parameters. PytorchLightning allows us to use state-of-the-art model parallelism techniques (e.g. Fully Sharded Training, FairScale, or DeepSpeed) without extra effort. With our framework, practitioners can directly use PytorchLightning for model parallelism to train gigantic embedding models.

Why Hugging-face Gradio¹⁰? Deploy a pre-trained embedding model without writing a single line of code.

¹ https://github.com/dice-group/dice-embeddings

² https://github.com/Demirrr

³ https://pandas.pydata.org/

⁴ https://pytorch.org/

⁵ https://huggingface.co/

⁶ https://pandas.pydata.org/

⁷ https://pytorch.org/

⁸ https://pytorch.org/

⁹ https://www.pytorchlightning.ai/

¹⁰ https://huggingface.co/gradio

2 Installation

2.1 Installation from Source

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

or

```
pip install dicee
```

3 Download Knowledge Graphs

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

→certificate && unzip KGs.zip
```

To test the Installation

```
python -m pytest -p no:warnings -x # Runs >114 tests leading to > 15 mins

python -m pytest -p no:warnings --lf # run only the last failed test

python -m pytest -p no:warnings --ff # to run the failures first and then the rest of the tests.
```

4 Knowledge Graph Embedding Models

- 1. TransE, DistMult, ComplEx, ConEx, QMult, OMult, ConvO, ConvQ, Keci
- 2. All 44 models available in https://github.com/pykeen/pykeen#models For more, please refer to examples.

5 How to Train

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

```
from dicee.executer import Execute
from dicee.config import Namespace
args = Namespace()
args.model = 'Keci'
args.scoring_technique = "KvsAll" # 1vsAll, or AllvsAll, or NegSample
args.dataset_dir = "KGs/UMLS"
args.path_to_store_single_run = "Keci_UMLS"
args.num_epochs = 100
args.embedding_dim = 32
args.batch_size = 1024
reports = Execute(args).start()
print(reports["Train"]["MRR"]) # => 0.9912
print(reports["Trest"]["MRR"]) # => 0.8155
# See the Keci_UMLS folder embeddings and all other files
```

where the data is in the following form

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

A KGE model can also be trained from the command line

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

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

```
# Train a model by only using the GPU-0

CUDA_VISIBLE_DEVICES=0 dicee --dataset_dir "KGs/UMLS" --model Keci --eval_model

--"train_val_test"

# Train a model by only using GPU-1

CUDA_VISIBLE_DEVICES=1 dicee --dataset_dir "KGs/UMLS" --model Keci --eval_model

--"train_val_test"

NCCL_P2P_DISABLE=1 CUDA_VISIBLE_DEVICES=0,1 python dicee/scripts/run.py --trainer PL -

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

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

```
# Two equivalent executions
# (1)
dicee --dataset_dir "KGs/UMLS" --model Keci --eval_model "train_val_test"
# Evaluate Keci on Train set: Evaluate Keci on Train set
# {'H01': 0.9518788343558282, 'H03': 0.9988496932515337, 'H010': 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
# {'H01': 0.6951588502269289, 'H03': 0.9039334341906202, 'H010': 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
# {'H01': 0.9518788343558282, 'H03': 0.9988496932515337, 'H010': 1.0, 'MRR': 0.
\leftrightarrow 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

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 --
wmodel Keci --p 0 --q 1 --embedding_dim 32 --adaptive_swa
```

6.2 Loading Embeddings into Qdrant Vector Database

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://0.0.0.8000/api/get?q=germany' -H 'accept: application/json'
```

Get most similar things to europe

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

7 Answering Complex Queries

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

```
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,
\hookrightarrow 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://
\rightarrowwww.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.py¹²:

Name	Stmts	Miss	Cover	Missing
dicee/initpy	7		100%	
dicee/abstracts.py	201	82		104–105, Litinues on next page)

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

¹² https://coverage.readthedocs.io/en/7.6.0/

```
→123, 146-147, 152, 165, 197, 240-254, 257-260, 263-266, 301, 314-317, 320-324, 364-
\Rightarrow375, 390-398, 413, 424-428, 555-575, 581-585, 589-591
dicee/callbacks.py
                                                           245
                                                                  102
\hookrightarrow67-73, 76, 88-93, 98-103, 106-109, 116-133, 138-142, 146-147, 276-280, 286-287, 305-
→311, 314, 319-320, 332-338, 344-353, 358-360, 405, 416-429, 433-468, 480-486
dicee/config.py
                                                            93
                                                                    2
                                                                          98%
                                                                                141-142
dicee/dataset_classes.py
                                                           299
                                                                   74
                                                                          75%
                                                                                41, 54, ...
→87, 93, 99-106, 109, 112, 115-139, 195-201, 204, 207-209, 314, 325-328, 344, 410-

→411, 429, 528-536, 539, 543-557, 700-707, 710-714

dicee/eval_static_funcs.py
                                                           227
                                                                    95
                                                                          58%
                                                                                101, 106,
→ 111, 258-353, 360-411
dicee/evaluator.py
                                                           262
                                                                   51
                                                                          81%
                                                                                46, 51,_
→56, 84, 89-90, 93, 109, 126, 137, 141, 146, 177-188, 195-206, 314, 344-367, 455, □
→465, 482-487
dicee/executer.py
                                                                          96%
                                                                                116, 258-
                                                           113
⇒259, 291
dicee/knowledge_graph.py
                                                            65
                                                                    3
                                                                          95%
                                                                                79, 110, _
⇔114
dicee/knowledge_graph_embeddings.py
                                                           636
                                                                  443
                                                                          30%
                                                                                27, 30-
→31, 39-52, 57-90, 93-127, 131-139, 170-184, 215-228, 254-274, 324-327, 330-333, 346,
→ 381-426, 484-486, 502-503, 509-517, 522-525, 528-533, 538, 547, 592-598, 630, 688-
→1053, 1084-1145, 1149-1177, 1200, 1227-1265
dicee/models/__init__.py
                                                             9
                                                                         100%
                                                           234
                                                                   31
                                                                          87%
dicee/models/base_model.py
                                                                                54, 56, ...
→82, 88-103, 157, 190, 230, 236, 245, 248, 252, 259, 263, 265, 280, 288-289, 296-297,
→ 351, 354, 427, 439
dicee/models/clifford.py
                                                                  357
→68-117, 122-133, 156-168, 190-220, 235, 237, 241, 248-249, 276-280, 303-311, 325-
→327, 332-333, 364-384, 406, 413, 417-478, 495-499, 511, 514, 519, 524, 571-607, 625-
→631, 644, 647, 652, 657, 686-692, 705, 708, 713, 718, 728-737, 753-754, 774-845, □
→856-859, 884-909, 933-966, 1002-1006, 1019, 1029, 1032, 1037, 1042, 1047, 1051, □
→1055, 1064-1065, 1095, 1102, 1107, 1135-1139, 1167-1176, 1186-1194, 1212-1214, 1232-
→1234, 1250-1252
dicee/models/complex.py
                                                           151
                                                                   15
                                                                          90%
                                                                                86-109
dicee/models/dualE.py
                                                            59
                                                                   10
                                                                          83%
                                                                                93-102,_
→142-156
                                                           262
                                                                  221
dicee/models/function_space.py
                                                                          16%
                                                                                10-24, _
\Rightarrow28-37, 40-49, 53-70, 77-86, 89-98, 101-110, 114-126, 134-156, 159-165, 168-185, 188-
→194, 197-205, 208, 213-234, 243-246, 250-254, 258-267, 271-292, 301-307, 311-328, □
→332-335, 344-352, 355, 366-372, 392-406, 424-438, 443-453, 461-465, 474-478
                                                           227
                                                                   83
                                                                          63%
dicee/models/octonion.py
                                                                                21-44,_
\Rightarrow320-329, 334-345, 348-370, 374-416, 426-474
dicee/models/pykeen_models.py
                                                            50
                                                                    5
                                                                          90%
                                                                                60-63, _
dicee/models/quaternion.py
                                                                                7-21, 30-
                                                           192
                                                                    69
                                                                          64%
→55, 68-72, 107, 185, 328-342, 345-364, 368-389, 399-426
dicee/models/real.py
                                                            61
                                                                   12
                                                                          80%
                                                                                36-39, _
\leftrightarrow 66-69, 87, 103-106
dicee/models/static_funcs.py
                                                            10
                                                                    0
                                                                         100%
dicee/models/transformers.py
                                                           236
                                                                   189
\hookrightarrow46, 60-75, 84-102, 105-116, 123-125, 128, 134-151, 155-180, 186-190, 193-197, 203-
→207, 210-212, 229-256, 265-268, 271-276, 279-304, 310-315, 319-372, 376-398, 404-414
```

```
dicee/query_generator.py
                                                              374
                                                                      346
                                                                               7%
                                                                                    18-52,_
\hookrightarrow56, 62-65, 69-70, 78-92, 100-147, 155-188, 192-206, 212-269, 274-303, 307-443, 453-
\hookrightarrow472, 480-501, 508-512, 517, 522-528
                                                                3
                                                                        0
                                                                            100%
dicee/read_preprocess_save_load_kg/__init__.py
dicee/read_preprocess_save_load_kg/preprocess.py
                                                              256
                                                                       41
                                                                             84%
                                                                                    34, 40, _
\hookrightarrow78, 102-127, 133, 138-151, 184, 214, 388-389, 444
dicee/read_preprocess_save_load_kg/read_from_disk.py
                                                               36
                                                                       11
                                                                             69%
                                                                                    33, 38-
\hookrightarrow40, 47, 55, 58-72
dicee/read_preprocess_save_load_kg/save_load_disk.py
                                                               45
                                                                       18
                                                                             60%
                                                                                    39-60
dicee/read_preprocess_save_load_kg/util.py
                                                              219
                                                                      126
                                                                             42%
                                                                                    65-67.
→72-73, 91-97, 100-102, 107-109, 121, 134, 140-143, 148-156, 161-167, 172-177, 182-
→187, 199-220, 226-282, 286-290, 294-295, 299, 303-304, 334, 351, 356, 363-364
                                                                       23
                                                                             57%
dicee/sanity_checkers.py
                                                               54
                                                                                    8-12, 21-
\rightarrow31, 46, 51, 58, 64-79, 85, 89, 96
dicee/static_funcs.py
                                                                      163
                                                                             61%
                                                                                    40, 50, _
                                                              418
→56-61, 83, 105-106, 115, 138, 152, 157-159, 163-165, 167, 194-198, 246, 254, 263-
→268, 290-304, 316-336, 340-357, 362, 386-387, 392-393, 410-411, 413-414, 416-417, □
→419-420, 428, 446-450, 467-470, 474-479, 483-487, 491-492, 498-500, 526-527, 539-
\hookrightarrow 542, 547-550, 559-610, 615-627, 644-658, 661-669
dicee/static_funcs_training.py
                                                              123
                                                                       63
                                                                             49%
                                                                                    118-215, _
⇔223-224
dicee/static_preprocess_funcs.py
                                                              100
                                                                       44
                                                                             56%
                                                                                    17-25.
\hookrightarrow 52, 56, 64, 67, 78, 91-115, 120-123, 128-131, 136-139
dicee/trainer/__init__.py
                                                                        0
                                                                            100%
                                                                1
dicee/trainer/dice_trainer.py
                                                              126
                                                                       13
                                                                             90%
                                                                                    27-32, _
\hookrightarrow 91, 98, 103-108, 147
dicee/trainer/torch_trainer.py
                                                               79
                                                                             95%
                                                                                    31, 196, _
→207-208
dicee/trainer/torch_trainer_ddp.py
                                                              152
                                                                      128
                                                                             16%
                                                                                    13-14,_
→43, 47-72, 83-112, 131-137, 140-149, 164-194, 204-217, 226-246, 251-260, 263-272, □
⇒275-299, 302-309
TOTAL
                                                             6181
                                                                     2828
                                                                             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:)

```
@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,
                   {Convolutional Hypercomplex Embeddings for Link Prediction},
 title =
                 {Demir, Caglar and Moussallem, Diego and Heindorf, Stefan and Ngonga
 author =
→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 \text{ Nov}\},
 publisher =
                 {PMLR},
                 {https://proceedings.mlr.press/v157/demir21a/demir21a.pdf},
 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

14.1 Subpackages

dicee.models

Submodules

dicee.models.base_model

Classes

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

Module Contents

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

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
\label{eq:continuous_step_outputs} \begin{tabular}{ll} \textbf{training\_step\_outputs} &= & [\ ] \\ \\ \textbf{mem\_of\_model} \ () &\to Dict \\ \\ \textbf{Size of model in MB and number of params} \\ \end{tabular}
```

Size of model in MD and number of param

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()
```

1 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 Light-ningModule 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])

```
\texttt{test\_dataloader}\,(\,)\,\to 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()



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

1 Note

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

${\tt val_dataloader}\,()\,\to None$

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

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

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

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

1 Note

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

1 Note

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

$predict_dataloader() \rightarrow None$

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()

1 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.

$train_dataloader() \rightarrow None$

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()

1 Note

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

configure_optimizers (parameters=None)

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

Returns

Any of these 6 options.

- · Single optimizer.
- List or Tuple of optimizers.
- Two lists The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_scheduler_config).
- Dictionary, with an "optimizer" key, and (optionally) a "lr_scheduler" key whose value is a single LR scheduler or lr_scheduler_config.
- None Fit will run without any optimizer.

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

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

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

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



1 Note

Some things to know:

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

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

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

args

```
embedding_dim = None
num_entities = None
num_relations = None
```

```
num_tokens = None
learning_rate = None
apply_unit_norm = None
input_dropout_rate = None
hidden_dropout_rate = None
optimizer_name = None
feature_map_dropout_rate = None
kernel_size = None
num_of_output_channels = None
weight_decay = None
loss
selected_optimizer = None
normalizer_class = None
normalize_head_entity_embeddings
normalize_relation_embeddings
normalize_tail_entity_embeddings
hidden_normalizer
param_init
input_dp_ent_real
input_dp_rel_real
hidden_dropout
loss_history = []
byte_pair_encoding
max_length_subword_tokens
block_size
forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)
        Parameters
           x (B x 2 x T)
forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors
        Parameters
```

```
init_params_with_sanity_checking()
     forward (x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
                 y idx: torch.LongTensor = None)
              Parameters
                  • x
                  y_idx
                  • ordered_bpe_entities
     \texttt{forward\_triples} \ (x: torch.LongTensor) \ \to torch.Tensor
              Parameters
     forward_k_vs_all(*args, **kwargs)
     forward_k_vs_sample(*args, **kwargs)
     get_triple_representation(idx_hrt)
     get_head_relation_representation(indexed_triple)
     get_sentence_representation(x: torch.LongTensor)
              Parameters
                  • (b (x shape)
                  • 3
                  • t)
     get_bpe_head_and_relation_representation(x: torch.LongTensor)
                 → Tuple[torch.FloatTensor, torch.FloatTensor]
              Parameters
                 x (B x 2 x T)
     get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]
class dicee.models.base_model.IdentityClass(args=None)
     Bases: torch.nn.Module
```

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to(), etc.



1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
args
\_\_call\_\_(x)
static forward (x)
```

dicee.models.clifford

Classes

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

Module Contents

```
class dicee.models.clifford.Keci(args)
    Bases: dicee.models.base_model.BaseKGE
```

Base class for all neural network modules.

Your models should also subclass this class.

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

```
import torch.nn as nn
import torch.nn.functional as F
class Model (nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to (), etc.



As per the example above, an <u>__init__</u>() call to the parent class must be made before assignment on the child.

Variables

name = 'Keci'

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

```
r
requires_grad_for_interactions = True
compute_sigma_pp (hp, rp)
   Compute sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k
   sigma_{pp} captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute
   interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops
   results = [] for i in range(p - 1):
        for k in range(i + 1, p):
        results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])
```

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

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

```
e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
```

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

```
compute\_sigma\_qq(hq, rq)
```

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

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

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

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

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

```
e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
```

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

 $\texttt{compute_sigma_pq} \, (\,^*\!, \, hp, \, hq, \, rp, \, rq)$

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

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

 ${\tt clifford_multiplication}\,(h0,\,hp,\,hq,\,r0,\,rp,\,rq)$

Compute our CL multiplication

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

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

eq j

$$h r = sigma_0 + sigma_p + sigma_q + sigma_{pp} + sigma_{q} + sig$$

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

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

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

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

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

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

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

→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

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

Parameter

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

returns

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

forward_k_vs_with_explicit(x: torch.Tensor)

k_vs_all_score (bpe_head_ent_emb, bpe_rel_ent_emb, E)

 $forward_k_vs_all(x: torch.Tensor) \rightarrow torch.FloatTensor$

Kvsall training

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

- (2) Construct head entity and relation embeddings according to $Cl_{p,q}(\mathbf{mathbb}_{R}^d)$.
- (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, |E|) shape

 $\begin{tabular}{ll} \begin{tabular}{ll} \beg$

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)

 $forward_k_vs_sample(x: torch.LongTensor, target_entity_idx: torch.LongTensor) \rightarrow torch.FloatTensor$

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

 $\verb"score"\,(h,\,r,\,t)$

 $forward_triples(x: torch.Tensor) \rightarrow torch.FloatTensor$

Parameter

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

rtype

torch.FloatTensor with (n) shape

class dicee.models.clifford.KeciBase(args)

Bases: Keci

Without learning dimension scaling

name = 'KeciBase'

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.



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

Variables

training $(b \circ o 1)$ – 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) \longrightarrow 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,

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

and return:

$$sigma_0t = \sigma_0 \cdot t_0 = s0 + s1 - s2s3, s4ands5$$

compute_sigmas_multivect(list_h_emb, list_r_emb)

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

For same bases vectors interaction we have

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

For different base vector interactions, we have

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

 $\textbf{forward_k_vs_all} \ (\textit{x: torch.Tensor}) \ \rightarrow \text{torch.FloatTensor}$

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations
- (2) Construct head entity and relation embeddings according to $Cl_{p,q}$, r_{mathbb} (R) d .
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

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

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

Multiplying a base vector with its scalar coefficient

construct_cl_multivector (x: torch.FloatTensor, re: int, p: int, q: int, r: int)

→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

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

Parameter

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

returns

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

 $compute_sigma_pp(hp, rp)$

Compute .. math:

sigma_{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):

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

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

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[:, :,
$$i$$
] * rq[:, :, k] - hq[:, :, k] * rq[:, :, i])

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

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

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)
```

dicee.models.complex

Classes

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

```
class dicee.models.complex.ConEx(args)
    Bases: dicee.models.base_model.BaseKGE
    Convolutional ComplEx Knowledge Graph Embeddings
    name = 'ConEx'
    conv2d
    fc_num_input
    fc1
    norm_fc1
    bn_conv2d
    feature_map_dropout
```

```
residual_convolution(C_1: Tuple[torch.Tensor, torch.Tensor],
                  C 2: Tuple[torch.Tensor, torch.Tensor]) \rightarrow 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) \rightarrow torch.FloatTensor
     forward\_triples(x: torch.Tensor) \rightarrow torch.FloatTensor
               Parameters
     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]) \rightarrow 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) \rightarrow torch.FloatTensor
     forward\_triples(x: torch.Tensor) \rightarrow torch.FloatTensor
               Parameters
     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 to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F
class Model (nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – 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

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

```
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
                     {\tt 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\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e\_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_t,e_4\_
                                                                    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) \rightarrow \text{torch.tensor}
                                        KvsAll scoring function
                                        Input
                                        x: torch.LongTensor with (n, ) shape
                                         Output
                                         torch.FloatTensor with (n) shape
                     forward\_triples(idx\_triple: torch.tensor) \rightarrow 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) \rightarrow torch.tensor
                                        Transpose function
                                         Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)
```

dicee.models.function space

Classes

FMult	Learning Knowledge Neural Graphs
GFMult	Learning Knowledge Neural Graphs
FMult2	Learning Knowledge Neural Graphs
LFMult1	Embedding with trigonometric functions. We represent all entities and relations in the complex number space as:
LFMult	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:

```
class dicee.models.function_space.FMult(args)
      Bases: dicee.models.base_model.BaseKGE
      Learning Knowledge Neural Graphs
      name = 'FMult'
      entity_embeddings
      relation_embeddings
      num_sample = 50
      gamma
      roots
      weights
      \verb|compute_func| (\textit{weights: torch.FloatTensor}, \textit{x}) \rightarrow \textit{torch.FloatTensor}
      chain_func(weights, x: torch.FloatTensor)
      \textbf{forward\_triples} \ (\textit{idx\_triple: torch.Tensor}) \ \rightarrow \textbf{torch.Tensor}) \ \rightarrow \textbf{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
      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
class dicee.models.function_space.FMult2(args)
     Bases: dicee.models.base_model.BaseKGE
     Learning Knowledge Neural Graphs
     name = 'FMult2'
     n_{ayers} = 3
     tuned_embedding_dim = False
     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
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=0}^{k=d-1}wk e^{kix}, and use the three differents scoring function as in the paper to evaluate
     the score
     name = 'LFMult1'
     entity_embeddings
```

```
relation_embeddings
     forward_triples (idx_triple)
               Parameters
                   x
     tri_score(h, r, t)
     \mathtt{vtp\_score}(h, r, t)
class dicee.models.function_space.LFMult(args)
     Bases: dicee.models.base_model.BaseKGE
     Embedding with polynomial functions. We represent all entities and relations in the polynomial space as: f(x) =
     sum_{i=0}^{d-1} a_k x^{i\%d} and use the three differents scoring function as in the paper to evaluate the score.
     We also consider combining with Neural Networks.
     name = 'LFMult'
     entity_embeddings
     relation_embeddings
     degree
     m
     x values
     forward_triples (idx_triple)
               Parameters
                   x
     construct_multi_coeff(X)
     poly_NN(x, coefh, coefr, coeft)
           Constructing a 2 layers NN to represent the embeddings. h = sigma(wh^T x + bh), r = sigma(wr^T x + br),
           t = sigma(wt^T x + bt)
     linear(x, w, b)
     scalar_batch_NN(a, b, c)
           element wise multiplication between a,b and c: Inputs: a, b, c ====> torch.tensor of size batch_size x m x
           d Output: a tensor of size batch size x d
     tri_score (coeff_h, coeff_r, coeff_t)
           this part implement the trilinear scoring techniques:
           score(h,r,t) = int_{0}{1} h(x)r(x)t(x) dx = sum_{i,j,k} = 0}^{d-1} dfrac{a_i*b_j*c_k}{1+(i+j+k)%d}
            1. generate the range for i, j and k from [0 d-1]
           2. perform dfrac\{a_i*b_j*c_k\}\{1+(i+j+k)\%d\} in parallel for every batch
            3. take the sum over each batch
```

```
\mathtt{vtp\_score}(h, r, t)
```

this part implement the vector triple product scoring techniques:

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

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

```
comp\_func(h, r, t)
```

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

```
polynomial(coeff, x, degree)
```

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

$$coeff[1][0] + coeff[1][1]x + ... + coeff[1][d]x^d$$

```
pop (coeff, x, degree)
```

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

and return a tensor (coeff[0][0] + coeff[0][1]
$$x$$
 +...+ coeff[0][d] x ^d, coeff[1][0] + coeff[1][1] x +...+ coeff[1][d] x ^d)

dicee.models.octonion

Classes

OMult	Base class for all neural network modules.
Conv0	Base class for all neural network modules.
AConvO	Additive Convolutional Octonion Knowledge Graph Embeddings

Functions

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

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

As per the example above, an <u>__init__()</u> call to the parent class must be made before assignment on the child.

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to (), etc.

1 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 = 'Conv0'

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) \rightarrow 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 l)
```

dicee.models.pykeen_models

Classes

PykeenKGE	A class for using knowledge graph embedding models im-
	plemented 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:

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:

 $h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.embeddin$ 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) \rightarrow torch.FloatTensor

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

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

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

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

abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)

dicee.models.quaternion

Classes

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

Functions

quaternion_mul_with_unit_norm(*, Q_1, Q_2)

Module Contents

```
dicee.models.quaternion.quaternion_mul_with_unit_norm(*, Q_1, Q_2)
class dicee.models.quaternion.QMult(args)
Bases: dicee.models.base_model.BaseKGE
```

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

As per the example above, an <u>__init__()</u> call to the parent class must be made before assignment on the child.

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
\label{eq:name} \begin{tabular}{ll} name = 'QMult' \\ \\ explicit = True \\ \\ quaternion_multiplication_followed_by_inner_product $(h,r,t)$ \\ \\ \end{tabular}
```

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:

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

Parameters

 \mathbf{x} – The vector.

Returns

The normalized vector.

 $k_vs_all_score$ (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
      {\tt residual\_convolution}\,(Q\_1,\,Q\_2)
      \textbf{forward\_triples} \ (\textit{indexed\_triple: torch.Tensor}) \ \rightarrow \textbf{torch.Tensor}) \ \rightarrow \textbf{torch.Tensor}
                Parameters
      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 l)
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) \rightarrow 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,|
```

dicee.models.real

Entities l)

Classes

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

Module Contents

```
class dicee.models.real.DistMult(args)
      Bases: dicee.models.base_model.BaseKGE
      Embedding Entities and Relations for Learning and Inference in Knowledge Bases https://arxiv.org/abs/1412.6575
      name = 'DistMult'
      k vs all score (emb h: torch.FloatTensor, emb r: torch.FloatTensor, emb E: torch.FloatTensor)
               Parameters
                    • emb h
                    • emb_r
                    • emb_E
      forward_k_vs_all (x: torch.LongTensor)
      forward_k_vs_sample (x: torch.LongTensor, target_entity_idx: torch.LongTensor)
      score(h, r, t)
class dicee.models.real.TransE(args)
      Bases: dicee.models.base_model.BaseKGE
      Translating Embeddings for Modeling Multi-relational Data https://proceedings.neurips.cc/paper/2013/file/
      1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf
      name = 'TransE'
     margin = 4
      score (head_ent_emb, rel_ent_emb, tail_ent_emb)
      forward_k_vs_all(x: torch.Tensor) \rightarrow 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_width
      shallom
      \mathtt{get\_embeddings}() \rightarrow \mathsf{Tuple}[\mathsf{numpy}.\mathsf{ndarray}, \mathsf{None}]
      \mathbf{forward\_k\_vs\_all}\;(x)\;\to \mathrm{torch.FloatTensor}
      forward_triples (x) \rightarrow \text{torch.FloatTensor}
               Parameters
               Returns
```

```
class dicee.models.real.Pyke(args)
    Bases: dicee.models.base_model.BaseKGE
    A Physical Embedding Model for Knowledge Graphs
    name = 'Pyke'
    dist_func
    margin = 1.0
    forward_triples(x: torch.LongTensor)
        Parameters
        x
```

dicee.models.static_funcs

Functions

```
quaternion\_mul( \rightarrow Tuple[torch.Tensor, torch.Tensor, \\ Perform quaternion multiplication \\ ...)
```

Module Contents

```
\label{eq:dicee.models.static_funcs.quaternion_mul} (*, Q_1, Q_2) \\ \rightarrow Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor] \\ Perform quaternion multiplication :param Q_1: :param Q_2: :return:
```

dicee.models.transformers

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

Classes

BytE	Base class for all neural network modules.
LayerNorm	LayerNorm but with an optional bias. PyTorch doesn't support simply bias=False
CausalSelfAttention	Base class for all neural network modules.
MLP	Base class for all neural network modules.
Block	Base class for all neural network modules.
GPTConfig	
GPT	Base class for all neural network modules.

Module Contents

```
class dicee.models.transformers.BytE(*args, **kwargs)
Bases: dicee.models.base_model.BaseKGE
```

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
name = 'BytE'
config
temperature = 0.5
topk = 2
transformer
lm_head
weight
loss_function(yhat_batch, y_batch)
```

Parameters

- yhat_batch
- y_batch

forward(x: torch.LongTensor)

Parameters

```
\mathbf{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
```

(continues on next page)

(continued from previous page)

```
opt2.step()
```

1 Note

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

class dicee.models.transformers.LayerNorm(ndim, bias)

Bases: torch.nn.Module

LayerNorm but with an optional bias. PyTorch doesn't support simply bias=False

weight

bias

forward(input)

class dicee.models.transformers.CausalSelfAttention(config)

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
c_attn
c_proj
attn_dropout
resid_dropout
n_head
n_embd
dropout
flash
forward(x)

class dicee.models.transformers.MLP(config)
Bases: torch.nn.Module
```

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

training $(b \circ \circ 1)$ – 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
```

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

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

Variables

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

```
ln_1
attn
ln_2
mlp
forward(x)

class dicee.models.transformers.GPTConfig
block_size: int = 1024
vocab_size: int = 50304
```

```
n_layer: int = 12

n_head: int = 12

n_embd: int = 768

dropout: float = 0.0

bias: bool = False

class dicee.models.transformers.GPT(config)
    Bases: torch.nn.Module
```

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

training $(b \circ \circ 1)$ – Boolean represents whether this module is in training or evaluation mode.

config

transformer

lm_head

weight

```
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.

Classes

BaseKGELightning	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
	Base class for all neural network modules.
IdentityClass	Base class for all neural network modules.
BaseKGE	
DistMult	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
TransE	Translating Embeddings for Modeling
Shallom	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
Pyke	A Physical Embedding Model for Knowledge Graphs
BaseKGE	Base class for all neural network modules.
ConEx	Convolutional ComplEx Knowledge Graph Embeddings
AConEx	Additive Convolutional ComplEx Knowledge Graph Embeddings
ComplEx	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
IdentityClass	Base class for all neural network modules.
OMult	Base class for all neural network modules.
ConvQ	Convolutional Quaternion Knowledge Graph Embeddings
AConvQ	Additive Convolutional Quaternion Knowledge Graph Embeddings
BaseKGE	Base class for all neural network modules.
IdentityClass	Base class for all neural network modules.
OMult	Base class for all neural network modules.
ConvO	Base class for all neural network modules.
AConv0	Additive Convolutional Octonion Knowledge Graph Embeddings
Keci	Base class for all neural network modules.
KeciBase	Without learning dimension scaling
DeCaL	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
PykeenKGE	A class for using knowledge graph embedding models im-
*	plemented in Pykeen
BaseKGE	Base class for all neural network modules.
FMult	Learning Knowledge Neural Graphs
GFMult	Learning Knowledge Neural Graphs
FMult2	Learning Knowledge Neural Graphs
LFMult1	Embedding with trigonometric functions. We represent
111 114 L L	all entities and relations in the complex number space as:

continues on next page

Table 1 - continued from previous page

LFMult	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
DualE	Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)

Functions

```
\begin{array}{ll} \textit{quaternion\_mul}(\rightarrow \text{Tuple[torch.Tensor, torch.Tensor,} & \textit{Perform quaternion multiplication} \\ \textit{...}) \\ \textit{quaternion\_mul\_with\_unit\_norm}(*, Q\_1, Q\_2) \\ \\ \textit{octonion\_mul}(*, O\_1, O\_2) \\ \\ \textit{octonion\_mul\_norm}(*, O\_1, O\_2) \\ \\ \end{array}
```

Package Contents

```
\verb"class dicee.models.BaseKGELightning" (*args, **kwargs")
```

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
\label{eq:training_step_outputs} \textbf{= []} \label{eq:mem_of_model()} \rightarrow \text{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()
```

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 = \dots
        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])

$\texttt{test_dataloader}\,()\,\to None$

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()

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

1 Note

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

${\tt val_dataloader}\,()\,\to None$

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()

1 Note

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

1 Note

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

$\texttt{predict_dataloader}\,() \, \to None$

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()

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.

$\texttt{train_dataloader}\,()\,\to None$

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.

A Warning

do not assign state in prepare_data

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

1 Note

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

configure_optimizers (parameters=None)

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

Returns

Any of these 6 options.

- · Single optimizer.
- List or Tuple of optimizers.
- Two lists The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_scheduler_config).

- Dictionary, with an "optimizer" key, and (optionally) a "lr_scheduler" key whose value is a single LR scheduler or lr_scheduler_config.
- None Fit will run without any optimizer.

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

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

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

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

1 Note

Some things to know:

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

```
class dicee.models.BaseKGE (args: dict)
```

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model (nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

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

args

```
embedding_dim = None
num_entities = None
num_relations = None
num_tokens = None
learning_rate = None
apply_unit_norm = None
input_dropout_rate = None
hidden_dropout_rate = None
optimizer_name = None
feature_map_dropout_rate = None
```

```
kernel_size = None
num_of_output_channels = None
weight_decay = None
loss
selected_optimizer = None
normalizer_class = None
normalize_head_entity_embeddings
normalize_relation_embeddings
normalize_tail_entity_embeddings
hidden_normalizer
param_init
input_dp_ent_real
input_dp_rel_real
hidden_dropout
loss_history = []
byte_pair_encoding
max_length_subword_tokens
block_size
forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)
        Parameters
           x (B x 2 x T)
forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors
        Parameters
init_params_with_sanity_checking()
forward(x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
          y_idx: torch.LongTensor = None
        Parameters
            • x
            • y_idx
            • ordered_bpe_entities
```

```
forward\_triples(x: torch.LongTensor) \rightarrow 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)
     \mathtt{get\_embeddings}() \rightarrow Tuple[numpy.ndarray, numpy.ndarray]
class dicee.models.IdentityClass(args=None)
     Bases: torch.nn.Module
```

Your models should also subclass this class.

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

```
import torch.nn as nn
import torch.nn.functional as F
class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to(), etc.

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
args
__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 to nest them in a tree structure. You can assign the submodules as regular attributes:

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
args
embedding_dim = None
num entities = None
```

```
num_relations = None
num_tokens = None
learning_rate = None
apply_unit_norm = None
input_dropout_rate = None
hidden_dropout_rate = None
optimizer_name = None
feature_map_dropout_rate = None
kernel_size = None
num_of_output_channels = None
weight_decay = None
loss
selected_optimizer = None
normalizer_class = None
normalize_head_entity_embeddings
normalize_relation_embeddings
normalize_tail_entity_embeddings
hidden_normalizer
param_init
input_dp_ent_real
input_dp_rel_real
hidden_dropout
loss_history = []
byte_pair_encoding
max_length_subword_tokens
block_size
forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)
        Parameters
           x (B x 2 x T)
forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors
        Parameters
```

```
init_params_with_sanity_checking()
     forward (x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
                 y idx: torch.LongTensor = None)
              Parameters
                  • x
                  y_idx
                  • ordered_bpe_entities
     forward\_triples(x: torch.LongTensor) \rightarrow torch.Tensor
              Parameters
     forward_k_vs_all(*args, **kwargs)
     forward_k_vs_sample(*args, **kwargs)
     get_triple_representation(idx_hrt)
     get_head_relation_representation(indexed_triple)
     get_sentence_representation(x: torch.LongTensor)
              Parameters
                  • (b (x shape)
                  • 3
                  • t)
     get_bpe_head_and_relation_representation(x: torch.LongTensor)
                  → Tuple[torch.FloatTensor, torch.FloatTensor]
              Parameters
                  x (B x 2 x T)
     get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]
class dicee.models.DistMult(args)
     Bases: dicee.models.base_model.BaseKGE
     Embedding Entities and Relations for Learning and Inference in Knowledge Bases https://arxiv.org/abs/1412.6575
     name = 'DistMult'
     k_vs_all_score (emb_h: torch.FloatTensor, emb_r: torch.FloatTensor, emb_E: torch.FloatTensor)
              Parameters
                  • emb h
                  • emb_r
                  • emb E
     forward_k_vs_all (x: torch.LongTensor)
```

```
forward_k_vs_sample (x: torch.LongTensor, target_entity_idx: torch.LongTensor)
     \mathtt{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) \rightarrow 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_width
     shallom
     \mathtt{get\_embeddings}() \rightarrow \mathsf{Tuple}[\mathsf{numpy}.\mathsf{ndarray}, \mathsf{None}]
     forward_k_vs_all(x) \rightarrow torch.FloatTensor
     forward\_triples(x) \rightarrow torch.FloatTensor
               Parameters
                   x
               Returns
class dicee.models.Pyke(args)
     Bases: dicee.models.base_model.BaseKGE
     A Physical Embedding Model for Knowledge Graphs
     name = 'Pyke'
     dist_func
     margin = 1.0
     forward_triples (x: torch.LongTensor)
               Parameters
                   x
class dicee.models.BaseKGE (args: dict)
     Bases: BaseKGELightning
     Base class for all neural network modules.
     Your models should also subclass this class.
```

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

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

```
args
embedding_dim = None
num_entities = None
num_relations = None
num_tokens = None
learning_rate = None
apply_unit_norm = None
input_dropout_rate = None
hidden_dropout_rate = None
optimizer_name = None
feature_map_dropout_rate = None
kernel_size = None
num_of_output_channels = None
weight_decay = None
loss
```

```
selected_optimizer = None
normalizer_class = None
normalize_head_entity_embeddings
normalize_relation_embeddings
normalize_tail_entity_embeddings
hidden_normalizer
param_init
input_dp_ent_real
input_dp_rel_real
hidden_dropout
loss_history = []
byte_pair_encoding
max_length_subword_tokens
block_size
forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)
        Parameters
            x (B x 2 x T)
forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
     byte pair encoded neural link predictors
        Parameters
init_params_with_sanity_checking()
forward (x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
           y_idx: torch.LongTensor = None)
        Parameters
            • x
            • y_idx
            • ordered_bpe_entities
\textbf{forward\_triples}~(\textit{x:torch.LongTensor})~\rightarrow torch.Tensor
        Parameters
forward_k_vs_all(*args, **kwargs)
forward_k_vs_sample(*args, **kwargs)
{\tt get\_triple\_representation}\,(idx\_hrt)
```

```
get_head_relation_representation(indexed_triple)
     get_sentence_representation(x: torch.LongTensor)
              Parameters
                   • (b (x shape)
                   • t)
     get_bpe_head_and_relation_representation(x: torch.LongTensor)
                  → Tuple[torch.FloatTensor, torch.FloatTensor]
              Parameters
                  x (B x 2 x T)
     get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]
class dicee.models.ConEx(args)
     Bases: dicee.models.base_model.BaseKGE
     Convolutional ComplEx Knowledge Graph Embeddings
     name = 'ConEx'
     conv2d
     fc_num_input
     fc1
     norm_fc1
     bn conv2d
     feature_map_dropout
     residual_convolution(C_1: Tuple[torch.Tensor, torch.Tensor],
                 C_2: Tuple[torch.Tensor, torch.Tensor]) \rightarrow 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) \rightarrow torch.FloatTensor
     forward\_triples(x: torch.Tensor) \rightarrow 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
```

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:

```
\label{eq:continuous_problem} \begin{split} \textbf{forward_k\_vs\_all} & (x: torch.Tensor) \rightarrow \text{torch.FloatTensor} \\ \textbf{forward\_triples} & (x: torch.Tensor) \rightarrow \text{torch.FloatTensor} \\ \textbf{Parameters} \\ & \textbf{x} \end{split}
```

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – 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.quaternion_mul(*, Q_1, Q_2)
             → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]
     Perform quaternion multiplication :param Q_1: :param Q_2: :return:
class dicee.models.BaseKGE (args: dict)
     Bases: BaseKGELightning
```

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

As per the example above, an <u>__init__()</u> call to the parent class must be made before assignment on the child.

Variables

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

```
args
embedding_dim = None
num_entities = None
num_relations = None
num_tokens = None
learning_rate = None
apply_unit_norm = None
input_dropout_rate = None
hidden_dropout_rate = None
optimizer_name = None
feature_map_dropout_rate = None
kernel_size = None
num_of_output_channels = None
weight_decay = None
loss
selected_optimizer = None
normalizer_class = None
normalize_head_entity_embeddings
normalize_relation_embeddings
normalize_tail_entity_embeddings
hidden_normalizer
param_init
input_dp_ent_real
input_dp_rel_real
hidden_dropout
loss_history = []
byte_pair_encoding
max_length_subword_tokens
block_size
forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)
       Parameters
          x (B x 2 x T)
```

```
forward_byte_pair_encoded_triple(x: Tuple[torch.LongTensor, torch.LongTensor])
          byte pair encoded neural link predictors
              Parameters
     init_params_with_sanity_checking()
     forward(x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
                 y idx: torch.LongTensor = None
              Parameters
                  • x
                  • y_idx
                  • ordered_bpe_entities
     forward\_triples(x: torch.LongTensor) \rightarrow torch.Tensor
              Parameters
     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)
                  \rightarrow Tuple[torch.FloatTensor, torch.FloatTensor]
              Parameters
                  x (B x 2 x T)
     \mathtt{get\_embeddings}() \rightarrow Tuple[numpy.ndarray, numpy.ndarray]
class dicee.models.IdentityClass(args=None)
     Bases: torch.nn.Module
```

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to (), etc.

1 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
__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 to nest them in a tree structure. You can assign the submodules as regular attributes:

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

name = 'OMult'

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.

 $static quaternion_normalizer(x: torch.FloatTensor) \rightarrow torch.FloatTensor$

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:

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

Parameters

 \mathbf{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 \text{ 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
     {\tt 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
```

```
\label{lem: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 to nest them in a tree structure. You can assign the submodules as regular attributes:

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

As per the example above, an <u>__init__()</u> call to the parent class must be made before assignment on the child.

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

args

embedding_dim = None

```
num_entities = None
num_relations = None
num_tokens = None
learning_rate = None
apply_unit_norm = None
input_dropout_rate = None
hidden_dropout_rate = None
optimizer_name = None
feature_map_dropout_rate = None
kernel_size = None
num_of_output_channels = None
weight_decay = None
loss
selected_optimizer = None
normalizer_class = None
normalize_head_entity_embeddings
normalize_relation_embeddings
normalize_tail_entity_embeddings
hidden_normalizer
param_init
input_dp_ent_real
input_dp_rel_real
hidden_dropout
loss_history = []
byte_pair_encoding
max_length_subword_tokens
block_size
forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)
       Parameters
          x (B x 2 x T)
```

```
forward_byte_pair_encoded_triple(x: Tuple[torch.LongTensor, torch.LongTensor])
          byte pair encoded neural link predictors
              Parameters
     init_params_with_sanity_checking()
     forward(x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
                 y idx: torch.LongTensor = None
              Parameters
                  • x
                  • y_idx
                  • ordered_bpe_entities
     forward\_triples(x: torch.LongTensor) \rightarrow torch.Tensor
              Parameters
     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)
                  \rightarrow Tuple[torch.FloatTensor, torch.FloatTensor]
              Parameters
                  x (B x 2 x T)
     \mathtt{get\_embeddings}() \rightarrow 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 to nest them in a tree structure. You can assign the submodules as regular attributes:

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to (), etc.

1 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
__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 to nest them in a tree structure. You can assign the submodules as regular attributes:

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

class Model (nn.Module):
    def __init__ (self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

def forward(self, x):
```

(continues on next page)

(continued from previous page)

```
x = F.relu(self.conv1(x))
return F.relu(self.conv2(x))
```

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

1 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.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

As per the example above, an <u>__init__()</u> call to the parent class must be made before assignment on the child.

Variables

name = 'ConvO'

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
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) \rightarrow 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 l)
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)
```

```
\begin{tabular}{ll} {\bf residual\_convolution} & (O\_1,O\_2) \\ {\bf forward\_triples} & (x:torch.Tensor) & \rightarrow {\bf torch}.{\bf Tensor} \\ & {\bf Parameters} \\ & {\bf x} \\ \end{tabular}
```

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.

forward_k_vs_all (x: torch.Tensor)

Your models should also subclass this class.

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

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

class Model (nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

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

Variables

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

```
name = 'Keci'
p
q
r
requires_grad_for_interactions = True
```

```
compute\_sigma\_pp(hp, rp)
          Compute sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k
          sigma {pp} captures the interactions between along p bases For instance, let p e 1, e 2, e 3, we compute
          interactions between e 1 e 2, e 1 e 3, and e 2 e 3 This can be implemented with a nested two for loops
                  results = [] for i in range(p - 1):
                          for k in range(i + 1, p):
                               results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])
                  sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))
          Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1,
          e1e2, e1e3,
                  e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
          Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.
compute\_sigma\_qq(hq, rq)
          Compute sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k sigma_{q}
          captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions
          between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops
                  results = [] for j in range(q - 1):
                          for k in range(j + 1, q):
                              results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])
                  sigma qq = torch.stack(results, dim=2) assert sigma qq.shape == (b, r, int((q * (q - 1)) / 2))
          Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1,
          e1e2, e1e3,
                  e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
          Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.
compute_sigma_pq(*, hp, hq, rp, rq)
          sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j
          results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):
                  for j in range(q):
                          sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
          print(sigma_pq.shape)
apply_coefficients(hp, hq, rp, rq)
          Multiplying a base vector with its scalar coefficient
clifford_multiplication (h0, hp, hq, r0, rp, rq)
          Compute our CL multiplication
                  sum_{j=p+1}^{p+q} r_j e_j
                  ei ^2 = +1 for i = < i = < p ej ^2 = -1 for p < j = < p+q ei ej = -eje1 for i
          eq j
                  h r = sigma_0 + sigma_p + sigma_q + sigma_{pp} + sigma_{q} + sig
```

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

- (2) $sigma_p = sum_{i=1}^p (h_0 r_i + h_i r_0) e_i$
- (3) $sigma_q = sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j$
- (4) $sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k h_k r_i) e_i e_k$
- (5) $sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k h_k r_j) e_j e_k$
- (6) $sigma_{pq} = sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j h_j r_i) e_i e_j$

construct cl multivector(x: torch.FloatTensor, r: int, p: int, q: int)

→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

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

Parameter

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

returns

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

forward_k_vs_with_explicit(x: torch.Tensor)

k_vs_all_score (bpe_head_ent_emb, bpe_rel_ent_emb, E)

 $forward_k_vs_all(x: torch.Tensor) \rightarrow torch.FloatTensor$

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,|E|) shape

construct_batch_selected_cl_multivector(x: torch.FloatTensor, r: int, p: int, q: int)

→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

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

Parameter

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

returns

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

 $forward_k_vs_sample$ (x: torch.LongTensor, target_entity_idx: torch.LongTensor) \rightarrow 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) \rightarrow torch.FloatTensor
          Parameter
          x: torch.LongTensor with (n,3) shape
               rtype
                   torch.FloatTensor with (n) shape
class dicee.models.KeciBase(args)
     Bases: Keci
     Without learning dimension scaling
     name = 'KeciBase'
     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 to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F
class Model (nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – 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) \longrightarrow 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))

 $\verb|compute_sigmas_single| (\textit{list}_h_\textit{emb}, \textit{list}_r_\textit{emb}, \textit{list}_t_\textit{emb})$

here we compute all the sums with no others vectors interaction taken with the scalar product with t, that is,

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

and return:

$$sigma_0t = \sigma_0 \cdot t_0 = s0 + s1 - s2s3, s4ands5$$

compute_sigmas_multivect (list_h_emb, list_r_emb)

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

For same bases vectors interaction we have

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

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

 $forward_k_vs_all(x: torch.Tensor) \rightarrow torch.FloatTensor$

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations
- (2) Construct head entity and relation embeddings according to Cl {p,q, r}(mathbb{R}^d).
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

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

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

Multiplying a base vector with its scalar coefficient

construct_cl_multivector (x: torch.FloatTensor, re: int, p: int, q: int, r: int)

→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

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

Parameter

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

returns

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

 $compute_sigma_pp(hp, rp)$

Compute .. math:

sigma_{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,

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

 $\texttt{compute_sigma_qq}\,(hq,rq)$

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])

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

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

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)

 $\texttt{compute_sigma_pr} \ (*, hp, hk, rp, rk)$

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)

 $\texttt{compute_sigma_qr} \ (*, hq, hk, rq, rk)$

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

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

for j in range(q):

$$sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]$$

print(sigma_pq.shape)

```
class dicee.models.BaseKGE (args: dict)
```

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

As per the example above, an <u>__init___()</u> call to the parent class must be made before assignment on the child.

Variables

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

args

```
embedding_dim = None
num_entities = None
num_relations = None
num_tokens = None
learning_rate = None
apply_unit_norm = None
input_dropout_rate = None
hidden_dropout_rate = None
optimizer_name = None
feature_map_dropout_rate = None
```

```
kernel_size = None
num_of_output_channels = None
weight_decay = None
loss
selected_optimizer = None
normalizer_class = None
normalize_head_entity_embeddings
normalize_relation_embeddings
normalize_tail_entity_embeddings
hidden_normalizer
param_init
input_dp_ent_real
input_dp_rel_real
hidden_dropout
loss_history = []
byte_pair_encoding
max_length_subword_tokens
block_size
forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)
        Parameters
           x (B x 2 x T)
forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors
        Parameters
init_params_with_sanity_checking()
forward(x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
          y_idx: torch.LongTensor = None
        Parameters
            • x
            • y_idx
            • ordered_bpe_entities
```

```
forward_triples (x: torch.LongTensor) \rightarrow 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)
              \mathtt{get\_embeddings}() \rightarrow 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: Py-
              keen HolE:
              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.embeddin
                                     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) \rightarrow 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:
 - $$\label{eq:hammon} \begin{split} &h = h.reshape(len(x), self.embedding_dim, self.last_dim) \ r = r.reshape(len(x), self.embedding_dim, self.last_dim) \end{split}$$
 $\ & t = t.reshape(len(x), self.embedding_dim, self.last_dim) \end{split}$
- # (3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice_size=None, slice_dim=0)

abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)

```
class dicee.models.BaseKGE(args: dict)
```

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

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

Variables

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

args

```
embedding_dim = None
num_entities = None
num_relations = None
num_tokens = None
learning_rate = None
apply_unit_norm = None
input_dropout_rate = None
hidden_dropout_rate = None
optimizer_name = None
feature_map_dropout_rate = None
kernel_size = None
num_of_output_channels = None
weight_decay = None
loss
selected_optimizer = None
normalizer_class = None
normalize_head_entity_embeddings
normalize_relation_embeddings
normalize_tail_entity_embeddings
hidden_normalizer
param_init
input_dp_ent_real
input_dp_rel_real
hidden_dropout
loss_history = []
byte_pair_encoding
max_length_subword_tokens
block_size
forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)
       Parameters
          x (B x 2 x T)
```

```
forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
          byte pair encoded neural link predictors
              Parameters
     init_params_with_sanity_checking()
     forward(x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
                 y_idx: torch.LongTensor = None
              Parameters
                  • x
                  • y_idx
                  • ordered_bpe_entities
     forward\_triples(x: torch.LongTensor) \rightarrow torch.Tensor
              Parameters
     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)
                  \rightarrow Tuple[torch.FloatTensor, torch.FloatTensor]
              Parameters
                  x (B x 2 x T)
     \mathtt{get\_embeddings}() \rightarrow Tuple[numpy.ndarray, numpy.ndarray]
class dicee.models.FMult(args)
     Bases: dicee.models.base_model.BaseKGE
     Learning Knowledge Neural Graphs
     name = 'FMult'
     entity_embeddings
     relation_embeddings
     k
```

```
num_sample = 50
      gamma
      roots
      weights
      compute\_func(weights: torch.FloatTensor, x) \rightarrow torch.FloatTensor
      chain_func (weights, x: torch.FloatTensor)
      forward\_triples(idx\_triple: torch.Tensor) \rightarrow torch.Tensor
               Parameters
class dicee.models.GFMult(args)
      Bases: dicee.models.base_model.BaseKGE
      Learning Knowledge Neural Graphs
      name = 'GFMult'
      entity_embeddings
      relation_embeddings
      num_sample = 250
      roots
      weights
      \verb|compute_func| (\textit{weights: torch.FloatTensor}, \textit{x}) \rightarrow \textit{torch.FloatTensor}
      chain_func(weights, x: torch.FloatTensor)
      \textbf{forward\_triples} (\textit{idx\_triple: torch.Tensor}) \rightarrow \text{torch.Tensor}
               Parameters
class dicee.models.FMult2(args)
      Bases: dicee.models.base_model.BaseKGE
      Learning Knowledge Neural Graphs
      name = 'FMult2'
     n_{\text{layers}} = 3
      tuned_embedding_dim = False
     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.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=0}^{k=d-1}wk e^{kix}, and use the three differents scoring function as in the paper to evaluate
     the score
     name = 'LFMult1'
     entity_embeddings
     relation_embeddings
     forward_triples (idx_triple)
               Parameters
                   x
     \texttt{tri\_score}(h, r, t)
     \mathtt{vtp\_score}(h, r, t)
class dicee.models.LFMult(args)
     Bases: dicee.models.base_model.BaseKGE
     Embedding with polynomial functions. We represent all entities and relations in the polynomial space as: f(x) =
     sum_{i=0}^{d-1} a_k x^{i\%} 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
     x_values
```

forward_triples (idx_triple)

Parameters

x

construct_multi_coeff(X)

poly NN (x, coefh, coefr, coeft)

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

linear(x, w, b)

$scalar_batch_NN(a, b, c)$

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

tri_score (coeff_h, coeff_r, coeff_t)

this part implement the trilinear scoring techniques:

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

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

$\mathtt{vtp_score}(h, r, t)$

this part implement the vector triple product scoring techniques:

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

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

```
comp_func(h, r, t)
```

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

```
polynomial(coeff, x, degree)
```

This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,...d] and return a vector tensor (coeff $[0][0] + \text{coeff}[0][1]x + ... + \text{coeff}[0][d]x^d$,

$$coeff[1][0] + coeff[1][1]x + ... + coeff[1][d]x^d$$

pop (coeff, x, degree)

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

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

$$coeff[1][0] + coeff[1][1]x + ... + coeff[1][d]x^d$$

class dicee.models.DualE(args)

Bases: dicee.models.base_model.BaseKGE

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

```
name = 'DualE'
      entity_embeddings
      relation_embeddings
      num_ent
      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) \rightarrow \text{torch.tensor}
           KvsAll scoring function
           Input
           x: torch.LongTensor with (n, ) shape
           Output
           torch.FloatTensor with (n) shape
      forward\_triples(idx\_triple: torch.tensor) \rightarrow 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) \rightarrow torch.tensor
           Transpose function
           Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)
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
     \mathtt{start}() \to \mathrm{None}
           Preprocess train, valid and test datasets stored in knowledge graph instance
           Parameter
               rtype
                   None
     preprocess_with_byte_pair_encoding()
     {\tt preprocess\_with\_byte\_pair\_encoding\_with\_padding}\,()\,\to None
     {\tt preprocess\_with\_pandas}\,()\,\to None
           Preprocess train, valid and test datasets stored in knowledge graph instance with pandas
           (1) Add recipriocal or noisy triples
           (2) Construct vocabulary
           (3) Index datasets
           Parameter
               rtype
                   None
     {\tt preprocess\_with\_polars}\,()\,\to None
     \verb"sequential_vocabulary_construction"\ () \ \to 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

```
\begin{tabular}{ll} {\bf class} & {\tt dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk.ReadFromDisk} \end{tabular} \label{table_load_kg}. \\ & {\tt Read} & {\tt the} & {\tt data} & {\tt from} & {\tt disk} & {\tt into} & {\tt memory} \\ & {\tt kg} & \\ \end{tabular}
```

```
start () → None
Read a knowledge graph from disk into memory

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

Parameter

None

rtype
None
add_noisy_triples_into_training()

dicee.read_preprocess_save_load_kg.save_load_disk
```

Classes

LoadSaveToDisk

Module Contents

```
class dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk(kg)
    kg
    save()
    load()
```

dicee.read_preprocess_save_load_kg.util

Functions

polars_dataframe_indexer(→ polars.DataFrame)	Replaces 'subject', 'relation', and 'object' columns in the input Polars DataFrame with their corresponding index values
<pre>pandas_dataframe_indexer(→ pandas.DataFrame)</pre>	Replaces 'subject', 'relation', and 'object' columns in the input Pandas DataFrame with their corresponding index values
dept_index_triples_with_pandas()	
<pre>apply_reciprical_or_noise(add_reciprical, eval_model)</pre>	
timeit(func)	
read_with_polars(→ polars.DataFrame)	Load and Preprocess via Polars
read_with_pandas(data_path[, read_only_few,])	Load and Treprocess via Totals
$\label{eq:read_from_disk} read_from_disk(\rightarrow Tuple[polars.DataFrame, pan-das.DataFrame])$	
<pre>read_from_triple_store([endpoint]) get_er_vocab(data[, file_path])</pre>	Read triples from triple store into pandas dataframe
<pre>get_re_vocab(data[, file_path])</pre>	
<pre>get_ee_vocab(data[, file_path])</pre>	
<pre>create_constraints(triples[, file_path])</pre>	
$load_with_pandas(\rightarrow None)$	Deserialize data
save_numpy_ndarray(*, data, file_path)	2 (50.1
<pre>load_numpy_ndarray(*, file_path)</pre>	
<pre>save_pickle(*, data[, file_path])</pre>	
<pre>load_pickle(*[, file_path])</pre>	
create_recipriocal_triples(x)	Add inverse triples into dask dataframe
dataset_sanity_checking(→ None)	•

Module Contents

```
dicee.read_preprocess_save_load_kg.util.polars_dataframe_indexer( df_polars: polars.DataFrame, idx_entity: polars.DataFrame, idx_relation: polars.DataFrame) <math>\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) <math>\rightarrow 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.

Parameters

- train_set pandas dataframe
- entity_to_idx a mapping from str to integer index
- relation_to_idx a mapping from str to integer index
- num_core number of cores to be used

Returns

indexed triples, i.e., pandas dataframe

dicee.read_preprocess_save_load_kg.util.timeit(func)

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

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

Load and Preprocess via Polars

```
dicee.read_preprocess_save_load_kg.util.read_with_pandas(data_path, read_only_few: int = None, sample_triples_ratio: float = None, separator: str = None)
```

```
dicee.read_preprocess_save_load_kg.util.read_from_disk (data_path: str, read_only_few: int = None, sample_triples_ratio: float = None, backend: str = None, separator: str = None) \rightarrow Tuple[polars.DataFrame, pandas.DataFrame]
```

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

Read triples from triple store into pandas dataframe

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

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

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

- num_entities
- num_relations

Returns

Classes

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

Package Contents

```
class dicee.read_preprocess_save_load_kg.PreprocessKG (kg)
Preprocess the data in memory

kg

start () \rightarrow None

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

```
Parameter
               rtvpe
                    None
      preprocess_with_byte_pair_encoding()
      {\tt preprocess\_with\_byte\_pair\_encoding\_with\_padding}\,()\,\to None
      {\tt preprocess\_with\_pandas}\,()\,\to None
           Preprocess train, valid and test datasets stored in knowledge graph instance with pandas
           (1) Add recipriocal or noisy triples
           (2) Construct vocabulary
           (3) Index datasets
           Parameter
               rtype
                   None
      {\tt preprocess\_with\_polars}\,()\,\to None
      sequential\_vocabulary\_construction() \rightarrow 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()
\verb"class" dicee.read_preprocess_save_load_kg.ReadFromDisk" (\textit{kg})
      Read the data from disk into memory
      kg
      \mathtt{start}() \rightarrow \mathrm{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.scripts

Submodules

dicee.scripts.index

Functions

```
get_default_arguments()
main()
```

Module Contents

```
dicee.scripts.index.get_default_arguments()
dicee.scripts.index.main()
```

dicee.scripts.run

Functions

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

Module Contents

dicee.scripts.serve

Attributes

```
app
neural_searcher
```

Classes

NeuralSearcher

Functions

```
get_default_arguments()
root()
search_embeddings(q)
retrieve_embeddings(q)
main()
```

Module Contents

```
dicee.scripts.serve.app
dicee.scripts.serve.neural_searcher = None
dicee.scripts.serve.get_default_arguments()
async dicee.scripts.serve.root()
async dicee.scripts.serve.search_embeddings(q: str)
async dicee.scripts.serve.retrieve_embeddings(q: str)
class dicee.scripts.serve.NeuralSearcher(args)
     collection_name
    model
    qdrant_client
    get (entity: str)
     search (entity: str)
dicee.scripts.serve.main()
dicee.trainer
```

Submodules

dicee.trainer.dice_trainer

Classes

DICE_Trainer

DICE_Trainer implement

Functions

```
load_term_mapping([file_path])
initialize_trainer(args, callbacks)
get_callbacks(args)
```

```
Module Contents
dicee.trainer.dice_trainer.load_term_mapping(file_path=str)
dicee.trainer.dice_trainer.initialize_trainer(args, callbacks)
dicee.trainer.dice_trainer.get_callbacks(args)
class dicee.trainer.dice_trainer.DICE_Trainer (args, is_continual_training, 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
     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)

 $initialize_trainer(callbacks: List) \rightarrow lightning.Trainer$

Initialize Trainer from input arguments

```
initialize_or_load_model()
```

 $\verb"init_dataloader" (\textit{dataset: torch.utils.data.Dataset}) \rightarrow torch.utils.data.DataLoader$

 $\verb"init_dataset" () \rightarrow torch.utils.data.Dataset"$

start (knowledge_graph: dicee.knowledge_graph.KG | numpy.memmap)

 \rightarrow 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.

 $k_fold_cross_validation(dataset) \rightarrow Tuple[dicee.models.base_model.BaseKGE, str]$

Perform K-fold Cross-Validation

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

Parameters

- self
- dataset

Returns

model

dicee.trainer.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
      \textbf{fit} \ (*args, train\_dataloaders, **kwargs) \ \rightarrow None
               Training starts
               Arguments
           kwargs:Tuple
               empty dictionary
               Return type
                   batch loss (float)
      \textbf{forward\_backward\_update} (x\_batch: torch.Tensor, y\_batch: torch.Tensor) \rightarrow \text{torch}.Tensor
               Compute forward, loss, backward, and parameter update
               Arguments
               Return type
                   batch loss (float)
      extract_input_outputs_set_device(batch: list) \rightarrow 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

A Trainer based on torch.nn.parallel.DistributedDataParallel

TorchDDPTrainer NodeTrainer

Functions

```
make_iterable_verbose(\rightarrow Iterable)
```

Module Contents

```
dicee.trainer.torch_trainer_ddp.make_iterable_verbose(iterable_object, verbose,
            desc='Default', position=None, leave=True) \rightarrow Iterable
class dicee.trainer.torch_trainer_ddp.TorchDDPTrainer(args, callbacks)
     Bases: dicee.abstracts.AbstractTrainer
          A Trainer based on torch.nn.parallel.DistributedDataParallel
          Arguments
     entity_idxs
          mapping.
     relation_idxs
          mapping.
     form
     store
     label_smoothing_rate
          Using hard targets (0,1) drives weights to infinity. An outlier produces enormous gradients.
          Return type
              torch.utils.data.Dataset
     fit (*args, **kwargs)
          Train model
class dicee.trainer.torch_trainer_ddp.NodeTrainer(trainer, model: torch.nn.Module,
            train_dataset_loader: torch.utils.data.DataLoader, callbacks, num_epochs: int)
     trainer
     local_rank
     global_rank
     optimizer
     train_dataset_loader
     loss_func
     callbacks
     model
```

```
num_epochs
loss_history = []
ptdtype
ctx
scaler
extract_input_outputs(z: list)
train()
    Training loop for DDP
```

Classes

DICE_Trainer

DICE_Trainer implement

Package Contents

class dicee.trainer.DICE_Trainer(args, is_continual_training, 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

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) \rightarrow lightning.Trainer
```

Initialize Trainer from input arguments

```
initialize_or_load_model()
```

 $\verb"init_dataloader" (\textit{dataset: torch.utils.data.Dataset}) \rightarrow torch.utils.data.DataLoader$

 $\verb"init_dataset" () \rightarrow torch.utils.data.Dataset"$

 $\begin{tabular}{ll} \textbf{start} & (knowledge_graph: dicee.knowledge_graph.KG \mid numpy.memmap) \\ & \rightarrow \textbf{Tuple}[dicee.models.base_model.BaseKGE, str] \end{tabular}$

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.

 $k_fold_cross_validation(dataset) \rightarrow Tuple[dicee.models.base_model.BaseKGE, str]$

Perform K-fold Cross-Validation

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

Parameters

- self
- dataset

Returns

model

14.2 Submodules

dicee. main

dicee.abstracts

Classes

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

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
strategy = None
on_fit_start(*args, **kwargs)
     A function to call callbacks before the training starts.
     Parameter
     args
     kwargs
         rtype
             None
on_fit_end(*args, **kwargs)
     A function to call callbacks at the ned of the training.
     Parameter
     args
     kwargs
         rtype
```

None

```
on_train_epoch_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
      \mathtt{static}\ \mathtt{save\_checkpoint}\ (\mathit{full\_path}: \mathit{str}, \mathit{model}) \ 	o \ \mathsf{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
      apply_semantic_constraint
      configs
      \texttt{get\_eval\_report}() \rightarrow dict
```

```
\texttt{get\_bpe\_token\_representation} (\textit{str\_entity\_or\_relation: List[str] | str}) \rightarrow \texttt{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)
\texttt{get\_padded\_bpe\_triple\_representation} (triples: List[List[str]]) \rightarrow Tuple[List, List, List]
          Parameters
              triples
\verb"set_model_train_mode"() \to None
     Setting the model into training mode
     Parameter
\verb"set_model_eval_mode"() \to None
     Setting the model into eval mode
     Parameter
property name
sample\_entity(n:int) \rightarrow List[str]
sample\_relation(n:int) \rightarrow List[str]
is\_seen(entity: str = None, relation: str = None) \rightarrow bool
save() \rightarrow 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.AbstractCallback
     Bases: \verb|abc.ABC|, \verb|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
{\tt store\_ensemble} \ (param\_ensemble) \ 	o None
```

dicee.analyse_experiments

This script should be moved to dicee/scripts

Classes

Experiment

Functions

```
get_default_arguments()
analyse(args)
```

Module Contents

```
\verb|dicee.analyse_experiments.get_default_arguments||\\
class dicee.analyse_experiments.Experiment
    model_name = []
    callbacks = []
    embedding_dim = []
    num_params = []
    num_epochs = []
    batch_size = []
    lr = []
    byte_pair_encoding = []
    aswa = []
    path_dataset_folder = []
    full_storage_path = []
    pq = []
    train_mrr = []
    train_h1 = []
    train_h3 = []
    train_h10 = []
    val_mrr = []
    val_h1 = []
    val_h3 = []
    val_h10 = []
    test_mrr = []
    test_h1 = []
    test_h3 = []
    test_h10 = []
    runtime = []
    normalization = []
    scoring_technique = []
    save_experiment(x)
```

```
to_df()
```

dicee.analyse_experiments.analyse(args)

dicee.callbacks

Classes

AccumulateEpochLossCallback	Abstract class for Callback class for knowledge graph embedding models
PrintCallback	Abstract class for Callback class for knowledge graph embedding models
KGESaveCallback	Abstract class for Callback class for knowledge graph embedding models
PseudoLabellingCallback	Abstract class for Callback class for knowledge graph embedding models
ASWA	Adaptive stochastic weight averaging
Eval	Abstract class for Callback class for knowledge graph embedding models
KronE	Abstract class for Callback class for knowledge graph embedding models
Perturb	A callback for a three-Level Perturbation

Functions

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

Module Contents

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

Bases: dicee.abstracts.AbstractCallback

Abstract class for Callback class for knowledge graph embedding models

Parameter

path

on_fit_end(trainer, model) → None

Store epoch loss

Parameter

trainer:

model:

rtype
```

None

```
class dicee.callbacks.PrintCallback
      Bases: dicee.abstracts.AbstractCallback
      Abstract class for Callback class for knowledge graph embedding models
      Parameter
      start_time
      \verb"on_fit_start" (\textit{trainer}, \textit{pl}\_\textit{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
      \verb"on_fit_start" (\textit{trainer}, \textit{pl}\_\textit{module})
           Call at the beginning of the training.
           Parameter
           trainer:
           model:
               rtype
                   None
      on_train_epoch_end(*args, **kwargs)
           Call at the end of each epoch during training.
           Parameter
           trainer:
           model:
               rtype
                   None
      on_fit_end(*args, **kwargs)
           Call at the end of the training.
```

```
Parameter
          trainer:
          model:
             rtype
                 None
     on_epoch_end (model, trainer, **kwargs)
class dicee.callbacks.PseudoLabellingCallback(data_module, kg, batch_size)
     Bases: dicee.abstracts.AbstractCallback
     Abstract class for Callback class for knowledge graph embedding models
     Parameter
     data_module
     kg
     num_of_epochs = 0
     unlabelled_size
     batch_size
     create_random_data()
     on\_epoch\_end(trainer, model)
dicee.callbacks.estimate_q(eps)
     estimate rate of convergence q from sequence esp
dicee.callbacks.compute_convergence (seq, i)
class dicee.callbacks.ASWA (num_epochs, path)
     Bases: dicee.abstracts.AbstractCallback
     Adaptive stochastic weight averaging ASWE keeps track of the validation performance and update s the ensemble
     model accordingly.
     path
     num_epochs
     initial_eval_setting = None
     epoch_count = 0
     alphas = []
     val_aswa
     on_fit_end(trainer, model)
```

Call at the end of the training.

```
Parameter
```

```
trainer:
          model:
              rtype
                  None
     static compute\_mrr(trainer, model) \rightarrow float
     get_aswa_state_dict(model)
     decide (running_model_state_dict, ensemble_state_dict, val_running_model,
                 mrr_updated_ensemble_model)
          Perform Hard Update, software or rejection
              Parameters
                   • running_model_state_dict
                   • ensemble_state_dict
                   • val_running_model
                   • mrr_updated_ensemble_model
     on_train_epoch_end(trainer, model)
          Call at the end of each epoch during training.
          Parameter
          trainer:
          model:
              rtype
                  None
class dicee.callbacks.Eval (path, epoch_ratio: int = None)
     Bases: dicee.abstracts.AbstractCallback
     Abstract class for Callback class for knowledge graph embedding models
     Parameter
     path
     reports = []
     epoch_ratio
     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
     \verb"on_train_epoch_end" (\textit{trainer}, \textit{model})
           Call at the end of each epoch during training.
           Parameter
           trainer:
           model:
               rtype
                   None
     on_train_batch_end(*args, **kwargs)
           Call at the end of each mini-batch during the training.
           Parameter
           trainer:
           model:
               rtype
                   None
class dicee.callbacks.KronE
     Bases: dicee.abstracts.AbstractCallback
     Abstract class for Callback class for knowledge graph embedding models
```

Parameter

```
f = None
```

```
static batch_kronecker_product(a, b)
```

Kronecker product of matrices a and b with leading batch dimensions. Batch dimensions are broadcast. The number of them mush :type a: torch.Tensor :type b: torch.Tensor :rtype: torch.Tensor

```
get_kronecker_triple_representation (indexed_triple: torch.LongTensor)
           Get kronecker embeddings
     on_fit_start (trainer, model)
           Call at the beginning of the training.
           Parameter
           trainer:
           model:
               rtvpe
                   None
class dicee.callbacks.Perturb(level: str = 'input', ratio: float = 0.0, method: str = None,
            scaler: float = None, frequency=None)
     Bases: dicee.abstracts.AbstractCallback
     A callback for a three-Level Perturbation
     Input Perturbation: During training an input x is perturbed by randomly replacing its element. In the context of
     knowledge graph embedding models, x can denote a triple, a tuple of an entity and a relation, or a tuple of two
     entities. A perturbation means that a component of x is randomly replaced by an entity or a relation.
     Parameter Perturbation:
     Output Perturbation:
     level
     ratio
     method
     scaler
     frequency
     on_train_batch_start (trainer, model, batch, batch_idx)
           Called when the train batch begins.
dicee.config
```

Classes

Namespace

Simple object for storing attributes.

Module Contents

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

Bases: argparse.Namespace

Simple object for storing attributes.

Implements equality by attribute names and values, and provides a simple string representation.

```
dataset_dir: str = None
     The path of a folder containing train.txt, and/or valid.txt and/or test.txt
save_embeddings_as_csv: bool = False
     Embeddings of entities and relations are stored into CSV files to facilitate easy usage.
storage_path: str = 'Experiments'
     A directory named with time of execution under -storage_path that contains related data about embeddings.
path_to_store_single_run: str = None
     A single directory created that contains related data about embeddings.
path_single_kg = None
     Path of a file corresponding to the input knowledge graph
sparql_endpoint = None
     An endpoint of a triple store.
model: str = 'Keci'
     KGE model
optim: str = 'Adam'
     Optimizer
embedding_dim: int = 64
     Size of continuous vector representation of an entity/relation
num_epochs: int = 150
     Number of pass over the training data
batch_size: int = 1024
     Mini-batch size if it is None, an automatic batch finder technique applied
lr: float = 0.1
     Learning rate
add_noise_rate: float = None
     The ratio of added random triples into training dataset
gpus = None
     Number GPUs to be used during training
callbacks
     10}}
         Type
             Callbacks, e.g., {"PPE"
             { "last_percent_to_consider"
backend: str = 'pandas'
     Backend to read, process, and index input knowledge graph. pandas, polars and rdflib available
separator: str = '\\s+'
     separator for extracting head, relation and tail from a triple
trainer: str = 'torchCPUTrainer'
     Trainer for knowledge graph embedding model
```

```
scoring_technique: str = 'KvsAll'
    Scoring technique for knowledge graph embedding models
neg_ratio: int = 0
    Negative ratio for a true triple in NegSample training_technique
weight decay: float = 0.0
    Weight decay for all trainable params
normalization: str = 'None'
    LayerNorm, BatchNorm1d, or None
init_param: str = None
    xavier_normal or None
gradient_accumulation_steps: int = 0
    Not tested e
num_folds_for_cv: int = 0
    Number of folds for CV
eval_model: str = 'train_val_test'
    ["None", "train", "train_val", "train_val_test", "test"]
        Type
            Evaluate trained model choices
save_model_at_every_epoch: int = None
    Not tested
label_smoothing_rate: float = 0.0
num_core: int = 0
    Number of CPUs to be used in the mini-batch loading process
random_seed: int = 0
    Random Seed
sample_triples_ratio: float = None
    Read some triples that are uniformly at random sampled. Ratio being between 0 and 1
read_only_few: int = None
    Read only first few triples
pykeen_model_kwargs
    Additional keyword arguments for pykeen models
kernel_size: int = 3
    Size of a square kernel in a convolution operation
num_of_output_channels: int = 32
    Number of slices in the generated feature map by convolution.
p: int = 0
    P parameter of Clifford Embeddings
q: int = 1
    Q parameter of Clifford Embeddings
```

```
input_dropout_rate: float = 0.0
    Dropout rate on embeddings of input triples
hidden_dropout_rate: float = 0.0
    Dropout rate on hidden representations of input triples
feature_map_dropout_rate: float = 0.0
    Dropout rate on a feature map generated by a convolution operation
byte_pair_encoding: bool = False
    Byte pair encoding
        Type
            WIP
adaptive_swa: bool = False
    Adaptive stochastic weight averaging
swa: bool = False
    Stochastic weight averaging
block_size: int = None
    block size of LLM
continual_learning = None
    Path of a pretrained model size of LLM
__iter__()
```

dicee.dataset_classes

Classes

BPE_NegativeSamplingDataset	An abstract class representing a Dataset.
MultiLabelDataset	An abstract class representing a Dataset.
MultiClassClassificationDataset	Dataset for the 1vsALL training strategy
OnevsAllDataset	Dataset for the 1vsALL training strategy
KvsAll	Creates a dataset for KvsAll training by inheriting from
	torch.utils.data.Dataset.
AllvsAll	Creates a dataset for AllvsAll training by inheriting from
	torch.utils.data.Dataset.
OnevsSample	A custom PyTorch Dataset class for knowledge graph em-
	beddings, which includes
KvsSampleDataset	KvsSample a Dataset:
NegSampleDataset	An abstract class representing a Dataset.
TriplePredictionDataset	Triple Dataset
CVDataModule	Create a Dataset for cross validation

Functions

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

Module Contents

Reload the files from disk to construct the Pytorch dataset

```
dicee.dataset_classes.construct_dataset (*, train_set: numpy.ndarray | list, valid_set=None, test_set=None, ordered_bpe_entities=None, train_target_indices=None, target_dim: int = None, entity_to_idx: dict, relation_to_idx: dict, form_of_labelling: str, scoring_technique: str, neg_ratio: int, label_smoothing_rate: float, byte_pair_encoding=None, block_size: int = None)

→ torch.utils.data.Dataset
```

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.



DataLoader by default constructs an index sampler that yields integral indices. To make it work with a mapstyle 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.

1 Note

Parameters

DataLoader by default constructs an index sampler that yields integral indices. To make it work with a mapstyle 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)
{\tt class} \ {\tt dicee.dataset\_classes.} \\ {\tt MultiClassClassificationDataset} \ (
            subword_units: numpy.ndarray, block_size: int = 8)
     Bases: torch.utils.data.Dataset
     Dataset for the 1vsALL training strategy
          Parameters
                • train_set_idx - Indexed triples for the training.
                • entity_idxs - mapping.
                • relation_idxs - mapping.
                • form - ?
                                                 https://pytorch.org/docs/stable/data.html#torch.utils.data.
                • num_workers - int
                  DataLoader
          Return type
              torch.utils.data.Dataset
     train_data
     block_size
     num_of_data_points
     collate_fn = None
     __len__()
     \__getitem__(idx)
class dicee.dataset_classes.OnevsAllDataset (train_set_idx: numpy.ndarray, entity_idxs)
     Bases: torch.utils.data.Dataset
     Dataset for the 1vsALL training strategy
```

```
• train_set_idx - Indexed triples for the training.
```

- entity_idxs mapping.
- relation_idxs mapping.
- form ?
- num_workers int for https://pytorch.org/docs/stable/data.html#torch.utils.data.
 DataLoader

Return type

torch.utils.data.Dataset

```
train_data
target_dim
collate_fn = None
__len__()
__getitem__(idx)
```

Bases: torch.utils.data.Dataset

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

Let D denote a dataset for KvsAll training and be defined as D:= $\{(x,y)_i\}_i ^N$, where x: (h,r) is an unique tuple of an entity h in E and a relation r in R that has been seed in the input graph. y: denotes a multi-label vector in $[0,1]^{\{E\}}$ is a binary label.

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



TODO

train_set_idx

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

entity_idxs

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

relation_idxs

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

self: torch.utils.data.Dataset

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

```
train_data = None
train_target = None
label_smoothing_rate
```

```
collate_fn = None
__len__()
\__getitem__(idx)
```

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

Bases: torch.utils.data.Dataset

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

Let D denote a dataset for AllvsAll training and be defined as D:= $\{(x,y)_i\}_i$ ^N, where x: (h,r) is a possible unique tuple of an entity h in E and a relation r in R. Hence $N = |E| \times |R|$ y; denotes a multi-label vector in $[0,1]^{[E]}$ is a binary label.

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



1 Note

AllvsAll extends KvsAll via none existing (h,r). Hence, it adds data points that are labelled

only with 0s.

train set idx

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

entity_idxs

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

relation_idxs

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

self: torch.utils.data.Dataset

```
>>> a = AllvsAll()
? 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
store
__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.

Туре

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
      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.
                    idx (int) – The index of the sample to retrieve.
                Returns
                    A tuple consisting of:
                      • x (torch.Tensor): The head and relation part of the triple.
                      • y_idx (torch.Tensor): The concatenated indices of the true object (tail entity) and the
                         indices of the negative samples.
                      • y_vec (torch.Tensor): A vector containing the labels for the positive and negative samples,
                         with label smoothing applied.
                Return type
                    tuple
class dicee.dataset_classes.KvsSampleDataset (train_set_idx: numpy.ndarray, entity_idxs,
             relation_idxs, form, store=None, neg_ratio=None, label_smoothing_rate: float = 0.0)
      Bases: torch.utils.data.Dataset
           KvsSample a Dataset:
                D := \{(x,y)_i\}_i ^N, where
                    . x:(h,r) is a unique h in E and a relation r in R and . y in [0,1]^{\{|E|\}} is a binary label.
      orall y_i = 1 s.t. (h r E_i) in KG
                At each mini-batch construction, we subsample(y), hence n
                    |new_y| << |E| new_y contains all 1's if sum(y)< neg_sample ratio new_y contains</pre>
           train_set_idx
                Indexed triples for the training.
           entity_idxs
                mapping.
           relation_idxs
                mapping.
           form
           store
           label_smoothing_rate
```

torch.utils.data.Dataset

```
train data = None
     train_target = None
     neg ratio
     num_entities
     label_smoothing_rate
     collate_fn = None
     store
     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.

1 Note

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

```
neg_sample_ratio
     train_set
     length
     num entities
     num relations
      __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, \text{ where }
                    . x:(h,r,t) in KG is a unique h in E and a relation r in R and . collact_fn => Generates
                   negative triples
               collect_fn:
     orall (h,r,t) in G obtain, create negative triples \{(h,r,x),(,r,t),(h,m,t)\}
               y:labels are represented in torch.float16
           train_set_idx
               Indexed triples for the training.
           entity_idxs
               mapping.
           relation_idxs
               mapping.
           form
           store
           label_smoothing_rate
           collate_fn: batch:List[torch.IntTensor] Returns ——- torch.utils.data.Dataset
     label_smoothing_rate
     neg_sample_ratio
     train_set
     length
     num_entities
     num_relations
     __len__()
     \__getitem__(idx)
     collate_fn (batch: List[torch.Tensor])
class dicee.dataset_classes.CVDataModule(train_set_idx: numpy.ndarray, num_entities,
            num_relations, neg_sample_ratio, batch_size, num_workers)
     Bases: pytorch_lightning.LightningDataModule
     Create a Dataset for cross validation
           Parameters
                 • train_set_idx - Indexed triples for the training.
                 • num_entities - entity to index mapping.
                 • num_relations - relation to index mapping.
                 • batch_size - int
                 • form - ?
```

• num workers int for https://pytorch.org/docs/stable/data.html#torch.utils.data. DataLoader

Return type

train set idx num_entities

num_relations

neg_sample_ratio

batch_size

num_workers

train_dataloader() → torch.utils.data.DataLoader

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

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

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

🛕 Warning

do not assign state in prepare_data

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

1 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, ...).

1 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 <code>self.trainer.training/testing/validating/predicting</code> 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)
        (continues on next page)
```

(continued from previous page)

```
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 alsomove_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.

A Warning

DO NOT set state to the model (use setup instead) since this is NOT called on every device

Example:

```
def prepare_data(self):
    # good
    download_data()
    tokenize()
    etc()

# bad
self.split = data_split
self.some_state = some_other_state()
```

In a distributed environment, prepare_data can be called in two ways (using prepare_data_per_node)

- 1. Once per node. This is the default and is only called on LOCAL_RANK=0.
- 2. Once in total. Only called on GLOBAL_RANK=0.

Example:

```
# DEFAULT
# called once per node on LOCAL_RANK=0 of that node
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = True
```

(continues on next page)

(continued from previous page)

```
# call on GLOBAL_RANK=0 (great for shared file systems)
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = False
```

This is called before requesting the dataloaders:

```
model.prepare_data()
initialize_distributed()
model.setup(stage)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
model.predict_dataloader()
```

dicee.eval_static_funcs

Functions

```
evaluate_link_prediction_performance(→
Dict)
evaluate_link_prediction_performance_with_.

evaluate_link_prediction_performance_with_i

evaluate_link_prediction_performance_with_i
...)
evaluate_lp_bpe_k_vs_all(model, triples[, er_vocab, ...])
```

Module Contents

Parameters

- model
- triples
- er_vocab
- re_vocab

Parameters

- model
- triples
- within_entities
- er_vocab
- re_vocab

dicee.evaluator

Classes

Evaluator

Evaluator class to evaluate KGE models in various downstream tasks

Module Contents

```
Class dicee.evaluator.Evaluator (args, is_continual_training=None)

Evaluator class to evaluate KGE models in various downstream tasks

Arguments

re_vocab = None

er_vocab = None

ee_vocab = None

func_triple_to_bpe_representation = None

is_continual_training

num_entities = None

num_relations = None

args

report

during_training = False

vocab_preparation (dataset) \rightarrow None

A function to wait future objects for the attributes of executor
```

Return type None

```
eval (dataset: dicee.knowledge_graph.KG, trained_model, form_of_labelling, during_training=False)
             \rightarrow None
eval_rank_of_head_and_tail_entity(*, train_set, valid_set=None, test_set=None, trained_model)
eval_rank_of_head_and_tail_byte_pair_encoded_entity(*, train_set=None, valid_set=None,
             test_set=None, ordered_bpe_entities, trained_model)
eval_with_byte (*, raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model,
             form\_of\_labelling) \rightarrow None
     Evaluate model after reciprocal triples are added
eval_with_bpe_vs_all (*, raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model,
            form\_of\_labelling) \rightarrow None
     Evaluate model after reciprocal triples are added
eval_with_vs_all (*, train_set, valid_set=None, test_set=None, trained_model, form_of_labelling)
              \rightarrow None
     Evaluate model after reciprocal triples are added
\verb|evaluate_lp_k_vs_all| (model, triple_idx, info=None, form_of_labelling=None)|
     Filtered link prediction evaluation. :param model: :param triple_idx: test triples :param info: :param
     form_of_labelling: :return:
evaluate_lp_with_byte (model, triples: List[List[str]], info=None)
evaluate_lp_bpe_k_vs_all (model, triples: List[List[str]], info=None, form_of_labelling=None)
         Parameters
              • model
              • triples (List of lists)
              • info
              • form_of_labelling
evaluate_lp (model, triple_idx, info: str)
dummy_eval (trained_model, form_of_labelling: str)
eval_with_data(dataset, trained_model, triple_idx: numpy.ndarray, form_of_labelling: str)
```

dicee.executer

Classes

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

Module Contents

class dicee.executer.Execute(args, continuous_training=False)

A class for Training, Retraining and Evaluation a model.

- (1) Loading & Preprocessing & Serializing input data.
- (2) Training & Validation & Testing

(3) Storing all necessary info
args
is_continual_training
trainer = None
trained_model = None
knowledge_graph = None
report
evaluator = None
start_time = None
setup_executor() → None
dept_read_preprocess_index_serialize_data() → None
 Read & Preprocess & Index & Serialize Input Data
 (1) Read or load the data from disk into memory.

Parameter

rtype

None

 ${\tt save_trained_model}\,()\,\to None$

Save a knowledge graph embedding model

(1) Send model to eval mode and cpu.

(2) Store the statistics of the data.

- (2) Store the memory footprint of the model.
- (3) Save the model into disk.
- (4) Update the stats of KG again?

Parameter

rtype

None

 $end(form_of_labelling: str) \rightarrow dict$

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

$write_report() \rightarrow None$

Report training related information in a report. json file

 $start() \rightarrow dict$

Start training

(1) Loading the Data # (2) Create an evaluator object. # (3) Create a trainer object. # (4) Start the training

Parameter

rtvpe

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.

previous_args

args

 $continual_start() \rightarrow dict$

Start Continual Training

- (1) Initialize training.
- (2) Start continual training.
- (3) Save trained model.

Parameter

rtype

A dict containing information about the training and/or evaluation

dicee.knowledge_graph

Classes

KG Knowledge Graph

Module Contents

```
class dicee.knowledge_graph.KG (dataset_dir: str = None, byte_pair_encoding: bool = False,
           padding: bool = False, add_noise_rate: float = None, sparql_endpoint: str = None,
           path\_single\_kg: str = None, path\_for\_deserialization: str = None, add\_reciprocal: bool = None,
           eval_model: str = None, read_only_few: int = None, sample_triples_ratio: float = None,
           path_for_serialization: str = None, entity_to_idx=None, relation_to_idx=None, backend=None,
           training\_technique: str = None, separator: str = None)
     Knowledge Graph
     dataset_dir
     sparql_endpoint
     path_single_kg
     byte_pair_encoding
     ordered_shaped_bpe_tokens = None
     add_noise_rate
     num_entities = None
     num_relations = None
     path_for_deserialization
     add_reciprocal
     eval_model
     read_only_few
     sample_triples_ratio
     path_for_serialization
     entity_to_idx
     relation_to_idx
     backend
     training_technique
     idx_entity_to_bpe_shaped
     enc
     num_tokens
     num_bpe_entities = None
     padding
     dummy_id
     max_length_subword_tokens = None
```

```
train_set_target = None

target_dim = None

train_target_indices = None

ordered_bpe_entities = None

separator

description_of_input = None

describe() \rightarrow None

property entities_str: List

property relations_str: List

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

__iter__()
__len__()

func_triple_to_bpe_representation (triple: List[str])
```

dicee.knowledge_graph_embeddings

Classes

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

Module Contents

```
class dicee.knowledge_graph_embeddings.KGE (path=None, url=None, construct_ensemble=False,
             model name=None)
      Bases: dicee.abstracts.BaseInteractiveKGE
      Knowledge Graph Embedding Class for interactive usage of pre-trained models
      __str__()
      to (device: str) \rightarrow None
      get_transductive_entity_embeddings (indices: torch.LongTensor | List[str], as_pytorch=False,
                   as\_numpy = False, as\_list = True) \rightarrow 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)
                    \rightarrow Tuple
           Given a relation and a tail entity, return top k ranked head entity.
           argmax_{e} in E  f(e,r,t), where r in R, t in E.
```

Parameter

```
relation: Union[List[str], str]
```

String representation of selected relations.

```
tail_entity: Union[List[str], str]
```

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

Given a head entity and a tail entity, return top k ranked relations.

```
argmax_{r in R} f(h,r,t), where h, t in E.
```

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

```
\label{eq:continuity} \begin{split} \texttt{predict\_missing\_tail\_entity} & (\textit{head\_entity: List[str]} \mid \textit{str}, \textit{relation: List[str]} \mid \textit{str}, \\ & \textit{within: List[str]} = \textit{None}) \rightarrow \textit{torch.FloatTensor} \end{split}
```

Given a head entity and a relation, return top k ranked entities

argmax_{e in E } f(h,r,e), where h in E and r in R.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

```
scores
```

```
predict(*, h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, within=None, logits=True) <math>\rightarrow torch. Float Tensor
```

Parameters

- logits
- h
- r
- t
- within

Predict missing item in a given triple.

Parameter

head_entity: Union[str, List[str]]

String representation of selected entities.

relation: Union[str, List[str]]

String representation of selected relations.

tail_entity: Union[str, List[str]]

String representation of selected entities.

k: int

Highest ranked k item.

Returns: Tuple

Highest K scores and items

```
\label{eq:core} \begin{split} \texttt{triple\_score} \ (h: List[str] \mid str = None, \, r: \, List[str] \mid str = None, \, t: \, List[str] \mid str = None, \, logits = False) \\ &\rightarrow \mathsf{torch.FloatTensor} \end{split}
```

Predict triple score

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

Returns: Tuple

```
pytorch tensor of triple score
```

```
t norm (tens 1: torch. Tensor, tens 2: torch. Tensor, tnorm: str = 'min') \rightarrow torch. Tensor
```

 $tensor_t_norm(subquery_scores: torch.FloatTensor, tnorm: str = 'min') \rightarrow torch.FloatTensor_t_norm(subquery_scores: torch.FloatTensor_t_norm(stresser)) + torch.FloatTensor_t_norm(stresser) + torch$

Compute T-norm over [0,1] ^{n imes d} where n denotes the number of hops and d denotes number of entities

 t_conorm (tens_1: torch. Tensor, tens_2: torch. Tensor, tconorm: str = 'min') \rightarrow torch. Tensor

 $negnorm(tens_1: torch.Tensor, lambda_: float, neg_norm: str = 'standard') \rightarrow torch.Tensor$

return_multi_hop_query_results (aggregated_query_for_all_entities, k: int, only_scores)

single_hop_query_answering(query: tuple, only_scores: bool = True, k: int = None)

```
answer_multi_hop_query (query_type: str = None, query: Tuple[str | Tuple[str, str], Ellipsis] = None, queries: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, tnorm: str = 'prod', neg_norm: str = 'standard', lambda_: float = 0.0, k: int = 10, only_scores=False)

→ List[Tuple[str, torch.Tensor]]
```

@TODO: Refactoring is needed # @TODO: Score computation for each query type should be done in a static function

Find an answer set for EPFO queries including negation and disjunction

Parameter

```
query_type: str The type of the query, e.g., "2p".
```

query: Union[str, Tuple[str, Tuple[str, str]]] The query itself, either a string or a nested tuple.

queries: List of Tuple[Union[str, Tuple[str, str]], ...]

tnorm: str The t-norm operator.

neg_norm: str The negation norm.

lambda_: float lambda parameter for sugeno and yager negation norms

k: int The top-k substitutions for intermediate variables.

returns

- List[Tuple[str, torch.Tensor]]
- · Entities and corresponding scores sorted in the descening order of scores

```
find_missing_triples (confidence: float, entities: List[str] = None, relations: List[str] = None, topk: int = 10, at_most: int = sys.maxsize) \rightarrow Set
```

Find missing triples

Iterative over a set of entities E and a set of relation R:

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

Return (e,r,x)

otin G and f(e,r,x) >confidence

```
confidence: float

A threshold for an output of a sigmoid function given a triple.

topk: int

Highest ranked k item to select triples with f(e,r,x) > confidence.

at_most: int

Stop after finding at_most missing triples
{(e,r,x) | f(e,r,x) > confidence land (e,r,x)

otin G

deploy (share: bool = False, top_k: int = 10)

train_triples (h: List[str], r: List[str], t: List[str], labels: List[float], iteration=2, optimizer=None)

train_k_vs_all (h, r, iteration=1, lr=0.001)

Train k vs all :param head_entity: :param relation: :param iteration: :param lr: :return:

train (kg, lr=0.1, epoch=10, batch_size=32, neg_sample_ratio=10, num_workers=1) → None

Retrained a pretrain model on an input KG via negative sampling.
```

dicee.query_generator

Classes

QueryGenerator

Module Contents

```
ent_in: Dict
ent_out: Dict
query_name_to_struct
list2tuple(list data)
tuple2list(x: List | Tuple) \rightarrow List | Tuple
     Convert a nested tuple to a nested list.
set_global_seed (seed: int)
     Set seed
construct\_graph(paths: List[str]) \rightarrow 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) \rightarrow bool
     Private method for fill_query logic.
achieve_answer (query: List[str | List], ent_in: Dict, ent_out: Dict) → set
     Private method for achieve_answer logic. @TODO: Document the code
write_links(ent_out, small_ent_out)
ground_queries (query_structure: List[str | List], ent_in: Dict, ent_out: Dict, small_ent_in: Dict,
             small_ent_out: Dict, gen_num: int, query_name: str)
     Generating queries and achieving answers
unmap (query_type, queries, tp_answers, fp_answers, fn_answers)
unmap_query (query_structure, query, id2ent, id2rel)
generate_queries (query_struct: List, gen_num: int, query_type: str)
     Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting
     queries and answers in return @ TODO: create a class for each single query struct
save_queries (query_type: str, gen_num: int, save_path: str)
abstract load_queries(path)
get_queries (query_type: str, gen_num: int)
static save_queries_and_answers (path: str, data: List[Tuple[str, Tuple[collections.defaultdict]]])
              \rightarrow None
     Save Queries into Disk
static load\_queries\_and\_answers (path: str) \rightarrow List[Tuple[str, Tuple[collections.defaultdict]]]
     Load Queries from Disk to Memory
```

dicee.sanity_checkers

Functions

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.static_funcs

Functions

<pre>create_recipriocal_triples(x)</pre>	Add inverse triples into dask dataframe
get_er_vocab(data[, file_path])	
(dotal flath))	
<pre>get_re_vocab(data[, file_path])</pre>	
get_ee_vocab(data[, file_path])	
-	
timeit(func)	
save_pickle(*[, data, file_path])	
parto_profite([, amm, mo_pmm])	
load_pickle([file_path])	
([61 ₂	
<pre>load_term_mapping([file_path])</pre>	
select_model(args[, is_continual_training, stor-	
age_path])	
$load_model(\rightarrow Tuple[object, Tuple[dict, dict]])$	Load weights and initialize pytorch module from names-
load_model_ensemble()	pace arguments Construct Ensemble Of weights and initialize pytorch
ioad_model_ensemble()	module from namespace arguments
save_numpy_ndarray(*, data, file_path)	ı C
numpy_data_type_changer(→ numpy.ndarray)	Detect most efficient data type for a given triples
$save_checkpoint_model(\rightarrow None)$ $store(\rightarrow None)$	Store Pytorch model into disk Store trained_model model and save embeddings into csv
Score(/ None)	file.
$add_noisy_triples(\rightarrow pandas.DataFrame)$	Add randomly constructed triples
read_or_load_kg(args, cls)	
intialize_model(→ Tuple[object, str])	
THETATIZE_HOUGET(→ Tupic[Object, Sti])	
load_json(→ dict)	
save_embeddings(→ None)	Save it as CSV if memory allows.
random_prediction(pre_trained_kge)	
deploy_triple_prediction(pre_trained_kge,	
str_subject,)	
	continues on next page

continues on next page

Table 2 - continued from previous page

```
deploy_tail_entity_prediction(pre_trained_kge,
...)
deploy_head_entity_prediction(pre_trained_kge,
...)
deploy_relation_prediction(pre_trained_kge,
...)
vocab_to_parquet(vocab_to_idx, name, ...)
create_experiment_folder([folder_name])
continual\_training\_setup\_executor(\rightarrow None)
exponential\_function(\rightarrow torch.FloatTensor)
load_numpy(\rightarrow numpy.ndarray)
evaluate(entity_to_idx,
                            scores,
                                       easy_answers,
                                                       # @TODO: CD: Renamed this function
hard answers)
download_file(url[, destination_folder])
download_files_from_url(\rightarrow None)
download_pretrained_model(\rightarrow str)
```

Module Contents

```
dicee.static_funcs.create_recipriocal_triples(x)
     Add inverse triples into dask dataframe :param x: :return:
dicee.static_funcs.get_er_vocab(data, file_path: str = None)
dicee.static_funcs.get_re_vocab(data, file_path: str = None)
dicee.static_funcs.get_ee_vocab(data, file_path: str = None)
dicee.static_funcs.timeit(func)
dicee.static_funcs.save_pickle(*, data: object = None, file_path=str)
dicee.static_funcs.load_pickle(file_path=str)
dicee.static_funcs.load_term_mapping(file_path=str)
dicee.static_funcs.select_model(args: dict, is_continual_training: bool = None,
           storage\_path: str = None
dicee.static_funcs.load_model(path_of_experiment_folder: str, model_name='model.pt', verbose=0)
            → Tuple[object, Tuple[dict, dict]]
     Load weights and initialize pytorch module from namespace arguments
dicee.static_funcs.load_model_ensemble(path_of_experiment_folder: str)
            → Tuple[dicee.models.base_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]
     Construct Ensemble Of weights and initialize pytorch module from namespace arguments
```

- (1) Detect models under given path
- (2) Accumulate parameters of detected models
- (3) Normalize parameters
- (4) Insert (3) into model.

```
dicee.static_funcs.save_numpy_ndarray (*, data: numpy.ndarray, file_path: str)
```

Detect most efficient data type for a given triples :param train_set: :param num: :return:

```
\texttt{dicee.static\_funcs.save\_checkpoint\_model} \ (\textit{model}, \textit{path: str}) \ \rightarrow None
```

Store Pytorch model into disk

```
dicee.static_funcs.store(trainer, trained_model, model_name: str = 'model', full_storage_path: str = None, save_embeddings_as_csv=False) \rightarrow None
```

Store trained_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param full_storage_path: path to save parameters. :param model_name: string representation of the name of the model. :param trained_model: an instance of BaseKGE see core.models.base_model . :param save_embeddings_as_csv: for easy access of embeddings. :return:

Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

```
dicee.static_funcs.read_or_load_kg (args, cls)  
dicee.static_funcs.intialize_model (args: dict, verbose=0) \rightarrow Tuple[object, str]  
dicee.static_funcs.load_json(p: str) \rightarrow dict
```

dicee.static_funcs.save_embeddings (embeddings: numpy.ndarray, indexes, path: $str) \rightarrow None$ Save it as CSV if memory allows.:param embeddings::param indexes::param path::return:

```
dicee.static_funcs.random_prediction(pre_trained_kge)
```

```
\label{local_condition} \mbox{disce.static\_funcs.deploy\_tail\_entity\_prediction} \mbox{ $(pre\_trained\_kge, str\_subject, str\_predicate, top\_k)$}
```

```
\label{local_condition} \begin{tabular}{ll} dice.static\_funcs.deploy\_head\_entity\_prediction (pre\_trained\_kge, str\_object, str\_predicate, \\ top\_k) \end{tabular}
```

```
dicee.static_funcs.deploy_relation_prediction(pre_trained_kge, str_subject, str_object, top_k)
```

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

dicee.static_funcs.create_experiment_folder(folder_name='Experiments')

```
dicee.static_funcs.continual_training_setup_executor(executor) \rightarrow None
```

 $\label{linear_discrete_discrete} \begin{tabular}{ll} \tt discrete_static_funcs.exponential_function (\it{x: numpy.ndarray, lam: float, ascending_order=True)} \\ \to torch. Float Tensor \end{tabular}$

```
dicee.static_funcs.load_numpy(path) \rightarrow 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-KvsA11")

dicee.static_funcs.download_pretrained_model($\mathit{url}: \mathit{str}$) \rightarrow str

dicee.static_funcs_training

Functions

```
evaluate_lp(model, triple_idx, num_entities, Evaluate model in a standard link prediction task
er_vocab, ...)
evaluate_bpe_lp(model, triple_idx, ...[, info])

efficient_zero_grad(model)
```

Module Contents

```
\label{local_condition} {\tt dicee.static\_funcs\_training.evaluate\_lp} \ (\textit{model, triple\_idx, num\_entities,} \\ \textit{er\_vocab: Dict[Tuple, List], re\_vocab: Dict[Tuple, List], info='Eval Starts')}
```

Evaluate model in a standard link prediction task

for each triple the rank is computed by taking the mean of the filtered missing head entity rank and the filtered missing tail entity rank :param model: :param triple_idx: :param info: :return:

dicee.static_funcs_training.efficient_zero_grad (model)

dicee.static_preprocess_funcs

Attributes

enable_log

Functions

```
timeit(func)
preprocesses\_input\_args(args) \qquad Sanity Checking in input arguments
create\_constraints(\rightarrow Tuple[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

- (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)
```

14.3 Attributes

```
__version__
```

14.4 Classes

D'-126 71	Each adding Eastities and Delations for Learning and Infor-
DistMult	Embedding Entities and Relations for Learning and Infer-
	ence in Knowledge Bases
KeciBase	Without learning dimension scaling

continues on next page

Table 3 - continued from previous page

Keci	Base class for all neural network modules.
TransE	Translating Embeddings for Modeling
DeCaL	Base class for all neural network modules.
DualE	Dual Quaternion Knowledge Graph Embeddings
	(https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)
ComplEx	Base class for all neural network modules.
AConEx	Additive Convolutional ComplEx Knowledge Graph Embeddings
AConvO	Additive Convolutional Octonion Knowledge Graph Embeddings
AConvQ	Additive Convolutional Quaternion Knowledge Graph Embeddings
ConvQ	Convolutional Quaternion Knowledge Graph Embeddings
ConvO	Base class for all neural network modules.
ConEx	Convolutional ComplEx Knowledge Graph Embeddings
QMult	Base class for all neural network modules.
OMult	Base class for all neural network modules.
Shallom	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
LFMult	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
PykeenKGE	A class for using knowledge graph embedding models implemented in Pykeen
BytE	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
DICE_Trainer	DICE_Trainer implement
KGE	Knowledge Graph Embedding Class for interactive usage of pre-trained models
Execute	A class for Training, Retraining and Evaluation a model.
BPE_NegativeSamplingDataset	An abstract class representing a Dataset.
MultiLabelDataset	An abstract class representing a Dataset.
MultiClassClassificationDataset	Dataset for the 1vsALL training strategy
OnevsAllDataset	Dataset for the 1vsALL training strategy
KvsAll	Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.
AllvsAll	Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.
OnevsSample	A custom PyTorch Dataset class for knowledge graph embeddings, which includes
KvsSampleDataset	KvsSample a Dataset:
NegSampleDataset	An abstract class representing a Dataset.
TriplePredictionDataset	Triple Dataset
CVDataModule	Create a Dataset for cross validation
QueryGenerator	

14.5 Functions

```
Add inverse triples into dask dataframe
create_recipriocal_triples(x)
get_er_vocab(data[, file_path])
get_re_vocab(data[, file_path])
get_ee_vocab(data[, file_path])
timeit(func)
save_pickle(*[, data, file_path])
load_pickle([file_path])
{\it load\_term\_mapping}([file\_path])
select_model(args[,
                         is_continual_training,
                                                 stor-
age_path])
load_model(→ Tuple[object, Tuple[dict, dict]])
                                                        Load weights and initialize pytorch module from names-
                                                        pace arguments
load_model_ensemble(...)
                                                        Construct Ensemble Of weights and initialize pytorch
                                                        module from namespace arguments
save_numpy_ndarray(*, data, file_path)
numpy\_data\_type\_changer(\rightarrow numpy.ndarray)
                                                        Detect most efficient data type for a given triples
                                                        Store Pytorch model into disk
save\_checkpoint\_model(\rightarrow None)
store(\rightarrow None)
                                                        Store trained model model and save embeddings into csv
add\_noisy\_triples(\rightarrow pandas.DataFrame)
                                                        Add randomly constructed triples
read_or_load_kg(args, cls)
intialize\_model(\rightarrow Tuple[object, str])
load_json(\rightarrow dict)
save\_embeddings(\rightarrow None)
                                                        Save it as CSV if memory allows.
random_prediction(pre_trained_kge)
deploy_triple_prediction(pre_trained_kge,
str_subject, ...)
deploy_tail_entity_prediction(pre_trained_kge,
deploy_head_entity_prediction(pre_trained_kge,
...)
deploy_relation_prediction(pre_trained_kge,
vocab_to_parquet(vocab_to_idx, name, ...)
create_experiment_folder([folder_name])
```

continues on next page

Table 4 - continued from previous page

```
continual\_training\_setup\_executor(\rightarrow None)
exponential\_function(\rightarrow torch.FloatTensor)
load_numpy(→ numpy.ndarray)
                                                        # @TODO: CD: Renamed this function
evaluate(entity_to_idx,
                             scores,
                                        easy_answers,
hard_answers)
download_file(url[, destination_folder])
download\_files\_from\_url(\rightarrow None)
download\_pretrained\_model(\rightarrow str)
mapping_from_first_two_cols_to_third(train_se
timeit(func)
load_term_mapping([file_path])
reload_dataset(path, form_of_labelling, ...)
                                                        Reload the files from disk to construct the Pytorch dataset
construct_dataset(→ torch.utils.data.Dataset)
```

14.6 Package Contents

```
class dicee.Pyke(args)
     Bases: dicee.models.base_model.BaseKGE
     A Physical Embedding Model for Knowledge Graphs
     name = 'Pyke'
     dist_func
     margin = 1.0
     forward_triples (x: torch.LongTensor)
              Parameters
                  x
class dicee.DistMult(args)
     Bases: dicee.models.base_model.BaseKGE
     Embedding Entities and Relations for Learning and Inference in Knowledge Bases https://arxiv.org/abs/1412.6575
     name = 'DistMult'
     k_vs_all_score (emb_h: torch.FloatTensor, emb_r: torch.FloatTensor, emb_E: torch.FloatTensor)
              Parameters
                  • emb_h
                  • emb_r
```

```
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.KeciBase (args)
Bases: Keci
Without learning dimension scaling
name = 'KeciBase'
requires_grad_for_interactions = False

class dicee.Keci (args)
Bases: dicee.models.base_model.BaseKGE
```

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model (nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
name = 'Keci'
p
q
```

eq j

```
r
requires_grad_for_interactions = True
compute\_sigma\_pp(hp, rp)
     Compute sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k
     sigma_{pp} captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute
     interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops
          results = [] for i in range(p - 1):
              for k in range(i + 1, p):
                results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])
          sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))
     Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1,
     e1e2, e1e3,
          e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
     Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.
compute\_sigma\_qq(hq, rq)
     Compute sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k sigma_{q}
     captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions
     between e 1 e 2, e 1 e 3, and e 2 e 3 This can be implemented with a nested two for loops
          results = [] for j in range(q - 1):
              for k in range(j + 1, q):
                results.append(hq[:,:,j]*rq[:,:,k] - hq[:,:,k]*rq[:,:,j]) \\
          sigma_q = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))
     Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1,
     e1e2, e1e3,
          e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
     Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.
compute sigma pq (*, hp, hq, rp, rq)
     sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j
     results = [] sigma pq = torch.zeros(b, r, p, q) for i in range(p):
          for j in range(q):
              sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
     print(sigma pq.shape)
apply_coefficients(hp, hq, rp, rq)
     Multiplying a base vector with its scalar coefficient
clifford_multiplication (h0, hp, hq, r0, rp, rq)
     Compute our CL multiplication
          h = h_0 + sum_{i=1}^p h_i e_i + sum_{j=p+1}^p h_j e_j r = r_0 + sum_{i=1}^p r_i e_i +
          sum_{j=p+1}^{p+q} r_j e_j
          ei ^2 = +1 for i =< i =< p ej ^2 = -1 for p < j =< p+q ei ej = -eje1 for i
```

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

- (1) $sigma_0 = h_0 r_0 + sum_{i=1}^p (h_0 r_i) e_i sum_{j=p+1}^{p+q} (h_j r_j) e_j$
- (2) $sigma_p = sum_{i=1}^p (h_0 r_i + h_i r_0) e_i$
- (3) $sigma_q = sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j$
- (4) $sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k h_k r_i) e_i e_k$
- (5) $sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k h_k r_j) e_j e_k$
- (6) $sigma_{pq} = sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j h_j r_i) e_i e_j$

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

→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

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

Parameter

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

returns

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

forward_k_vs_with_explicit(x: torch.Tensor)

k_vs_all_score (bpe_head_ent_emb, bpe_rel_ent_emb, E)

 $forward_k_vs_all(x: torch.Tensor) \rightarrow torch.FloatTensor$

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations mathbb $\{R\}^d$.
- (2) Construct head entity and relation embeddings according to $Cl_{p,q}(\mathsf{mathbb}\{R\}^d)$.
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

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

construct_batch_selected_cl_multivector(x: torch.FloatTensor, r: int, p: int, q: int)

→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

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

Parameter

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

returns

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

 $\textbf{forward_k_vs_sample} \ (x: torch.LongTensor, target_entity_idx: torch.LongTensor) \ \rightarrow \textbf{torch.FloatTensor}$

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) \rightarrow torch.FloatTensor
```

Parameter

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

rtype

torch.FloatTensor with (n) shape

```
class dicee.TransE(args)
```

Bases: dicee.models.base model.BaseKGE

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

```
name = 'TransE'

margin = 4

score (head_ent_emb, rel_ent_emb, tail_ent_emb)

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

class dicee.DeCaL (args)

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

name = 'DeCaL'

entity_embeddings

relation_embeddings

р

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) \longrightarrow 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,

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

and return:

$$sigma_0t = \sigma_0 \cdot t_0 = s0 + s1 - s2s3, s4ands5$$

compute_sigmas_multivect(list_h_emb, list_r_emb)

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

For same bases vectors interaction we have

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

For different base vector interactions, we have

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

 $forward_k_vs_all(x: torch.Tensor) \rightarrow torch.FloatTensor$

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations
- (2) Construct head entity and relation embeddings according to $Cl_{p,q, r}(mathbb{R}^d)$.
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

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

 $\verb"apply_coefficients" (h0, hp, hq, hk, r0, rp, rq, rk)$

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:

```
\label{eq:sigma_pp} $$ \sum_{p=1}^{p} \sup_{i=1}^{p-1}\sum_{i'=i+1}^{p} (x_iy_{i'}-x_{i'}) $$
```

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

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

```
for k in range(i + 1, p):
```

```
results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])
```

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

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

```
e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
```

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

 $compute_sigma_qq(hq, rq)$

Compute

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

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

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

for k in range(j + 1, q):

results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

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

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

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

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

for j in range(q):

$$sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]$$

print(sigma_pq.shape)

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

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

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

```
for j in range(q):
                                                                   sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
                                      print(sigma_pq.shape)
class dicee.DualE(args)
                    Bases: dicee.models.base_model.BaseKGE
                    Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/
                    16657)
                    name = 'DualE'
                    entity_embeddings
                    relation_embeddings
                    num_ent
                    \texttt{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\_4\_t, e\_6\_h, e\_7\_h, e\_8\_h, e\_1\_t, e\_2\_t, e\_3\_t, e\_4\_t, e\_4\_t, e\_6\_h, e\_7\_h, e\_8\_h, e\_1\_t, e\_6\_t, e\_6\_h, e\_7\_h, e\_8\_h, e\_1\_t, e\_8\_t, e\_8
                                                                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) \rightarrow \text{torch.tensor}
                                       KvsAll scoring function
                                      Input
                                      x: torch.LongTensor with (n, ) shape
                                       Output
                                       torch.FloatTensor with (n) shape
                    forward\_triples(idx\_triple: torch.tensor) \rightarrow 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) \rightarrow torch.tensor
                                       Transpose function
                                       Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)
```

```
class dicee.ComplEx(args)
```

Bases: dicee.models.base_model.BaseKGE

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

As per the example above, an <u>__init___()</u> call to the parent class must be made before assignment on the child.

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

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)

```
class dicee.AConEx(args)
```

 $Bases: \ \textit{dicee.models.base_model.BaseKGE}$

Additive Convolutional ComplEx Knowledge Graph Embeddings

```
name = 'AConEx'
     conv2d
     fc_num_input
     fc1
     norm_fc1
     bn_conv2d
     feature_map_dropout
     {\tt residual\_convolution}~(C\_1: Tuple[torch.Tensor, torch.Tensor],
                  C_2: Tuple[torch.Tensor, torch.Tensor]) \rightarrow 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) \rightarrow torch.FloatTensor
     forward\_triples(x: torch.Tensor) \rightarrow torch.FloatTensor
               Parameters
     forward_k_vs_sample (x: torch.Tensor, target_entity_idx: torch.Tensor)
class dicee.AConvO(args: dict)
     Bases: dicee.models.base_model.BaseKGE
     Additive Convolutional Octonion Knowledge Graph Embeddings
     name = 'AConvO'
     conv2d
     fc_num_input
     fc1
     bn_conv2d
     norm_fc1
     feature_map_dropout
     static octonion_normalizer(emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4,
                  emb_rel_e5, emb_rel_e6, emb_rel_e7)
     {\tt residual\_convolution}\,(O\_1,\,O\_2)
     forward\_triples(x: torch.Tensor) \rightarrow torch.Tensor
               Parameters
                   x
```

```
Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] =>
          [0.0,0.1,...,0.8], shape=> (1, |Entities|) Given a batch of head entities and relations => shape (size of batch,)
          Entities()
class dicee.AConvQ(args)
     Bases: dicee.models.base_model.BaseKGE
     Additive Convolutional Quaternion Knowledge Graph Embeddings
     name = 'AConvQ'
     entity_embeddings
     relation_embeddings
     conv2d
     fc_num_input
     fc1
     bn_conv1
     bn_conv2
     feature_map_dropout
     residual_convolution (Q_1, Q_2)
     forward\_triples(indexed\_triple: torch.Tensor) \rightarrow torch.Tensor
               Parameters
                   x
     forward_k_vs_all (x: torch.Tensor)
          Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] =>
          [0.0,0.1,...,0.8], shape=> (1, |Entities|) Given a batch of head entities and relations => shape (size of batch,|
          Entities()
class dicee.ConvQ(args)
     Bases: dicee.models.base_model.BaseKGE
     Convolutional Quaternion Knowledge Graph Embeddings
     name = 'ConvQ'
     entity_embeddings
     relation_embeddings
     conv2d
     fc_num_input
     fc1
     bn_conv1
     bn_conv2
```

forward_k_vs_all (x: torch.Tensor)

```
feature_map_dropout
```

```
residual_convolution (Q_1, Q_2)
```

 $forward_triples (indexed_triple: torch.Tensor) \rightarrow torch.Tensor$

Parameters

x

```
forward_k_vs_all (x: torch.Tensor)
```

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, |Entities|) Given a batch of head entities and relations => shape (size of batch,| Entities|)

class dicee.ConvO(args: dict)

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

Base class for all neural network modules.

Your models should also subclass this class.

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

```
import torch.nn as nn
import torch.nn.functional as F
class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to(), etc.

1 Note

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

Variables

training $(b \circ o 1)$ – 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) \rightarrow torch.Tensor
               Parameters
     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 l)
class dicee.ConEx(args)
     Bases: dicee.models.base model.BaseKGE
     Convolutional ComplEx Knowledge Graph Embeddings
     name = 'ConEx'
     conv2d
     fc_num_input
     fc1
     norm_fc1
     bn_conv2d
     feature_map_dropout
     residual_convolution (C_1: Tuple[torch.Tensor, torch.Tensor],
                  C_2: Tuple[torch.Tensor, torch.Tensor]) \rightarrow 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) \rightarrow torch.FloatTensor
     forward\_triples(x: torch.Tensor) \rightarrow torch.FloatTensor
               Parameters
     forward_k_vs_sample (x: torch.Tensor, target_entity_idx: torch.Tensor)
class dicee.QMult(args)
     Bases: dicee.models.base_model.BaseKGE
     Base class for all neural network modules.
```

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__ (self) -> 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 have their parameters converted too when you call to (), etc.



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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
\label{eq:name} \begin{tabular}{ll} name = 'QMult' \\ \\ explicit = True \\ \\ quaternion_multiplication_followed_by_inner_product $(h,r,t)$ \\ \\ \end{tabular}
```

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.

 $static quaternion_normalizer(x: torch.FloatTensor) \rightarrow torch.FloatTensor$

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:

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

Parameters

x – The vector.

Returns

The normalized vector.

 $k_vs_all_score$ (bpe_head_ent_emb, bpe_rel_ent_emb, E)

Parameters

- bpe_head_ent_emb
- bpe_rel_ent_emb
- E

forward_k_vs_all (X)

Parameters

x

 $forward_k_vs_sample(x, target_entity_idx)$

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples, i.e., [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, |Entities|) Given a batch of head entities and relations => shape (size of batch, |Entities|)

```
class dicee.OMult(args)
```

Bases: dicee.models.base_model.BaseKGE

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to (), etc.

1 Note

As per the example above, an <u>__init__()</u> 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 \text{ in Entities}] => [0.0,0.1,...,0.8], shape=> (1, |Entities|) Given a batch of head entities and
           relations => shape (size of batch, | Entities|)
class dicee.Shallom(args)
     Bases: dicee.models.base_model.BaseKGE
     A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
     name = 'Shallom'
     shallom_width
     shallom
     get_embeddings() → Tuple[numpy.ndarray, None]
     forward_k_vs_all (x) \rightarrow \text{torch.FloatTensor}
     forward\_triples(x) \rightarrow torch.FloatTensor
               Parameters
                   x
               Returns
class dicee.LFMult(args)
     Bases: dicee.models.base_model.BaseKGE
     Embedding with polynomial functions. We represent all entities and relations in the polynomial space as: f(x) =
     sum_{i=0}^{d-1} a_k x^{i}d and use the three differents scoring function as in the paper to evaluate the score.
     We also consider combining with Neural Networks.
     name = 'LFMult'
     entity_embeddings
     relation_embeddings
     degree
     m
     x_values
     forward_triples (idx_triple)
               Parameters
                   x
```

```
construct_multi_coeff(X)
```

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

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

linear(x, w, b)

$scalar_batch_NN(a, b, c)$

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

tri_score (coeff_h, coeff_r, coeff_t)

this part implement the trilinear scoring techniques:

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

vtp score (h, r, t)

this part implement the vector triple product scoring techniques:

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

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

$comp_func(h, r, t)$

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

polynomial (coeff, x, degree)

This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,...d] and return a vector tensor (coeff $[0][0] + \text{coeff}[0][1]x + ... + \text{coeff}[0][d]x^d$,

$$coeff[1][0] + coeff[1][1]x + ... + coeff[1][d]x^d$$

pop (coeff, x, degree)

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

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

$$coeff[1][0] + coeff[1][1]x + ... + coeff[1][d]x^d$$

class dicee.PykeenKGE(args: dict)

 $Bases: \ \textit{dicee.models.base_model.BaseKGE}$

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:

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) \rightarrow torch.FloatTensor
           # => Explicit version by this we can apply bn and dropout
           # (1) Retrieve embeddings of heads, relations and tails and apply Dropout & Normalization if given. h, r, t =
           self.get_triple_representation(x) \# (2) Reshape (1). if self.last_dim > 0:
                h = h.reshape(len(x), self.embedding\_dim, self.last\_dim) r = r.reshape(len(x), self.embedding\_dim,
                self.last_dim) t = t.reshape(len(x), self.embedding_dim, self.last_dim)
           # (3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice size=None, slice dim=0)
      abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)
class dicee.BytE(*args, **kwargs)
```

Bases: dicee.models.base_model.BaseKGE

Base class for all neural network modules.

Your models should also subclass this class.

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

```
import torch.nn as nn
import torch.nn.functional as F
class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to(), etc.



1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
name = 'BytE'
config
temperature = 0.5
topk = 2
transformer
lm_head
weight
loss_function(yhat_batch, y_batch)
```

Parameters

- yhat_batch
- y_batch

forward(x: torch.LongTensor)

Parameters

```
\mathbf{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()
```

1 Note

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

class dicee.BaseKGE (args: dict)

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self) -> 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 have their parameters converted too when you call to(), etc.

1 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
learning_rate = None
apply_unit_norm = None
input_dropout_rate = None
hidden_dropout_rate = None
optimizer_name = None
feature_map_dropout_rate = None
kernel_size = None
num_of_output_channels = None
weight_decay = None
loss
selected_optimizer = None
```

```
normalizer_class = None
normalize_head_entity_embeddings
normalize_relation_embeddings
normalize_tail_entity_embeddings
hidden_normalizer
param_init
input_dp_ent_real
input_dp_rel_real
hidden_dropout
loss_history = []
byte_pair_encoding
max_length_subword_tokens
block_size
forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)
        Parameters
            x (B x 2 x T)
forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors
        Parameters
init_params_with_sanity_checking()
forward(x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
           y_idx: torch.LongTensor = None)
        Parameters
            • x
            • y_idx
            • ordered_bpe_entities
forward\_triples(x: torch.LongTensor) \rightarrow torch.Tensor
        Parameters
forward_k_vs_all(*args, **kwargs)
forward_k_vs_sample(*args, **kwargs)
get_triple_representation(idx_hrt)
{\tt 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]
dicee.create_recipriocal_triples(x)
     Add inverse triples into dask dataframe :param x: :return:
dicee.get_er_vocab(data, file_path: str = None)
dicee.get_re_vocab(data, file_path: str = None)
dicee.get_ee_vocab(data, file_path: str = None)
dicee.timeit(func)
dicee.save_pickle(*, data: object = None, file_path=str)
dicee.load_pickle(file_path=str)
dicee.load_term_mapping(file_path=str)
dicee.select_model(args: dict, is_continual_training: bool = None, storage_path: str = None)
dicee.load_model(path_of_experiment_folder: str, model_name='model.pt', verbose=0)
             → Tuple[object, Tuple[dict, dict]]
     Load weights and initialize pytorch module from namespace arguments
dicee.load_model_ensemble(path_of_experiment_folder: str)
             → Tuple[dicee.models.base_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]
     Construct Ensemble Of weights and initialize pytorch module from namespace arguments
       (1) Detect models under given path
       (2) Accumulate parameters of detected models
       (3) Normalize parameters
       (4) Insert (3) into model.
dicee.save_numpy_ndarray(*, data: numpy.ndarray, file_path: str)
\texttt{dicee.numpy\_data\_type\_changer} (\textit{train\_set: numpy.ndarray}, \textit{num: int}) \rightarrow \texttt{numpy.ndarray}
     Detect most efficient data type for a given triples :param train_set: :param num: :return:
dicee.save_checkpoint_model(model, path: str) \rightarrow None
     Store Pytorch model into disk
```

```
dicee.store(trainer, trained_model, model_name: str = 'model', full_storage_path: str = None,
            save embeddings as csv=False) \rightarrow None
      Store trained_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param
      full_storage_path: path to save parameters. :param model_name: string representation of the name of the model.
      :param trained_model: an instance of BaseKGE see core.models.base_model . :param save_embeddings_as_csv:
      for easy access of embeddings. :return:
dicee.add\_noisy\_triples (train_set: pandas.DataFrame, add_noise_rate: float) \rightarrow pandas.DataFrame
      Add randomly constructed triples :param train set: :param add noise rate: :return:
dicee.read_or_load_kg(args, cls)
dicee.intialize_model(args: dict, verbose=0) → Tuple[object, str]
dicee.load_json(p: str) \rightarrow dict
dicee.save_embeddings(embeddings: numpy.ndarray, indexes, path: str) \rightarrow None
      Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:
dicee.random_prediction(pre_trained_kge)
dicee.deploy_triple_prediction(pre_trained_kge, str_subject, str_predicate, str_object)
dicee.deploy_tail_entity_prediction(pre_trained_kge, str_subject, str_predicate, top_k)
dicee.deploy_head_entity_prediction(pre_trained_kge, str_object, str_predicate, top_k)
dicee.deploy_relation_prediction(pre_trained_kge, str_subject, str_object, top_k)
dicee.vocab_to_parquet(vocab_to_idx, name, path_for_serialization, print_into)
dicee.create_experiment_folder(folder_name='Experiments')
{\tt dicee.continual\_training\_setup\_executor}(\textit{executor}) \rightarrow None
dicee.exponential_function (x: numpy.ndarray, lam: float, ascending\_order=True) \rightarrow torch.FloatTensor
dicee.load_numpy(path) \rightarrow numpy.ndarray
dicee.evaluate(entity_to_idx, scores, easy_answers, hard_answers)
      # @TODO: CD: Renamed this function Evaluate multi hop query answering on different query types
dicee.download_file(url, destination_folder='.')
dicee.download_files_from_url(base\_url: str, destination\_folder='.') \rightarrow 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")

 ${\tt dicee.download_pretrained_model}\,(\mathit{url}:\mathit{str})\,\to \mathit{str}$

class dicee.DICE_Trainer(args, is_continual_training, 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
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) \rightarrow lightning.Trainer
     Initialize Trainer from input arguments
initialize_or_load_model()
\verb"init_dataloader" (dataset: torch.utils.data.Dataset") 	o torch.utils.data.DataLoader
\verb"init_dataset"() \rightarrow 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) \rightarrow Tuple[dicee.models.base\_model.BaseKGE, str]
```

Perform K-fold Cross-Validation

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

Parameters

- self
- dataset

Returns

model

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

Bases: dicee.abstracts.BaseInteractiveKGE

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)
```

Given a relation and a tail entity, return top k ranked head entity.

 $argmax_{e} in E$ f(e,r,t), where r in R, t in E.

Parameter

relation: Union[List[str], str]

String representation of selected relations.

tail_entity: Union[List[str], str]

String representation of selected entities.

k: int

Highest ranked k entities.

```
Returns: Tuple
```

```
Highest K scores and entities
```

```
\label{limitsing_relations} \begin{split} &\texttt{predict\_missing\_relations} \ (\textit{head\_entity: List[str]} \mid \textit{str, tail\_entity: List[str]} \mid \textit{str, within=None}) \\ &\rightarrow \mathsf{Tuple} \end{split}
```

Given a head entity and a tail entity, return top k ranked relations.

```
argmax_{r in R} f(h,r,t), where h, t in E.
```

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

```
\verb|predict_missing_tail_entity| (\textit{head\_entity: List[str]} \mid \textit{str}, \textit{relation: List[str]} \mid \textit{str},
```

within: List[str] = None \rightarrow torch.FloatTensor

Given a head entity and a relation, return top k ranked entities

 $argmax_{e} in E$ f(h,r,e), where h in E and r in R.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

scores

 $predict(*, h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, within=None, logits=True) <math>\rightarrow$ torch.FloatTensor

Parameters

- logits
- h
- r
- t
- within

```
predict_topk(*, h: str \mid List[str] = None, r: str \mid List[str] = None, t: str \mid List[str] = None, topk: int = 10,
              within: List[str] = None)
      Predict missing item in a given triple.
      Parameter
      head entity: Union[str, List[str]]
      String representation of selected entities.
      relation: Union[str, List[str]]
      String representation of selected relations.
      tail_entity: Union[str, List[str]]
      String representation of selected entities.
      k: int
      Highest ranked k item.
      Returns: Tuple
      Highest K scores and items
 \texttt{triple\_score} \ (h: List[str] \mid str = None, \, r: \, List[str] \mid str = None, \, t: \, List[str] \mid str = None, \, logits = False) 
               \rightarrow torch.FloatTensor
      Predict triple score
      Parameter
      head_entity: List[str]
      String representation of selected entities.
      relation: List[str]
      String representation of selected relations.
      tail_entity: List[str]
      String representation of selected entities.
      logits: bool
      If logits is True, unnormalized score returned
      Returns: Tuple
      pytorch tensor of triple score
t_norm(tens\_1: torch.Tensor, tens\_2: torch.Tensor, tnorm: str = 'min') \rightarrow torch.Tensor
tensor_t_norm(subquery\_scores: torch.FloatTensor, tnorm: str = 'min') \rightarrow torch.FloatTensor
      Compute T-norm over [0,1] ^{n imes d} where n denotes the number of hops and d denotes number of
      entities
t\_conorm (tens_1: torch.Tensor, tens_2: torch.Tensor, tconorm: str = 'min') \rightarrow torch.Tensor
```

 $negnorm(tens_1: torch.Tensor, lambda_: float, neg_norm: str = 'standard') \rightarrow torch.Tensor$

return_multi_hop_query_results(aggregated_query_for_all_entities, k: int, only_scores)

```
single_hop_query_answering(query: tuple, only_scores: bool = True, k: int = None)
answer_multi_hop_query (query_type: str = None, query: Tuple[str | Tuple[str, str], Ellipsis] = None,
              queries: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, tnorm: str = 'prod',
             neg norm: str = 'standard', lambda : float = 0.0, k: int = 10, only scores=False)
              → List[Tuple[str, torch.Tensor]]
     # @TODO: Refactoring is needed # @TODO: Score computation for each query type should be done in a
     static function
     Find an answer set for EPFO queries including negation and disjunction
     Parameter
     query_type: str The type of the query, e.g., "2p".
     query: Union[str, Tuple[str, Tuple[str, str]]] The query itself, either a string or a nested tuple.
     queries: List of Tuple[Union[str, Tuple[str, str]], ...]
     tnorm: str The t-norm operator.
     neg_norm: str The negation norm.
     lambda_: float lambda parameter for sugeno and yager negation norms
     k: int The top-k substitutions for intermediate variables.
          returns
               • List[Tuple[str, torch.Tensor]]
               • Entities and corresponding scores sorted in the descening order of scores
find_missing_triples (confidence: float, entities: List[str] = None, relations: List[str] = None,
              topk: int = 10, at_most: int = sys.maxsize) \rightarrow 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.
          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
deploy(share: bool = False, top_k: int = 10)
train_triples (h: List[str], r: List[str], t: List[str], labels: List[float], iteration=2, optimizer=None)
```

```
train_k_vs_all(h, r, iteration=1, lr=0.001)
```

Train k vs all :param head_entity: :param relation: :param iteration: :param lr: :return:

train (kg, lr=0.1, epoch=10, $batch_size=32$, $neg_sample_ratio=10$, $num_workers=1$) \rightarrow None Retrained a pretrain model on an input KG via negative sampling.

class dicee.Execute(args, continuous training=False)

A class for Training, Retraining and Evaluation a model.

- (1) Loading & Preprocessing & Serializing input data.
- (2) Training & Validation & Testing
- (3) Storing all necessary info

args

```
is_continual_training
```

trainer = None

trained_model = None

knowledge_graph = None

report

evaluator = None

start_time = None

 $\mathtt{setup_executor}\left(\right) \to None$

 ${\tt dept_read_preprocess_index_serialize_data}\,()\,\to None$

Read & Preprocess & Index & Serialize Input Data

- (1) Read or load the data from disk into memory.
- (2) Store the statistics of the data.

Parameter

rtype

None

 ${\tt save_trained_model}\,()\,\to None$

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) \rightarrow 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

```
\textbf{write\_report}\,(\,)\,\to None
```

Report training related information in a report. json file

 $start() \rightarrow dict$

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

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.

1 Note

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

```
train_set
  ordered_bpe_entities
  num_bpe_entities
  neg_ratio
  num_datapoints
  __len__()
  __getitem__(idx)
  collate_fn (batch_shaped_bpe_triples: List[Tuple[torch.Tensor, torch.Tensor]])
class dicee.MultiLabelDataset (train_set: torch.LongTensor, train_indices_target: torch.LongTensor, target_dim: int, torch_ordered_shaped_bpe_entities: torch.LongTensor)
  Bases: torch.utils.data.Dataset
An abstract class representing a Dataset.
```

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

Note

DataLoader by default constructs an index sampler that yields integral indices. To make it work with a mapstyle 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)
```

```
Bases: torch.utils.data.Dataset
     Dataset for the 1vsALL training strategy
          Parameters
                • train_set_idx - Indexed triples for the training.
                • entity_idxs - mapping.
                • relation_idxs - mapping.
                • form - ?
                • num_workers - int
                                           for
                                                 https://pytorch.org/docs/stable/data.html#torch.utils.data.
                  DataLoader
          Return type
              torch.utils.data.Dataset
     train_data
     block_size
     num_of_data_points
     collate_fn = None
     __len__()
     \__{\texttt{getitem}} (idx)
class dicee.OnevsAllDataset(train_set_idx: numpy.ndarray, entity_idxs)
     Bases: torch.utils.data.Dataset
     Dataset for the 1vsALL training strategy
          Parameters
                • train_set_idx - Indexed triples for the training.
                • entity_idxs - mapping.
                • relation_idxs - mapping.
                • form - ?
                                          for https://pytorch.org/docs/stable/data.html#torch.utils.data.
                • num_workers - int
                  DataLoader
          Return type
              torch.utils.data.Dataset
     train_data
     target_dim
     collate_fn = None
     __len__()
     \__getitem__(idx)
```

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

class dicee. KvsAll (train_set_idx: numpy.ndarray, entity_idxs, relation_idxs, form, store=None, label_smoothing_rate: float = 0.0)

Bases: torch.utils.data.Dataset

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

Let D denote a dataset for KvsAll training and be defined as D:= $\{(x,y)_i\}_i ^N$, where x: (h,r) is an unique tuple of an entity h in E and a relation r in R that has been seed in the input graph. y: denotes a multi-label vector in $[0,1]^{\{E\}}$ is a binary label.

orall y i = 1 s.t. (h r E i) in KG



TODO

train_set_idx

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

entity idxs

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

relation_idxs

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

self: torch.utils.data.Dataset

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

train_data = None
train_target = None
label_smoothing_rate
collate_fn = None
__len__()

 $__getitem__(idx)$

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

Bases: torch.utils.data.Dataset

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

Let D denote a dataset for AllvsAll training and be defined as D:= $\{(x,y)_i\}_i^n N$, 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 E_i) in KG

1 Note

AllysAll extends KysAll via none existing (h,r). Hence, it adds data points that are labelled without 1s,

only with 0s.

train_set_idx

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

entity_idxs

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

relation_idxs

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

self: torch.utils.data.Dataset

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

```
train_data = None
train_target = None
label_smoothing_rate
collate_fn = None
target_dim
store
__len__()
__getitem__(idx)
```

class dicee.OnevsSample(train_set: numpy.ndarray, num_entities, num_relations,

neg sample ratio: int = None, label smoothing rate: float = 0.0)

Bases: torch.utils.data.Dataset

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

Parameters

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

```
train_data
     The input data converted into a PyTorch tensor.
         Type
             torch.Tensor
num_entities
     Number of entities in the dataset.
         Type
             int
num_relations
     Number of relations in the dataset.
         Type
             int
neg_sample_ratio
     Ratio of negative samples to be drawn for each positive sample.
         Type
             int
label_smoothing_rate
     The smoothing factor applied to the labels.
         Type
             torch.Tensor
collate_fn
     A function that can be used to collate data samples into batches (set to None by default).
         Type
             function, optional
train_data
num_entities
num_relations
neg_sample_ratio
label_smoothing_rate
collate_fn = None
```

Returns the number of samples in the dataset.

 $__{getitem}_{_}(idx)$

__len__()

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
```

__len__()

```
class dicee.KvsSampleDataset(train_set_idx: numpy.ndarray, entity_idxs, relation_idxs, form,
            store=None, neg_ratio=None, label_smoothing_rate: float = 0.0)
     Bases: torch.utils.data.Dataset
           KvsSample a Dataset:
               D := \{(x,y)_i\}_i \ ^N, where
                   . x:(h,r) is a unique h in E and a relation r in R and . y in [0,1]^{\{|E|\}} is a binary label.
     orall y_i = 1 s.t. (h r E_i) in KG
               At each mini-batch construction, we subsample(y), hence n
                   lnew_yl << |E| new_y contains all 1's if sum(y)< neg_sample ratio new_y contains</pre>
           train_set_idx
               Indexed triples for the training.
           entity_idxs
               mapping.
           relation_idxs
               mapping.
           form
           store
           label_smoothing_rate
           torch.utils.data.Dataset
     train_data = None
     train_target = None
     neg_ratio
     num_entities
     label_smoothing_rate
     collate_fn = None
     store
     max_num_of_classes
```

```
\__getitem_(idx)
```

class dicee.NegSampleDataset ($train_set$: numpy.ndarray, $num_entities$: int, $num_relations$: int, $neg\ sample\ ratio$: int = 1)

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

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

1 Note

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

```
neg_sample_ratio
      train_set
      length
      num_entities
      num_relations
      __len__()
      \underline{\phantom{a}}getitem\underline{\phantom{a}} (idx)
class dicee. TriplePredictionDataset (train_set: numpy.ndarray, num_entities: int, num_relations: int,
              neg sample ratio: int = 1, label smoothing rate: float = 0.0)
      Bases: torch.utils.data.Dataset
            Triple Dataset
                 D := \{(x)_i\}_i \ ^N, \text{ where }
                     . x:(h,r, t) in KG is a unique h in E and a relation r in R and . collact fn => Generates
                     negative triples
                 collect_fn:
      orall (h,r,t) in G obtain, create negative triples \{(h,r,x),(r,t),(h,m,t)\}
                 y:labels are represented in torch.float16
            train_set_idx
                 Indexed triples for the training.
            entity_idxs
                 mapping.
            relation_idxs
                 mapping.
            form
```

```
store
          label_smoothing_rate
          collate_fn: batch:List[torch.IntTensor] Returns ——- torch.utils.data.Dataset
     label_smoothing_rate
     neg_sample_ratio
     train_set
     length
     num_entities
     num_relations
     __len__()
     \__{\texttt{getitem}} (idx)
     collate_fn (batch: List[torch.Tensor])
class dicee. CVDataModule (train_set_idx: numpy.ndarray, num_entities, num_relations, neg_sample_ratio,
            batch_size, num_workers)
     Bases: pytorch_lightning.LightningDataModule
     Create a Dataset for cross validation
          Parameters
                • train_set_idx - Indexed triples for the training.
                • num_entities - entity to index mapping.
                • num_relations - relation to index mapping.
                • batch_size - int
                • form - ?
                                          for https://pytorch.org/docs/stable/data.html#torch.utils.data.
                • num_workers - int
                  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.

will dataloader return not be reloaded unless :paramyou 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.

A Warning

do not assign state in prepare_data

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

1 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.

```
stage - either 'fit', 'validate', 'test', or 'predict'
```

Example:

```
class LitModel(...):
   def __init__(self):
        self.11 = 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, ...).



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.

A Warning

DO NOT set state to the model (use setup instead) since this is NOT called on every device

Example:

```
def prepare_data(self):
    # good
    download_data()
    tokenize()
    etc()

# bad
self.split = data_split
self.some_state = some_other_state()
```

In a distributed environment, prepare_data can be called in two ways (using prepare_data_per_node)

- 1. Once per node. This is the default and is only called on LOCAL_RANK=0.
- 2. Once in total. Only called on GLOBAL_RANK=0.

Example:

```
# DEFAULT
# called once per node on LOCAL_RANK=0 of that node
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = True

# call on GLOBAL_RANK=0 (great for shared file systems)
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = False
```

This is called before requesting the dataloaders:

```
model.prepare_data()
initialize_distributed()
model.setup(stage)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
model.predict_dataloader()
```

class dicee.QueryGenerator(train_path, val_path: str, test_path: str, ent2id: Dict = None, rel2id: Dict = None, seed: int = 1, gen_valid: bool = False, gen_test: bool = True)

```
train_path
val_path
test_path
gen_valid
gen_test
seed
max_ans_num = 1000000.0
mode
ent2id
rel2id: Dict
ent_in: Dict
ent_out: Dict
query_name_to_struct
list2tuple(list data)
tuple2list(x: List | Tuple) \rightarrow List | Tuple
     Convert a nested tuple to a nested list.
set_global_seed (seed: int)
     Set seed
construct\_graph(paths: List[str]) \rightarrow 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) \rightarrow bool
     Private method for fill_query logic.
\verb"achieve_answer" (query: List[str \mid List], ent\_in: Dict, ent\_out: Dict) \rightarrow set
     Private method for achieve_answer logic. @TODO: Document the code
write_links (ent_out, small_ent_out)
ground_queries (query_structure: List[str | List], ent_in: Dict, ent_out: Dict, small_ent_in: Dict,
             small_ent_out: Dict, gen_num: int, query_name: str)
     Generating queries and achieving answers
unmap (query_type, queries, tp_answers, fp_answers, fn_answers)
unmap_query (query_structure, query, id2ent, id2rel)
generate_queries (query_struct: List, gen_num: int, query_type: str)
     Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting
     queries and answers in return @ TODO: create a class for each single query struct
save_queries (query_type: str, gen_num: int, save_path: str)
abstract load_queries(path)
```

Python Module Index

d

```
dicee, 12
dicee.__main__,112
dicee.abstracts, 112
dicee.analyse_experiments, 118
dicee.callbacks, 120
dicee.config, 126
dicee.dataset_classes, 129
dicee.eval_static_funcs, 141
dicee.evaluator, 142
dicee.executer, 143
dicee.knowledge_graph, 145
dicee.knowledge_graph_embeddings, 147
dicee.models, 12
dicee.models.base_model, 12
dicee.models.clifford, 21
dicee.models.complex, 28
dicee.models.dualE, 30
dicee.models.function_space, 32
dicee.models.octonion, 35
dicee.models.pykeen_models, 38
dicee.models.quaternion, 39
dicee.models.real, 42
dicee.models.static_funcs, 44
dicee.models.transformers, 44
dicee.query_generator, 151
dicee.read_preprocess_save_load_kg, 97
dicee.read_preprocess_save_load_kg.preprocess,
dicee.read_preprocess_save_load_kg.read_from_disk,
dicee.read_preprocess_save_load_kg.save_load_disk,
dicee.read_preprocess_save_load_kg.util, 99
dicee.sanity_checkers, 152
dicee.scripts, 105
dicee.scripts.index, 105
dicee.scripts.run, 105
dicee.scripts.serve, 105
dicee.static_funcs, 153
dicee.static_funcs_training, 156
dicee.static_preprocess_funcs, 156
dicee.trainer, 106
dicee.trainer.dice_trainer, 106
dicee.trainer.torch_trainer, 108
dicee.trainer.torch_trainer_ddp, 109
```

Index

Non-alphabetical

```
__call__() (dicee.models.base_model.IdentityClass method), 21
 _call__() (dicee.models.IdentityClass method), 61, 72, 78
__getitem__() (dicee.AllvsAll method), 195
__getitem__() (dicee.BPE_NegativeSamplingDataset method), 192
__getitem__() (dicee.dataset_classes.AllvsAll method), 133
__getitem__() (dicee.dataset_classes.BPE_NegativeSamplingDataset method), 130
__getitem__() (dicee.dataset_classes.KvsAll method), 133
__getitem__() (dicee.dataset_classes.KvsSampleDataset method), 136
__getitem__() (dicee.dataset_classes.MultiClassClassificationDataset method), 131
__getitem__() (dicee.dataset_classes.MultiLabelDataset method), 131
__getitem__() (dicee.dataset_classes.NegSampleDataset method), 136
__getitem__() (dicee.dataset_classes.OnevsAllDataset method), 132
__getitem__() (dicee.dataset_classes.OnevsSample method), 135
__getitem__() (dicee.dataset_classes.TriplePredictionDataset method), 137
__getitem__() (dicee.KvsAll method), 194
__getitem__() (dicee.KvsSampleDataset method), 197
__getitem__() (dicee.MultiClassClassificationDataset method), 193
__getitem__() (dicee.MultiLabelDataset method), 192
__getitem__() (dicee.NegSampleDataset method), 198
__getitem__() (dicee.OnevsAllDataset method), 193
__getitem__() (dicee.OnevsSample method), 196
__getitem__() (dicee.TriplePredictionDataset method), 199
__iter__() (dicee.config.Namespace method), 129
__iter__() (dicee.knowledge_graph.KG method), 147
__len__() (dicee.AllvsAll method), 195
  _len__() (dicee.BPE_NegativeSamplingDataset method), 192
__len__() (dicee.dataset_classes.AllvsAll method), 133
__len__() (dicee.dataset_classes.BPE_NegativeSamplingDataset method), 130
__len__() (dicee.dataset_classes.KvsAll method), 133
__len__() (dicee.dataset_classes.KvsSampleDataset method), 136
  _len__() (dicee.dataset_classes.MultiClassClassificationDataset method), 131
__len__() (dicee.dataset_classes.MultiLabelDataset method), 131
__len__() (dicee.dataset_classes.NegSampleDataset method), 136
__len__() (dicee.dataset_classes.OnevsAllDataset method), 132
__len__() (dicee.dataset_classes.OnevsSample method), 135
__len__() (dicee.dataset_classes.TriplePredictionDataset method), 137
__len__() (dicee.knowledge_graph.KG method), 147
__len__() (dicee.KvsAll method), 194
__len__() (dicee.KvsSampleDataset method), 197
__len__() (dicee.MultiClassClassificationDataset method), 193
__len__() (dicee.MultiLabelDataset method), 192
__len__() (dicee.NegSampleDataset method), 198
__len__() (dicee.OnevsAllDataset method), 193
__len__() (dicee.OnevsSample method), 196
  _len__() (dicee.TriplePredictionDataset method), 199
__str__() (dicee.KGE method), 186
__str__() (dicee.knowledge_graph_embeddings.KGE method), 147
__version__ (in module dicee), 204
AbstractCallback (class in dicee.abstracts), 116
AbstractPPECallback (class in dicee.abstracts), 117
AbstractTrainer (class in dicee.abstracts), 113
AccumulateEpochLossCallback (class in dicee.callbacks), 120
achieve_answer() (dicee.query_generator.QueryGenerator method), 152
achieve_answer() (dicee.QueryGenerator method), 203
AConEx (class in dicee), 169
AConEx (class in dicee.models), 67
AConEx (class in dicee.models.complex), 29
AConvO (class in dicee), 170
AConvO (class in dicee.models), 80
AConvO (class in dicee.models.octonion), 37
AConvo (class in dicee), 171
AConvQ (class in dicee.models), 74
```

```
AConvo (class in dicee.models.quaternion), 42
adaptive_swa (dicee.config.Namespace attribute), 129
add_new_entity_embeddings() (dicee.abstracts.BaseInteractiveKGE method), 116
add_noise_rate (dicee.config.Namespace attribute), 127
add_noise_rate (dicee.knowledge_graph.KG attribute), 146
add_noisy_triples() (in module dicee), 184
add_noisy_triples() (in module dicee.static_funcs), 155
add_noisy_triples_into_training() (dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method), 99
add_noisy_triples_into_training() (dicee.read_preprocess_save_load_kg.ReadFromDisk method), 104
\verb"add_reciprocal" (\textit{dicee.knowledge\_graph.KG attribute}), 146
AllvsAll (class in dicee), 194
AllvsAll (class in dicee.dataset_classes), 133
alphas (dicee.abstracts.AbstractPPECallback attribute), 118
alphas (dicee.callbacks.ASWA attribute), 123
analyse() (in module dicee.analyse_experiments), 120
answer_multi_hop_query() (dicee.KGE method), 189
answer_multi_hop_query() (dicee.knowledge_graph_embeddings.KGE method), 150
app (in module dicee.scripts.serve), 106
apply_coefficients() (dicee.DeCaL method), 166
apply_coefficients() (dicee.Keci method), 162
apply_coefficients() (dicee.models.clifford.DeCaL method), 26
apply_coefficients() (dicee.models.clifford.Keci method), 23
apply_coefficients() (dicee.models.DeCaL method), 86
apply_coefficients() (dicee.models.Keci method), 82
apply_reciprical_or_noise() (in module dicee.read_preprocess_save_load_kg.util), 102
apply_semantic_constraint (dicee.abstracts.BaseInteractiveKGE attribute), 114
apply_unit_norm (dicee.BaseKGE attribute), 181
apply_unit_norm (dicee.models.base_model.BaseKGE attribute), 19
apply_unit_norm (dicee.models.BaseKGE attribute), 58, 62, 65, 70, 76, 88, 92
args (dicee.BaseKGE attribute), 181
args (dicee.DICE_Trainer attribute), 185
args (dicee.evaluator.Evaluator attribute), 142
args (dicee.Execute attribute), 190
args (dicee.executer.ContinuousExecute attribute), 145
args (dicee.executer.Execute attribute), 144
args (dicee.models.base_model.BaseKGE attribute), 18
args (dicee.models.base model.IdentityClass attribute), 21
args (dicee.models.BaseKGE attribute), 58, 61, 65, 69, 75, 88, 91
args (dicee.models.IdentityClass attribute), 61, 72, 78
args (dicee.models.pykeen_models.PykeenKGE attribute), 39
args (dicee.models.PykeenKGE attribute), 90
args (dicee.PykeenKGE attribute), 178
args (dicee.trainer.DICE_Trainer attribute), 111
args (dicee.trainer.dice_trainer.DICE_Trainer attribute), 107
ASWA (class in dicee.callbacks), 123
aswa (dicee.analyse\_experiments.Experiment attribute), 119
attn (dicee.models.transformers.Block attribute), 49
attn_dropout (dicee.models.transformers.CausalSelfAttention attribute), 48
attributes (dicee.abstracts.AbstractTrainer attribute), 113
В
backend (dicee.config.Namespace attribute), 127
backend (dicee.knowledge_graph.KG attribute), 146
BaseInteractiveKGE (class in dicee.abstracts), 114
BaseKGE (class in dicee), 180
BaseKGE (class in dicee.models), 57, 61, 64, 69, 75, 87, 91
BaseKGE (class in dicee.models.base_model), 18
BaseKGELightning (class in dicee.models), 52
BaseKGELightning (class in dicee.models.base_model), 12
batch_kronecker_product() (dicee.callbacks.KronE static method), 125
batch_size (dicee.analyse_experiments.Experiment attribute), 119
batch_size (dicee.callbacks.PseudoLabellingCallback attribute), 123
batch_size (dicee.config.Namespace attribute), 127
batch_size (dicee.CVDataModule attribute), 199
batch_size (dicee.dataset_classes.CVDataModule attribute), 138
bias (dicee.models.transformers.GPTConfig attribute), 50
bias (dicee.models.transformers.LayerNorm attribute), 47
```

```
Block (class in dicee.models.transformers), 49
block_size (dicee.BaseKGE attribute), 182
block_size (dicee.config.Namespace attribute), 129
block_size (dicee.dataset_classes.MultiClassClassificationDataset attribute), 131
block_size (dicee.models.base_model.BaseKGE attribute), 19
block_size (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
block_size (dicee.models.transformers.GPTConfig attribute), 49
block size (dicee.MultiClassClassificationDataset attribute), 193
bn_conv1 (dicee.AConvQ attribute), 171
bn_conv1 (dicee.ConvQ attribute), 171
bn_conv1 (dicee.models.AConvQ attribute), 74
bn_conv1 (dicee.models.ConvQ attribute), 74
bn_conv1 (dicee.models.quaternion.AConvQ attribute), 42
bn_conv1 (dicee.models.quaternion.ConvQ attribute), 41
bn_conv2 (dicee.AConvQ attribute), 171
bn_conv2 (dicee.ConvQ attribute), 171
bn_conv2 (dicee.models.AConvQ attribute), 75
bn_conv2 (dicee.models.ConvQ attribute), 74
bn_conv2 (dicee.models.quaternion.AConvQ attribute), 42
bn_conv2 (dicee.models.quaternion.ConvQ attribute), 41
bn_conv2d (dicee.AConEx attribute), 170
bn_conv2d (dicee.AConvO attribute), 170
bn_conv2d (dicee.ConEx attribute), 173
bn_conv2d (dicee.ConvO attribute), 172
bn_conv2d (dicee.models.AConEx attribute), 68
bn_conv2d (dicee.models.AConvO attribute), 80
bn_conv2d (dicee.models.complex.AConEx attribute), 29
bn_conv2d (dicee.models.complex.ConEx attribute), 28
bn_conv2d (dicee.models.ConEx attribute), 67
bn_conv2d (dicee.models.ConvO attribute), 80
bn_conv2d (dicee.models.octonion.AConvO attribute), 38
bn conv2d (dicee.models.octonion.ConvO attribute), 37
BPE_NegativeSamplingDataset (class in dicee), 191
{\tt BPE\_NegativeSamplingDataset} \ \textit{(class in dicee.dataset\_classes)}, \ 130
build_chain_funcs() (dicee.models.FMult2 method), 95
build_chain_funcs() (dicee.models.function_space.FMult2 method), 33
build func() (dicee.models.FMult2 method), 95
build_func() (dicee.models.function_space.FMult2 method), 33
BytE (class in dicee), 178
BytE (class in dicee.models.transformers), 45
byte_pair_encoding (dicee.analyse_experiments.Experiment attribute), 119
byte_pair_encoding (dicee.BaseKGE attribute), 182
byte_pair_encoding (dicee.config.Namespace attribute), 129
byte_pair_encoding (dicee.knowledge_graph.KG attribute), 146
byte_pair_encoding (dicee.models.base_model.BaseKGE attribute), 19
byte_pair_encoding (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
c_attn (dicee.models.transformers.CausalSelfAttention attribute), 47
c_fc (dicee.models.transformers.MLP attribute), 48
c_proj (dicee.models.transformers.CausalSelfAttention attribute), 48
c_proj (dicee.models.transformers.MLP attribute), 48
callbacks (dicee.abstracts.AbstractTrainer attribute), 113
callbacks (dicee.analyse_experiments.Experiment attribute), 119
{\tt callbacks}~(\textit{dicee.config.Namespace attribute}),~127
callbacks (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 110
CausalSelfAttention (class in dicee.models.transformers), 47
chain_func() (dicee.models.FMult method), 94
chain_func() (dicee.models.function_space.FMult method), 32
chain_func() (dicee.models.function_space.GFMult method), 33
chain_func() (dicee.models.GFMult method), 94
cl_pqr() (dicee.DeCaL method), 165
cl_pgr() (dicee.models.clifford.DeCaL method), 25
cl_pqr() (dicee.models.DeCaL method), 85
clifford_multiplication() (dicee.Keci method), 162
clifford_multiplication() (dicee.models.clifford.Keci method), 23
clifford_multiplication() (dicee.models.Keci method), 82
```

```
collate fn (dicee. Allvs All attribute), 195
collate_fn (dicee.dataset_classes.AllvsAll attribute), 133
collate fn (dicee.dataset classes.KvsAll attribute), 132
collate_fn (dicee.dataset_classes.KvsSampleDataset attribute), 136
collate_fn (dicee.dataset_classes.MultiClassClassificationDataset attribute), 131
collate_fn (dicee.dataset_classes.MultiLabelDataset attribute), 131
collate_fn (dicee.dataset_classes.OnevsAllDataset attribute), 132
collate fn (dicee.dataset classes.OnevsSample attribute), 134, 135
collate_fn (dicee.KvsAll attribute), 194
collate_fn (dicee.KvsSampleDataset attribute), 197
collate_fn (dicee.MultiClassClassificationDataset attribute), 193
collate_fn (dicee.MultiLabelDataset attribute), 192
collate_fn (dicee.OnevsAllDataset attribute), 193
collate_fn (dicee.OnevsSample attribute), 196
collate_fn() (dicee.BPE_NegativeSamplingDataset method), 192
collate_fn() (dicee.dataset_classes.BPE_NegativeSamplingDataset method), 130
collate_fn() (dicee.dataset_classes.TriplePredictionDataset method), 137
collate_fn() (dicee.TriplePredictionDataset method), 199
collection_name (dicee.scripts.serve.NeuralSearcher attribute), 106
comp_func() (dicee.LFMult method), 177
comp_func() (dicee.models.function_space.LFMult method), 35
comp_func() (dicee.models.LFMult method), 96
Complex (class in dicee), 168
Complex (class in dicee.models), 68
Complex (class in dicee.models.complex), 29
compute_convergence() (in module dicee.callbacks), 123
compute_func() (dicee.models.FMult method), 94
compute_func() (dicee.models.FMult2 method), 95
compute_func() (dicee.models.function_space.FMult method), 32
compute_func() (dicee.models.function_space.FMult2 method), 33
compute_func() (dicee.models.function_space.GFMult method), 33
compute func() (dicee.models.GFMult method), 94
compute_mrr() (dicee.callbacks.ASWA static method), 124
compute_sigma_pp() (dicee.DeCaL method), 166
compute_sigma_pp() (dicee.Keci method), 162
compute_sigma_pp() (dicee.models.clifford.DeCaL method), 26
compute_sigma_pp() (dicee.models.clifford.Keci method), 22
compute_sigma_pp() (dicee.models.DeCaL method), 86
compute_sigma_pp() (dicee.models.Keci method), 81
compute_sigma_pq() (dicee.DeCaL method), 167
compute_sigma_pq() (dicee.Keci method), 162
compute_sigma_pq() (dicee.models.clifford.DeCaL method), 27
compute_sigma_pq() (dicee.models.clifford.Keci method), 23
compute_sigma_pq() (dicee.models.DeCaL method), 87
compute_sigma_pq() (dicee.models.Keci method), 82
compute_sigma_pr() (dicee.DeCaL method), 167
compute_sigma_pr() (dicee.models.clifford.DeCaL method), 28
compute_sigma_pr() (dicee.models.DeCaL method), 87
compute_sigma_qq() (dicee.DeCaL method), 166
compute_sigma_qq() (dicee.Keci method), 162
\verb|compute_sigma_qq()| \textit{(dicee.models.clifford.DeCaL method)}, 27
compute_sigma_qq() (dicee.models.clifford.Keci method), 22
compute_sigma_qq() (dicee.models.DeCaL method), 86
compute_sigma_qq() (dicee.models.Keci method), 82
compute_sigma_qr() (dicee.DeCaL method), 167
compute_sigma_qr() (dicee.models.clifford.DeCaL method), 28
compute_sigma_qr() (dicee.models.DeCaL method), 87
compute_sigma_rr() (dicee.DeCaL method), 167
compute_sigma_rr() (dicee.models.clifford.DeCaL method), 27
compute_sigma_rr() (dicee.models.DeCaL method), 87
compute_sigmas_multivect() (dicee.DeCaL method), 165
compute_sigmas_multivect() (dicee.models.clifford.DeCaL method), 26
compute_sigmas_multivect() (dicee.models.DeCaL method), 85
compute_sigmas_single() (dicee.DeCaL method), 165
\verb|compute_sigmas_single()| \textit{(dicee.models.clifford.DeCaL method)}, 26
compute_sigmas_single() (dicee.models.DeCaL method), 85
ConEx (class in dicee), 173
ConEx (class in dicee.models), 67
```

```
ConEx (class in dicee.models.complex), 28
config (dicee.BytE attribute), 179
config (dicee.models.transformers.BytE attribute), 45
config (dicee.models.transformers.GPT attribute), 50
configs (dicee.abstracts.BaseInteractiveKGE attribute), 114
configure_optimizers() (dicee.models.base_model.BaseKGELightning method), 16
configure_optimizers() (dicee.models.BaseKGELightning method), 56
configure_optimizers() (dicee.models.transformers.GPT method), 51
construct_batch_selected_cl_multivector() (dicee.Keci method), 163
\verb|construct_batch_selected_cl_multivector()| \textit{(dicee.models.clifford.Keci method)}, 24
construct_batch_selected_cl_multivector() (dicee.models.Keci method), 83
construct_cl_multivector() (dicee.DeCaL method), 166
construct_cl_multivector() (dicee.Keci method), 163
construct_cl_multivector() (dicee.models.clifford.DeCaL method), 26
construct_cl_multivector() (dicee.models.clifford.Keci method), 23
construct_cl_multivector() (dicee.models.DeCaL method), 86
construct_cl_multivector() (dicee.models.Keci method), 83
construct_dataset() (in module dicee), 191
construct_dataset() (in module dicee.dataset_classes), 130
construct_ensemble (dicee.abstracts.BaseInteractiveKGE attribute), 114
construct_graph() (dicee.query_generator.QueryGenerator method), 152
construct_graph() (dicee.QueryGenerator method), 203
construct_input_and_output() (dicee.abstracts.BaseInteractiveKGE method), 116
construct_multi_coeff() (dicee.LFMult method), 176
construct_multi_coeff() (dicee.models.function_space.LFMult method), 34
construct_multi_coeff() (dicee.models.LFMult method), 96
continual_learning (dicee.config.Namespace attribute), 129
continual_start() (dicee.DICE_Trainer method), 185
continual_start() (dicee.executer.ContinuousExecute method), 145
continual_start() (dicee.trainer.DICE_Trainer method), 111
continual_start() (dicee.trainer.dice_trainer.DICE_Trainer method), 107
\verb|continual_training_setup_executor()| \textit{(in module dicee)}, 184
continual_training_setup_executor() (in module dicee.static_funcs), 155
ContinuousExecute (class in dicee.executer), 145
conv2d (dicee.AConEx attribute), 170
conv2d (dicee.AConvO attribute), 170
conv2d (dicee.AConvO attribute), 171
conv2d (dicee.ConEx attribute), 173
conv2d (dicee.ConvO attribute), 172
conv2d (dicee.ConvQ attribute), 171
conv2d (dicee.models.AConEx attribute), 67
conv2d (dicee.models.AConvO attribute), 80
conv2d (dicee.models.AConvQ attribute), 74
conv2d (dicee.models.complex.AConEx attribute), 29
conv2d (dicee.models.complex.ConEx attribute), 28
conv2d (dicee.models.ConEx attribute), 67
conv2d (dicee.models.ConvO attribute), 80
conv2d (dicee.models.ConvQ attribute), 74
conv2d (dicee.models.octonion.AConvO attribute), 38
conv2d (dicee.models.octonion.ConvO attribute), 37
conv2d (dicee.models.quaternion.AConvO attribute), 42
conv2d (dicee.models.quaternion.ConvQ attribute), 41
ConvO (class in dicee), 172
ConvO (class in dicee.models), 79
ConvO (class in dicee.models.octonion), 36
ConvO (class in dicee), 171
ConvQ (class in dicee.models), 74
ConvQ (class in dicee.models.quaternion), 41
create_constraints() (in module dicee.read_preprocess_save_load_kg.util), 103
create_constraints() (in module dicee.static_preprocess_funcs), 157
create experiment folder() (in module dicee), 184
create_experiment_folder() (in module dicee.static_funcs), 155
create_random_data() (dicee.callbacks.PseudoLabellingCallback method), 123
create_recipriocal_triples() (in module dicee), 183
create_recipriocal_triples() (in module dicee.read_preprocess_save_load_kg.util), 103
create_recipriocal_triples() (in module dicee.static_funcs), 154
create_vector_database() (dicee.KGE method), 186
create_vector_database() (dicee.knowledge_graph_embeddings.KGE method), 147
```

```
crop block size() (dicee.models.transformers.GPT method), 51
ctx (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 111
CVDataModule (class in dicee), 199
CVDataModule (class in dicee.dataset_classes), 137
D
data_module (dicee.callbacks.PseudoLabellingCallback attribute), 123
dataset_dir (dicee.config.Namespace attribute), 126
dataset_dir(dicee.knowledge_graph.KG attribute), 146
dataset_sanity_checking() (in module dicee.read_preprocess_save_load_kg.util), 103
DeCaL (class in dicee), 164
DeCal (class in dicee.models), 84
DeCal (class in dicee.models.clifford), 24
decide() (dicee.callbacks.ASWA method), 124
degree (dicee.LFMult attribute), 176
degree (dicee.models.function_space.LFMult attribute), 34
degree (dicee.models.LFMult attribute), 95
deploy() (dicee.KGE method), 189
deploy() (dicee.knowledge_graph_embeddings.KGE method), 151
deploy_head_entity_prediction() (in module dicee), 184
{\tt deploy\_head\_entity\_prediction()} \ \textit{(in module dicee.static\_funcs)}, 155
deploy_relation_prediction() (in module dicee), 184
deploy_relation_prediction() (in module dicee.static_funcs), 155
deploy_tail_entity_prediction() (in module dicee), 184
deploy_tail_entity_prediction() (in module dicee.static_funcs), 155
deploy_triple_prediction() (in module dicee), 184
deploy_triple_prediction() (in module dicee.static_funcs), 155
dept_index_triples_with_pandas() (in module dicee.read_preprocess_save_load_kg.util), 102
dept_read_preprocess_index_serialize_data() (dicee.Execute method), 190
dept_read_preprocess_index_serialize_data() (dicee.executer.Execute method), 144
describe() (dicee.knowledge_graph.KG method), 147
description_of_input (dicee.knowledge_graph.KG attribute), 147
DICE_Trainer (class in dicee), 184
DICE_Trainer (class in dicee.trainer), 111
DICE_Trainer (class in dicee.trainer.dice_trainer), 107
dicee
     module, 12
dicee.__main_
     module, 112
dicee.abstracts
     module, 112
dicee.analyse_experiments
     module, 118
dicee.callbacks
     module, 120
dicee.config
     module, 126
dicee.dataset_classes
     module, 129
dicee.eval_static_funcs
     module, 141
dicee.evaluator
    module, 142
dicee.executer
     module, 143
dicee.knowledge_graph
     module, 145
dicee.knowledge_graph_embeddings
     module, 147
dicee.models
     module, 12
dicee.models.base_model
     module, 12
dicee.models.clifford
     module, 21
dicee.models.complex
     module, 28
```

```
dicee.models.dualE
     module, 30
dicee.models.function_space
    module, 32
dicee.models.octonion
    module, 35
dicee.models.pykeen_models
    module, 38
dicee.models.quaternion
    module.39
dicee.models.real
    module, 42
dicee.models.static_funcs
    module, 44
dicee.models.transformers
    module, 44
dicee.query_generator
    module, 151
dicee.read_preprocess_save_load_kg
    module, 97
dicee.read_preprocess_save_load_kg.preprocess
    module, 97
dicee.read_preprocess_save_load_kg.read_from_disk
    module, 98
dicee.read_preprocess_save_load_kg.save_load_disk
    module, 99
dicee.read_preprocess_save_load_kg.util
    module, 99
dicee.sanity_checkers
    module, 152
dicee.scripts
    module, 105
dicee.scripts.index
    module, 105
dicee.scripts.run
    module, 105
dicee.scripts.serve
    module, 105
dicee.static_funcs
    module, 153
dicee.static_funcs_training
    module, 156
dicee.static_preprocess_funcs
    module, 156
dicee.trainer
    module, 106
dicee.trainer.dice_trainer
    module, 106
dicee.trainer.torch_trainer
    module, 108
dicee.trainer.torch_trainer_ddp
    module, 109
discrete_points (dicee.models.FMult2 attribute), 94
discrete_points (dicee.models.function_space.FMult2 attribute), 33
dist_func (dicee.models.Pyke attribute), 64
dist_func (dicee.models.real.Pyke attribute), 44
dist_func (dicee.Pyke attribute), 160
DistMult (class in dicee), 160
DistMult (class in dicee.models), 63
DistMult (class in dicee.models.real), 43
download_file() (in module dicee), 184
download_file() (in module dicee.static_funcs), 156
download_files_from_url() (in module dicee), 184
download_files_from_url() (in module dicee.static_funcs), 156
download_pretrained_model() (in module dicee), 184
download_pretrained_model() (in module dicee.static_funcs), 156
{\tt dropout}~(\textit{dicee.models.transformers.CausalSelfAttention~attribute}),\,48
dropout (dicee.models.transformers.GPTConfig attribute), 50
```

```
dropout (dicee.models.transformers.MLP attribute), 49
DualE (class in dicee), 168
DualE (class in dicee.models), 96
DualE (class in dicee.models.dualE), 31
dummy_eval() (dicee.evaluator.Evaluator method), 143
dummy_id (dicee.knowledge_graph.KG attribute), 146
during_training (dicee.evaluator.Evaluator attribute), 142
F
ee_vocab (dicee.evaluator.Evaluator attribute), 142
efficient_zero_grad() (in module dicee.static_funcs_training), 156
embedding_dim (dicee.analyse_experiments.Experiment attribute), 119
embedding_dim (dicee.BaseKGE attribute), 181
embedding_dim (dicee.config.Namespace attribute), 127
embedding_dim (dicee.models.base_model.BaseKGE attribute), 18
embedding_dim (dicee.models.BaseKGE attribute), 58, 61, 65, 70, 75, 88, 91
enable_log (in module dicee.static_preprocess_funcs), 157
enc (dicee.knowledge_graph.KG attribute), 146
end() (dicee.Execute method), 191
end() (dicee.executer.Execute method), 144
ent2id (dicee.query_generator.QueryGenerator attribute), 151
ent2id (dicee.QueryGenerator attribute), 203
\verb"ent_in" (\textit{dicee.query\_generator.QueryGenerator attribute}), 151
ent_in (dicee.QueryGenerator attribute), 203
ent_out (dicee.query_generator.QueryGenerator attribute), 152
ent_out (dicee.QueryGenerator attribute), 203
entities_str (dicee.knowledge_graph.KG property), 147
entity_embeddings (dicee.AConvQ attribute), 171
entity_embeddings (dicee.ConvQ attribute), 171
entity_embeddings (dicee.DeCaL attribute), 165
entity_embeddings (dicee.DualE attribute), 168
entity_embeddings (dicee.LFMult attribute), 176
entity_embeddings (dicee.models.AConvQ attribute), 74
entity_embeddings (dicee.models.clifford.DeCaL attribute), 25
entity_embeddings (dicee.models.ConvQ attribute), 74
entity_embeddings (dicee.models.DeCaL attribute), 85
entity_embeddings (dicee.models.DualE attribute), 97
entity_embeddings (dicee.models.dualE.DualE attribute), 31
entity_embeddings (dicee.models.FMult attribute), 93
entity_embeddings (dicee.models.FMult2 attribute), 95
entity_embeddings (dicee.models.function_space.FMult attribute), 32
entity_embeddings (dicee.models.function_space.FMult2 attribute), 33
entity_embeddings (dicee.models.function_space.GFMult attribute), 32
entity_embeddings (dicee.models.function_space.LFMult attribute), 34
entity_embeddings (dicee.models.function_space.LFMult1 attribute), 33
entity_embeddings (dicee.models.GFMult attribute), 94
entity_embeddings (dicee.models.LFMult attribute), 95
entity embeddings (dicee.models.LFMult1 attribute), 95
entity_embeddings (dicee.models.pykeen_models.PykeenKGE attribute), 39
entity_embeddings (dicee.models.PykeenKGE attribute), 90
entity_embeddings (dicee.models.quaternion.AConvQ attribute), 42
\verb"entity_embeddings" (\textit{dicee.models.quaternion.ConvQ attribute}), 41
entity_embeddings (dicee.PykeenKGE attribute), 178
entity_to_idx (dicee.knowledge_graph.KG attribute), 146
epoch count (dicee.abstracts.AbstractPPECallback attribute), 118
epoch_count (dicee.callbacks.ASWA attribute), 123
epoch_counter (dicee.callbacks.Eval attribute), 124
epoch_counter (dicee.callbacks.KGESaveCallback attribute), 122
epoch_ratio (dicee.callbacks.Eval attribute), 124
er_vocab (dicee.evaluator.Evaluator attribute), 142
estimate_mfu() (dicee.models.transformers.GPT method), 51
estimate_q() (in module dicee.callbacks), 123
Eval (class in dicee.callbacks), 124
\verb|eval()| (\textit{dicee.evaluator.Evaluator method}), 142
eval_lp_performance() (dicee.KGE method), 186
eval_lp_performance() (dicee.knowledge_graph_embeddings.KGE method), 147
eval_model (dicee.config.Namespace attribute), 128
```

```
eval_model (dicee.knowledge_graph.KG attribute), 146
eval_rank_of_head_and_tail_byte_pair_encoded_entity() (dicee.evaluator.Evaluator method), 143
eval_rank_of_head_and_tail_entity() (dicee.evaluator.Evaluator method), 143
eval_with_bpe_vs_all() (dicee.evaluator.Evaluator method), 143
eval_with_byte() (dicee.evaluator.Evaluator method), 143
eval_with_data() (dicee.evaluator.Evaluator method), 143
eval_with_vs_all() (dicee.evaluator.Evaluator method), 143
evaluate() (in module dicee), 184
evaluate() (in module dicee.static_funcs), 155
evaluate_bpe_lp() (in module dicee.static_funcs_training), 156
evaluate_link_prediction_performance() (in module dicee.eval_static_funcs), 141
evaluate_link_prediction_performance_with_bpe() (in module dicee.eval_static_funcs), 141
evaluate_link_prediction_performance_with_bpe_reciprocals() (in module dicee.eval_static_funcs), 141
evaluate_link_prediction_performance_with_reciprocals() (in module dicee.eval_static_funcs), 141
\verb| evaluate_lp()| \textit{ (dicee.evaluator.Evaluator method), } 143
evaluate_lp() (in module dicee.static_funcs_training), 156
evaluate_lp_bpe_k_vs_all() (dicee.evaluator.Evaluator method), 143
evaluate_lp_bpe_k_vs_all() (in module dicee.eval_static_funcs), 142
evaluate_lp_k_vs_all() (dicee.evaluator.Evaluator method), 143
evaluate_lp_with_byte() (dicee.evaluator.Evaluator method), 143
Evaluator (class in dicee.evaluator), 142
evaluator (dicee.DICE_Trainer attribute), 185
evaluator (dicee. Execute attribute), 190
evaluator (dicee.executer.Execute attribute), 144
evaluator (dicee.trainer.DICE_Trainer attribute), 111
evaluator (dicee.trainer.dice_trainer.DICE_Trainer attribute), 107
every_x_epoch (dicee.callbacks.KGESaveCallback attribute), 122
Execute (class in dicee), 190
Execute (class in dicee.executer), 143
exists() (dicee.knowledge_graph.KG method), 147
Experiment (class in dicee.analyse_experiments), 119
explicit (dicee.models.QMult attribute), 73
explicit (dicee.models.quaternion.QMult attribute), 40
explicit (dicee.QMult attribute), 174
exponential_function() (in module dicee), 184
exponential_function() (in module dicee.static_funcs), 155
extract input outputs() (dicee.trainer.torch trainer ddp.NodeTrainer method), 111
extract_input_outputs_set_device() (dicee.trainer.torch_trainer.TorchTrainer method), 109
f (dicee.callbacks.KronE attribute), 125
fc1 (dicee.AConEx attribute), 170
fc1 (dicee.AConvO attribute), 170
fc1 (dicee.AConvQ attribute), 171
fc1 (dicee.ConEx attribute), 173
fc1 (dicee.ConvO attribute), 172
fc1 (dicee.ConvQ attribute), 171
fc1 (dicee.models.AConEx attribute), 68
fc1 (dicee.models.AConvO attribute), 80
fc1 (dicee.models.AConvQ attribute), 74
fc1 (dicee.models.complex.AConEx attribute), 29
fc1 (dicee.models.complex.ConEx attribute), 28
fc1 (dicee.models.ConEx attribute), 67
fc1 (dicee.models.ConvO attribute), 80
fc1 (dicee.models.ConvO attribute), 74
fc1 (dicee.models.octonion.AConvO attribute), 38
fc1 (dicee.models.octonion.ConvO attribute), 37
fc1 (dicee.models.quaternion.AConvQ attribute), 42
fc1 (dicee.models.quaternion.ConvQ attribute), 41
fc_num_input (dicee.AConEx attribute), 170
fc_num_input (dicee.AConvO attribute), 170
fc_num_input (dicee.AConvQ attribute), 171
fc_num_input (dicee.ConEx attribute), 173
{\tt fc\_num\_input}~(\textit{dicee.ConvO attribute}),~172
fc_num_input (dicee.ConvQ attribute), 171
fc_num_input (dicee.models.AConEx attribute), 67
fc_num_input (dicee.models.AConvO attribute), 80
```

```
fc num input (dicee.models.AConvO attribute), 74
fc_num_input (dicee.models.complex.AConEx attribute), 29
fc_num_input (dicee.models.complex.ConEx attribute), 28
fc_num_input (dicee.models.ConEx attribute), 67
fc_num_input (dicee.models.ConvO attribute), 80
fc_num_input (dicee.models.ConvQ attribute), 74
fc_num_input (dicee.models.octonion.AConvO attribute), 38
fc num input (dicee.models.octonion.ConvO attribute), 37
fc_num_input (dicee.models.quaternion.AConvQ attribute), 42
fc_num_input (dicee.models.quaternion.ConvQ attribute), 41
feature_map_dropout (dicee.AConEx attribute), 170
feature_map_dropout (dicee.AConvO attribute), 170
feature_map_dropout (dicee.AConvQ attribute), 171
feature_map_dropout (dicee.ConEx attribute), 173
feature_map_dropout (dicee.ConvO attribute), 173
feature_map_dropout (dicee.ConvQ attribute), 171
feature_map_dropout (dicee.models.AConEx attribute), 68
feature_map_dropout (dicee.models.AConvO attribute), 80
feature_map_dropout (dicee.models.AConvQ attribute), 75
feature_map_dropout (dicee.models.complex.AConEx attribute), 29
feature_map_dropout (dicee.models.complex.ConEx attribute), 28
feature_map_dropout (dicee.models.ConEx attribute), 67
feature_map_dropout (dicee.models.ConvO attribute), 80
feature_map_dropout (dicee.models.ConvQ attribute), 74
feature_map_dropout (dicee.models.octonion.AConvO attribute), 38
feature_map_dropout (dicee.models.octonion.ConvO attribute), 37
feature_map_dropout (dicee.models.quaternion.AConvQ attribute), 42
feature_map_dropout (dicee.models.quaternion.ConvQ attribute), 41
feature_map_dropout_rate (dicee.BaseKGE attribute), 181
feature_map_dropout_rate (dicee.config.Namespace attribute), 129
feature_map_dropout_rate (dicee.models.base_model.BaseKGE attribute), 19
feature_map_dropout_rate (dicee.models.BaseKGE attribute), 58, 62, 65, 70, 76, 88, 92
fill_query() (dicee.query_generator.QueryGenerator method), 152
fill_query() (dicee.QueryGenerator method), 203
find_missing_triples() (dicee.KGE method), 189
find_missing_triples() (dicee.knowledge_graph_embeddings.KGE method), 150
fit () (dicee.trainer.torch_trainer_ddp.TorchDDPTrainer method), 110
fit () (dicee.trainer.torch_trainer.TorchTrainer method), 109
flash (dicee.models.transformers.CausalSelfAttention attribute), 48
FMult (class in dicee.models), 93
FMult (class in dicee.models.function_space), 32
FMult2 (class in dicee.models), 94
FMult2 (class in dicee.models.function_space), 33
form_of_labelling (dicee.DICE_Trainer attribute), 185
form_of_labelling (dicee.trainer.DICE_Trainer attribute), 111
form_of_labelling (dicee.trainer.dice_trainer.DICE_Trainer attribute), 107
forward() (dicee.BaseKGE method), 182
forward() (dicee.BytE method), 179
forward() (dicee.models.base_model.BaseKGE method), 20
forward() (dicee.models.base_model.IdentityClass static method), 21
forward() (dicee.models.BaseKGE method), 59, 63, 66, 71, 77, 89, 93
forward() (dicee.models.IdentityClass static method), 61, 72, 78
{\tt forward()} \ ({\it dicee.models.transformers.Block\ method}), 49
forward() (dicee.models.transformers.BytE method), 45
forward() (dicee.models.transformers.CausalSelfAttention method), 48
{\tt forward()} \ (\textit{dicee.models.transformers.GPT method}), 50
forward() (dicee.models.transformers.LayerNorm method), 47
forward() (dicee.models.transformers.MLP method), 49
forward_backward_update() (dicee.trainer.torch_trainer.TorchTrainer method), 109
forward_byte_pair_encoded_k_vs_all() (dicee.BaseKGE method), 182
forward_byte_pair_encoded_k_vs_all() (dicee.models.base_model.BaseKGE method), 19
forward_byte_pair_encoded_k_vs_all() (dicee.models.BaseKGE method), 59, 62, 66, 70, 76, 89, 92
forward_byte_pair_encoded_triple() (dicee.BaseKGE method), 182
forward_byte_pair_encoded_triple() (dicee.models.base_model.BaseKGE method), 19
forward_byte_pair_encoded_triple() (dicee.models.BaseKGE method), 59, 62, 66, 70, 76, 89, 92
forward_k_vs_all() (dicee.AConEx method), 170
forward_k_vs_all() (dicee.AConvO method), 170
forward_k_vs_all() (dicee.AConvQ method), 171
```

```
forward_k_vs_all() (dicee.BaseKGE method), 182
forward_k_vs_all() (dicee.ComplEx method), 169
forward_k_vs_all() (dicee.ConEx method), 173
forward_k_vs_all() (dicee.ConvO method), 173
forward_k_vs_all() (dicee.ConvQ method), 172
forward_k_vs_all() (dicee.DeCaL method), 166
forward_k_vs_all() (dicee.DistMult method), 161
forward k vs all() (dicee.DualE method), 168
forward_k_vs_all() (dicee.Keci method), 163
forward_k_vs_all() (dicee.models.AConEx method), 68
forward_k_vs_all() (dicee.models.AConvO method), 81
{\tt forward\_k\_vs\_all()} \ (\textit{dicee.models.AConvQ method}), 75
forward_k_vs_all() (dicee.models.base_model.BaseKGE method), 20
forward_k_vs_all() (dicee.models.BaseKGE method), 60, 63, 66, 71, 77, 90, 93
forward_k_vs_all() (dicee.models.clifford.DeCaL method), 26
forward_k_vs_all() (dicee.models.clifford.Keci method), 23
forward_k_vs_all() (dicee.models.ComplEx method), 69
forward_k_vs_all() (dicee.models.complex.AConEx method), 29
forward_k_vs_all() (dicee.models.complex.ComplEx method), 30
forward_k_vs_all() (dicee.models.complex.ConEx method), 29
forward_k_vs_all() (dicee.models.ConEx method), 67
forward_k_vs_all() (dicee.models.ConvO method), 80
forward_k_vs_all() (dicee.models.ConvQ method), 74
forward_k_vs_all() (dicee.models.DeCaL method), 85
forward_k_vs_all() (dicee.models.DistMult method), 63
forward_k_vs_all() (dicee.models.DualE method), 97
forward_k_vs_all() (dicee.models.dualE.DualE method), 31
forward_k_vs_all() (dicee.models.Keci method), 83
{\tt forward\_k\_vs\_all()} \ (\textit{dicee.models.octonion.AConvO method}), 38
forward_k_vs_all() (dicee.models.octonion.ConvO method), 37
forward_k_vs_all() (dicee.models.octonion.OMult method), 36
forward_k_vs_all() (dicee.models.OMult method), 79
forward_k_vs_all() (dicee.models.pykeen_models.PykeenKGE method), 39
forward_k_vs_all() (dicee.models.PykeenKGE method), 90
forward_k_vs_all() (dicee.models.QMult method), 73
{\tt forward\_k\_vs\_all()} \ (\textit{dicee.models.quaternion.AConvQ method}), 42
forward k vs all() (dicee.models.quaternion.ConvO method), 42
forward_k_vs_all() (dicee.models.quaternion.QMult method), 41
forward_k_vs_all() (dicee.models.real.DistMult method), 43
forward_k_vs_all() (dicee.models.real.Shallom method), 43
forward_k_vs_all() (dicee.models.real.TransE method), 43
forward_k_vs_all() (dicee.models.Shallom method), 64
forward_k_vs_all() (dicee.models.TransE method), 64
forward_k_vs_all() (dicee.OMult method), 176
forward_k_vs_all() (dicee.PykeenKGE method), 178
forward_k_vs_all() (dicee.QMult method), 175
forward_k_vs_all() (dicee.Shallom method), 176
forward_k_vs_all() (dicee. TransE method), 164
forward_k_vs_sample() (dicee.AConEx method), 170
forward_k_vs_sample() (dicee.BaseKGE method), 182
forward_k_vs_sample() (dicee.ComplEx method), 169
forward_k_vs_sample() (dicee.ConEx method), 173
forward_k_vs_sample() (dicee.DistMult method), 161
forward_k_vs_sample() (dicee.Keci method), 163
forward_k_vs_sample() (dicee.models.AConEx method), 68
forward_k_vs_sample() (dicee.models.base_model.BaseKGE method), 20
forward_k_vs_sample() (dicee.models.BaseKGE method), 60, 63, 66, 71, 77, 90, 93
forward_k_vs_sample() (dicee.models.clifford.Keci method), 24
forward_k_vs_sample() (dicee.models.ComplEx method), 69
forward_k_vs_sample() (dicee.models.complex.AConEx method), 29
forward_k_vs_sample() (dicee.models.complex.ComplEx method), 30
forward_k_vs_sample() (dicee.models.complex.ConEx method), 29
forward_k_vs_sample() (dicee.models.ConEx method), 67
forward_k_vs_sample() (dicee.models.DistMult method), 63
forward_k_vs_sample() (dicee.models.Keci method), 83
forward_k_vs_sample() (dicee.models.pykeen_models.PykeenKGE method), 39
forward_k_vs_sample() (dicee.models.PykeenKGE method), 91
forward_k_vs_sample() (dicee.models.QMult method), 74
```

```
forward k vs sample() (dicee.models.quaternion.OMult method), 41
forward_k_vs_sample() (dicee.models.real.DistMult method), 43
forward_k_vs_sample() (dicee.PykeenKGE method), 178
forward_k_vs_sample() (dicee.QMult method), 175
forward_k_vs_with_explicit() (dicee.Keci method), 163
forward_k_vs_with_explicit() (dicee.models.clifford.Keci method), 23
forward_k_vs_with_explicit() (dicee.models.Keci method), 83
forward triples() (dicee.AConEx method), 170
forward_triples() (dicee.AConvO method), 170
forward_triples() (dicee.AConvQ method), 171
forward_triples() (dicee.BaseKGE method), 182
forward_triples() (dicee.ConEx method), 173
forward_triples() (dicee.ConvO method), 173
forward_triples() (dicee.ConvQ method), 172
forward_triples() (dicee.DeCaL method), 165
forward_triples() (dicee.DualE method), 168
forward_triples() (dicee.Keci method), 164
forward_triples() (dicee.LFMult method), 176
forward_triples() (dicee.models.AConEx method), 68
forward_triples() (dicee.models.AConvO method), 81
forward_triples() (dicee.models.AConvQ method), 75
forward triples() (dicee.models.base model.BaseKGE method), 20
forward_triples() (dicee.models.BaseKGE method), 59, 63, 66, 71, 77, 89, 93
forward_triples() (dicee.models.clifford.DeCaL method), 25
forward_triples() (dicee.models.clifford.Keci method), 24
forward_triples() (dicee.models.complex.AConEx method), 29
forward_triples() (dicee.models.complex.ConEx method), 29
forward_triples() (dicee.models.ConEx method), 67
forward_triples() (dicee.models.ConvO method), 80
forward_triples() (dicee.models.ConvQ method), 74
forward_triples() (dicee.models.DeCaL method), 85
forward triples() (dicee.models.DualE method), 97
forward_triples() (dicee.models.dualE.DualE method), 31
forward_triples() (dicee.models.FMult method), 94
forward_triples() (dicee.models.FMult2 method), 95
forward_triples() (dicee.models.function_space.FMult method), 32
forward triples() (dicee.models.function space.FMult2 method), 33
forward_triples() (dicee.models.function_space.GFMult method), 33
forward_triples() (dicee.models.function_space.LFMult method), 34
forward_triples() (dicee.models.function_space.LFMult1 method), 34
forward_triples() (dicee.models.GFMult method), 94
forward_triples() (dicee.models.Keci method), 84
forward_triples() (dicee.models.LFMult method), 95
forward_triples() (dicee.models.LFMult1 method), 95
forward_triples() (dicee.models.octonion.AConvO method), 38
forward_triples() (dicee.models.octonion.ConvO method), 37
forward_triples() (dicee.models.Pyke method), 64
forward_triples() (dicee.models.pykeen_models.PykeenKGE method), 39
forward_triples() (dicee.models.PykeenKGE method), 91
forward_triples() (dicee.models.quaternion.AConvQ method), 42
forward_triples() (dicee.models.quaternion.ConvQ method), 42
forward_triples() (dicee.models.real.Pyke method), 44
forward_triples() (dicee.models.real.Shallom method), 43
forward_triples() (dicee.models.Shallom method), 64
forward_triples() (dicee.Pyke method), 160
forward_triples() (dicee.PykeenKGE method), 178
forward_triples() (dicee.Shallom method), 176
frequency (dicee.callbacks.Perturb attribute), 126
from_pretrained() (dicee.models.transformers.GPT class method), 51
full_storage_path (dicee.analyse_experiments.Experiment attribute), 119
func_triple_to_bpe_representation (dicee.evaluator.Evaluator attribute), 142
func_triple_to_bpe_representation() (dicee.knowledge_graph.KG method), 147
function() (dicee.models.FMult2 method), 95
function() (dicee.models.function_space.FMult2 method), 33
```

gamma (dicee.models.FMult attribute), 94

```
gamma (dicee.models.function space.FMult attribute), 32
gelu (dicee.models.transformers.MLP attribute), 48
gen_test (dicee.query_generator.QueryGenerator attribute), 151
gen_test (dicee.QueryGenerator attribute), 203
gen_valid (dicee.query_generator.QueryGenerator attribute), 151
gen_valid (dicee.QueryGenerator attribute), 203
generate() (dicee.BytE method), 179
generate() (dicee.KGE method), 186
generate() (dicee.knowledge_graph_embeddings.KGE method), 147
generate() (dicee.models.transformers.BytE method), 46
generate_queries() (dicee.query_generator.QueryGenerator method), 152
generate_queries() (dicee.QueryGenerator method), 203
get () (dicee.scripts.serve.NeuralSearcher method), 106
get_aswa_state_dict() (dicee.callbacks.ASWA method), 124
{\tt get\_bpe\_head\_and\_relation\_representation()} \ (\textit{dicee.BaseKGE method}), \, 183
get_bpe_head_and_relation_representation() (dicee.models.base_model.BaseKGE method), 20
get_bpe_head_and_relation_representation() (dicee.models.BaseKGE method), 60, 63, 67, 71, 77, 90, 93
get_bpe_token_representation() (dicee.abstracts.BaseInteractiveKGE method), 114
get_callbacks() (in module dicee.trainer.dice_trainer), 107
get_default_arguments() (in module dicee.analyse_experiments), 119
get_default_arguments() (in module dicee.scripts.index), 105
get_default_arguments() (in module dicee.scripts.run), 105
get_default_arguments() (in module dicee.scripts.serve), 106
get_ee_vocab() (in module dicee), 183
get_ee_vocab() (in module dicee.read_preprocess_save_load_kg.util), 102
get_ee_vocab() (in module dicee.static_funcs), 154
get_ee_vocab() (in module dicee.static_preprocess_funcs), 157
get_embeddings() (dicee.BaseKGE method), 183
get_embeddings() (dicee.models.base_model.BaseKGE method), 20
get_embeddings() (dicee.models.BaseKGE method), 60, 63, 67, 71, 77, 90, 93
get_embeddings() (dicee.models.real.Shallom method), 43
get_embeddings() (dicee.models.Shallom method), 64
get_embeddings() (dicee.Shallom method), 176
\verb|get_entity_embeddings()| \textit{(dicee.abstracts.BaseInteractiveKGE method)}, 116
get_entity_index() (dicee.abstracts.BaseInteractiveKGE method), 115
get_er_vocab() (in module dicee), 183
get_er_vocab() (in module dicee.read_preprocess_save_load_kg.util), 102
get_er_vocab() (in module dicee.static_funcs), 154
get_er_vocab() (in module dicee.static_preprocess_funcs), 157
get_eval_report() (dicee.abstracts.BaseInteractiveKGE method), 114
get_head_relation_representation() (dicee.BaseKGE method), 182
get_head_relation_representation() (dicee.models.base_model.BaseKGE method), 20
get_head_relation_representation() (dicee.models.BaseKGE method), 60, 63, 66, 71, 77, 90, 93
get_kronecker_triple_representation() (dicee.callbacks.KronE method), 125
get_num_params() (dicee.models.transformers.GPT method), 50
get_padded_bpe_triple_representation() (dicee.abstracts.BaseInteractiveKGE method), 115
get_queries() (dicee.query_generator.QueryGenerator method), 152
get_queries() (dicee.QueryGenerator method), 203
get_re_vocab() (in module dicee), 183
get_re_vocab() (in module dicee.read_preprocess_save_load_kg.util), 102
get_re_vocab() (in module dicee.static_funcs), 154
get_re_vocab() (in module dicee.static_preprocess_funcs), 157
\verb|get_relation_embeddings()| \textit{(dicee.abstracts.BaseInteractiveKGE method)}, 116
get_relation_index() (dicee.abstracts.BaseInteractiveKGE method), 115
get_sentence_representation() (dicee.BaseKGE method), 182
{\tt get\_sentence\_representation()} \ (\textit{dicee.models.base\_model.BaseKGE method}), 20
get_sentence_representation() (dicee.models.BaseKGE method), 60, 63, 67, 71, 77, 90, 93
get_transductive_entity_embeddings() (dicee.KGE method), 186
get_transductive_entity_embeddings() (dicee.knowledge_graph_embeddings.KGE method), 147
get_triple_representation() (dicee.BaseKGE method), 182
get_triple_representation() (dicee.models.base_model.BaseKGE method), 20
get_triple_representation() (dicee.models.BaseKGE method), 60, 63, 66, 71, 77, 90, 93
GFMult (class in dicee.models), 94
GFMult (class in dicee.models.function_space), 32
global_rank (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 110
GPT (class in dicee.models.transformers), 50
GPTConfig (class in dicee.models.transformers), 49
gpus (dicee.config.Namespace attribute), 127
```

```
gradient_accumulation_steps (dicee.config.Namespace attribute), 128
ground_queries() (dicee.query_generator.QueryGenerator method), 152
ground_queries() (dicee.QueryGenerator method), 203
Н
hidden_dropout (dicee.BaseKGE attribute), 182
hidden_dropout (dicee.models.base_model.BaseKGE attribute), 19
hidden_dropout (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
hidden_dropout_rate (dicee.BaseKGE attribute), 181
hidden_dropout_rate (dicee.config.Namespace attribute), 129
hidden_dropout_rate (dicee.models.base_model.BaseKGE attribute), 19
hidden_dropout_rate (dicee.models.BaseKGE attribute), 58, 62, 65, 70, 76, 88, 92
hidden_normalizer (dicee.BaseKGE attribute), 182
hidden_normalizer (dicee.models.base_model.BaseKGE attribute), 19
hidden_normalizer (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
IdentityClass (class in dicee.models), 60, 71, 77
IdentityClass (class in dicee.models.base_model), 20
idx_entity_to_bpe_shaped (dicee.knowledge_graph.KG attribute), 146
index_triple() (dicee.abstracts.BaseInteractiveKGE method), 115
init_dataloader() (dicee.DICE_Trainer method), 185
init_dataloader() (dicee.trainer.DICE_Trainer method), 112
init_dataloader() (dicee.trainer.dice_trainer.DICE_Trainer method), 108
init_dataset() (dicee.DICE_Trainer method), 185
init_dataset() (dicee.trainer.DICE_Trainer method), 112
init_dataset() (dicee.trainer.dice_trainer.DICE_Trainer method), 108
init_param (dicee.config.Namespace attribute), 128
init_params_with_sanity_checking() (dicee.BaseKGE method), 182
\verb|init_params_with_sanity_checking()| \textit{(dicee.models.base\_model.BaseKGE method)}, 19
init_params_with_sanity_checking() (dicee.models.BaseKGE method), 59, 62, 66, 71, 77, 89, 93
initial_eval_setting (dicee.callbacks.ASWA attribute), 123
initialize_or_load_model() (dicee.DICE_Trainer method), 185
initialize_or_load_model() (dicee.trainer.DICE_Trainer method), 112
initialize or load model() (dicee.trainer.dice trainer.DICE Trainer method), 108
initialize_trainer() (dicee.DICE_Trainer method), 185
initialize_trainer() (dicee.trainer.DICE_Trainer method), 112
initialize_trainer() (dicee.trainer.dice_trainer.DICE_Trainer method), 108
initialize_trainer() (in module dicee.trainer.dice_trainer), 107
input_dp_ent_real (dicee.BaseKGE attribute), 182
input_dp_ent_real (dicee.models.base_model.BaseKGE attribute), 19
input_dp_ent_real (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
input_dp_rel_real (dicee.BaseKGE attribute), 182
\verb"input_dp_rel_real" (\textit{dicee.models.base\_model.BaseKGE attribute}), 19
input_dp_rel_real (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
input_dropout_rate (dicee.BaseKGE attribute), 181
input_dropout_rate (dicee.config.Namespace attribute), 128
input_dropout_rate (dicee.models.base_model.BaseKGE attribute), 19
input_dropout_rate (dicee.models.BaseKGE attribute), 58, 62, 65, 70, 76, 88, 92
intialize_model() (in module dicee), 184
intialize_model() (in module dicee.static_funcs), 155
is_continual_training (dicee.DICE_Trainer attribute), 185
is_continual_training (dicee.evaluator.Evaluator attribute), 142
\verb|is_continual_training| (\textit{dicee.Execute attribute}), 190
is_continual_training (dicee.executer.Execute attribute), 144
is_continual_training (dicee.trainer.DICE_Trainer attribute), 111
is_continual_training (dicee.trainer.dice_trainer.DICE_Trainer attribute), 107
is_global_zero (dicee.abstracts.AbstractTrainer attribute), 113
is seen () (dicee.abstracts.BaseInteractiveKGE method), 115
is_sparql_endpoint_alive() (in module dicee.sanity_checkers), 153
K
k (dicee.models.FMult attribute), 93
k (dicee.models.FMult2 attribute), 94
k (dicee.models.function_space.FMult attribute), 32
k (dicee.models.function_space.FMult2 attribute), 33
```

```
k (dicee.models.function space.GFMult attribute), 32
k (dicee.models.GFMult attribute), 94
k_fold_cross_validation() (dicee.DICE_Trainer method), 185
k_fold_cross_validation() (dicee.trainer.DICE_Trainer method), 112
k_fold_cross_validation() (dicee.trainer.dice_trainer.DICE_Trainer method), 108
k_vs_all_score() (dicee.ComplEx static method), 169
k_vs_all_score() (dicee.DistMult method), 160
k vs all score() (dicee.Keci method), 163
k_vs_all_score() (dicee.models.clifford.Keci method), 23
k_vs_all_score() (dicee.models.ComplEx static method), 69
k_vs_all_score() (dicee.models.complex.ComplEx static method), 30
k_vs_all_score() (dicee.models.DistMult method), 63
k_vs_all_score() (dicee.models.Keci method), 83
k\_vs\_all\_score() (dicee.models.octonion.OMult method), 36
k_vs_all_score() (dicee.models.OMult method), 79
k_vs_all_score() (dicee.models.QMult method), 73
k_vs_all_score() (dicee.models.quaternion.QMult method), 41
k_vs_all_score() (dicee.models.real.DistMult method), 43
k_vs_all_score() (dicee.OMult method), 176
k_vs_all_score() (dicee.QMult method), 175
Keci (class in dicee), 161
Keci (class in dicee.models), 81
Keci (class in dicee.models.clifford), 21
KeciBase (class in dicee), 161
KeciBase (class in dicee.models), 84
KeciBase (class in dicee.models.clifford), 24
kernel_size (dicee.BaseKGE attribute), 181
kernel_size (dicee.config.Namespace attribute), 128
kernel_size (dicee.models.base_model.BaseKGE attribute), 19
kernel_size (dicee.models.BaseKGE attribute), 58, 62, 65, 70, 76, 88, 92
KG (class in dicee.knowledge_graph), 146
kg (dicee.callbacks.PseudoLabellingCallback attribute), 123
kg (dicee.read_preprocess_save_load_kg.LoadSaveToDisk attribute), 104
kg (dicee.read_preprocess_save_load_kg.PreprocessKG attribute), 103
kg (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG attribute), 98
kg (dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk attribute), 98
kg (dicee.read preprocess save load kg.ReadFromDisk attribute), 104
kg (dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk attribute), 99
KGE (class in dicee), 186
KGE (class in dicee.knowledge_graph_embeddings), 147
KGESaveCallback (class in dicee.callbacks), 121
knowledge_graph (dicee.Execute attribute), 190
knowledge_graph (dicee.executer.Execute attribute), 144
KronE (class in dicee.callbacks), 125
KvsAll (class in dicee), 193
KvsAll (class in dicee.dataset_classes), 132
kvsall_score() (dicee.DualE method), 168
kvsall_score() (dicee.models.DualE method), 97
kvsall_score() (dicee.models.dualE.DualE method), 31
KvsSampleDataset (class in dicee), 197
KvsSampleDataset (class in dicee.dataset_classes), 135
label_smoothing_rate (dicee.AllvsAll attribute), 195
label_smoothing_rate (dicee.config.Namespace attribute), 128
label_smoothing_rate (dicee.dataset_classes.AllvsAll attribute), 133
label_smoothing_rate (dicee.dataset_classes.KvsAll attribute), 132
label_smoothing_rate (dicee.dataset_classes.KvsSampleDataset attribute), 136
label_smoothing_rate (dicee.dataset_classes.OnevsSample attribute), 134, 135
label_smoothing_rate (dicee.dataset_classes.TriplePredictionDataset attribute), 137
label_smoothing_rate (dicee.KvsAll attribute), 194
label_smoothing_rate (dicee.KvsSampleDataset attribute), 197
label_smoothing_rate (dicee.OnevsSample attribute), 196
label_smoothing_rate (dicee. TriplePredictionDataset attribute), 199
LayerNorm (class in dicee.models.transformers), 47
learning_rate (dicee.BaseKGE attribute), 181
learning_rate (dicee.models.base_model.BaseKGE attribute), 19
```

```
learning rate (dicee.models.BaseKGE attribute), 58, 62, 65, 70, 76, 88, 92
length (dicee.dataset_classes.NegSampleDataset attribute), 136
length (dicee.dataset_classes.TriplePredictionDataset attribute), 137
length (dicee.NegSampleDataset attribute), 198
length (dicee. TriplePredictionDataset attribute), 199
level (dicee.callbacks.Perturb attribute), 126
LFMult (class in dicee), 176
LFMult (class in dicee.models), 95
LFMult (class in dicee.models.function_space), 34
LFMult1 (class in dicee.models), 95
LFMult1 (class in dicee.models.function_space), 33
linear() (dicee.LFMult method), 177
linear() (dicee.models.function_space.LFMult method), 34
linear() (dicee.models.LFMult method), 96
list2tuple() (dicee.query_generator.QueryGenerator method), 152
list2tuple() (dicee.QueryGenerator method), 203
lm_head (dicee.BytE attribute), 179
lm_head (dicee.models.transformers.BytE attribute), 45
lm_head (dicee.models.transformers.GPT attribute), 50
ln_1 (dicee.models.transformers.Block attribute), 49
ln_2 (dicee.models.transformers.Block attribute), 49
load() (dicee.read_preprocess_save_load_kg.LoadSaveToDisk method), 104
load() (dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk method), 99
load_json() (in module dicee), 184
load_json() (in module dicee.static_funcs), 155
load_model() (in module dicee), 183
load_model() (in module dicee.static_funcs), 154
load_model_ensemble() (in module dicee), 183
load_model_ensemble() (in module dicee.static_funcs), 154
load_numpy() (in module dicee), 184
load_numpy() (in module dicee.static_funcs), 155
load_numpy_ndarray() (in module dicee.read_preprocess_save_load_kg.util), 103
load_pickle() (in module dicee), 183
load_pickle() (in module dicee.read_preprocess_save_load_kg.util), 103
load_pickle() (in module dicee.static_funcs), 154
load_queries() (dicee.query_generator.QueryGenerator method), 152
load gueries () (dicee. Ouery Generator method), 203
load_queries_and_answers() (dicee.query_generator.QueryGenerator static method), 152
load_queries_and_answers() (dicee.QueryGenerator static method), 204
load_term_mapping() (in module dicee), 183, 191
load_term_mapping() (in module dicee.static_funcs), 154
load_term_mapping() (in module dicee.trainer.dice_trainer), 107
load_with_pandas() (in module dicee.read_preprocess_save_load_kg.util), 103
LoadSaveToDisk (class in dicee.read_preprocess_save_load_kg), 104
LoadSaveToDisk (class in dicee.read_preprocess_save_load_kg.save_load_disk), 99
local_rank (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 110
loss (dicee.BaseKGE attribute), 181
loss (dicee.models.base_model.BaseKGE attribute), 19
loss (dicee.models.BaseKGE attribute), 59, 62, 65, 70, 76, 89, 92
loss_func (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 110
{\tt loss\_function}~(\textit{dicee.trainer.torch\_trainer.TorchTrainer}~attribute),~109
loss_function() (dicee.BytE method), 179
{\tt loss\_function()} \ (\textit{dicee.models.base\_model.BaseKGELightning method}), 14
loss_function() (dicee.models.BaseKGELightning method), 54
loss_function() (dicee.models.transformers.BytE method), 45
loss_history (dicee.BaseKGE attribute), 182
loss_history (dicee.models.base_model.BaseKGE attribute), 19
loss_history (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
loss_history (dicee.models.pykeen_models.PykeenKGE attribute), 38
loss_history (dicee.models.PykeenKGE attribute), 90
loss_history (dicee.PykeenKGE attribute), 178
loss_history (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 111
1r (dicee.analyse_experiments.Experiment attribute), 119
1r (dicee.config.Namespace attribute), 127
```

M

m (dicee.LFMult attribute), 176

```
m (dicee.models.function space.LFMult attribute), 34
m (dicee.models.LFMult attribute), 95
main() (in module dicee.scripts.index), 105
main() (in module dicee.scripts.run), 105
main() (in module dicee.scripts.serve), 106
make_iterable_verbose() (in module dicee.trainer.torch_trainer_ddp), 110
mapping_from_first_two_cols_to_third() (in module dicee), 191
mapping_from_first_two_cols_to_third() (in module dicee.static_preprocess_funcs), 157
margin (dicee.models.Pyke attribute), 64
margin (dicee.models.real.Pyke attribute), 44
margin (dicee.models.real.TransE attribute), 43
margin (dicee.models.TransE attribute), 64
margin (dicee. Pyke attribute), 160
margin (dicee. TransE attribute), 164
max_ans_num (dicee.query_generator.QueryGenerator attribute), 151
max_ans_num (dicee.QueryGenerator attribute), 203
max_epochs (dicee.callbacks.KGESaveCallback attribute), 122
max_length_subword_tokens (dicee.BaseKGE attribute), 182
max_length_subword_tokens (dicee.knowledge_graph.KG attribute), 146
max_length_subword_tokens (dicee.models.base_model.BaseKGE attribute), 19
max_length_subword_tokens (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
max_num_of_classes (dicee.dataset_classes.KvsSampleDataset attribute), 136
max_num_of_classes (dicee.KvsSampleDataset attribute), 197
mem_of_model() (dicee.models.base_model.BaseKGELightning method), 13
mem_of_model() (dicee.models.BaseKGELightning method), 53
method (dicee.callbacks.Perturb attribute), 126
MLP (class in dicee.models.transformers), 48
mlp (dicee.models.transformers.Block attribute), 49
mode (dicee.query_generator.QueryGenerator attribute), 151
mode (dicee.QueryGenerator attribute), 203
model (dicee.config.Namespace attribute), 127
model (dicee.models.pykeen_models.PykeenKGE attribute), 38
model (dicee.models.PykeenKGE attribute), 90
model (dicee. Pykeen KGE attribute), 178
model (dicee.scripts.serve.NeuralSearcher attribute), 106
model (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 110
model (dicee.trainer.torch trainer.TorchTrainer attribute), 109
{\tt model\_kwargs} (dicee.models.pykeen_models.PykeenKGE attribute), 38
model_kwargs (dicee.models.PykeenKGE attribute), 90
model_kwargs (dicee.PykeenKGE attribute), 177
model_name (dicee.analyse_experiments.Experiment attribute), 119
module
     dicee, 12
     dicee.___main___, 112
     dicee.abstracts, 112
     dicee.analyse_experiments, 118
     dicee.callbacks, 120
     dicee.config, 126
     dicee.dataset_classes, 129
     dicee.eval_static_funcs, 141
     dicee.evaluator, 142
     dicee.executer, 143
     {\tt dicee.knowledge\_graph, 145}
     dicee.knowledge_graph_embeddings, 147
     dicee.models, 12
     dicee.models.base_model, 12
     dicee.models.clifford, 21
     dicee.models.complex, 28
     dicee.models.dualE, 30
     dicee.models.function_space, 32
     dicee.models.octonion.35
     dicee.models.pykeen_models, 38
     dicee.models.quaternion, 39
     dicee.models.real, 42
     dicee.models.static_funcs, 44
     dicee.models.transformers, 44
     dicee.query_generator, 151
     dicee.read_preprocess_save_load_kg,97
```

```
dicee.read_preprocess_save_load_kg.preprocess, 97
      dicee.read_preprocess_save_load_kg.read_from_disk,98
      dicee.read_preprocess_save_load_kg.save_load_disk,99
      dicee.read_preprocess_save_load_kg.util,99
     {\tt dicee.sanity\_checkers, 152}
      dicee.scripts, 105
      dicee.scripts.index, 105
      dicee.scripts.run, 105
      dicee.scripts.serve, 105
      dicee.static_funcs, 153
      dicee.static_funcs_training, 156
      dicee.static_preprocess_funcs, 156
      dicee.trainer, 106
      dicee.trainer.dice_trainer, 106
      dicee.trainer.torch_trainer, 108
      dicee.trainer.torch_trainer_ddp, 109
MultiClassClassificationDataset (class in dicee), 192
MultiClassClassificationDataset (class in dicee.dataset_classes), 131
MultiLabelDataset (class in dicee), 192
MultiLabelDataset (class in dicee.dataset_classes), 130
Ν
n (dicee.models.FMult2 attribute), 94
n (dicee.models.function_space.FMult2 attribute), 33
n_embd (dicee.models.transformers.CausalSelfAttention attribute), 48
n_embd (dicee.models.transformers.GPTConfig attribute), 50
n_head (dicee.models.transformers.CausalSelfAttention attribute), 48
n_head (dicee.models.transformers.GPTConfig attribute), 50
n_layer (dicee.models.transformers.GPTConfig attribute), 49
n_layers (dicee.models.FMult2 attribute), 94
n_layers (dicee.models.function_space.FMult2 attribute), 33
name (dicee.abstracts.BaseInteractiveKGE property), 115
name (dicee.AConEx attribute), 169
name (dicee.AConvO attribute), 170
name (dicee.AConvQ attribute), 171
name (dicee.BytE attribute), 179
name (dicee.ComplEx attribute), 169
name (dicee.ConEx attribute), 173
name (dicee.ConvO attribute), 172
name (dicee.ConvQ attribute), 171
name (dicee.DeCaL attribute), 165
name (dicee.DistMult attribute), 160
name (dicee.DualE attribute), 168
name (dicee. Keci attribute), 161
name (dicee.KeciBase attribute), 161
name (dicee.LFMult attribute), 176
name (dicee.models.AConEx attribute), 67
name (dicee.models.AConvO attribute), 80
name (dicee.models.AConvQ attribute), 74
name (dicee.models.clifford.DeCaL attribute), 25
name (dicee.models.clifford.Keci attribute), 22
name (dicee.models.clifford.KeciBase attribute), 24
name (dicee.models.ComplEx attribute), 69
\verb"name" (\textit{dicee.models.complex.AConEx attribute}), 29
name (dicee.models.complex.ComplEx attribute), 30
name (dicee.models.complex.ConEx attribute), 28
name (dicee.models.ConEx attribute), 67
name (dicee.models.ConvO attribute), 80
name (dicee.models.ConvQ attribute), 74
name (dicee.models.DeCaL attribute), 85
name (dicee.models.DistMult attribute), 63
name (dicee.models.DualE attribute), 96
name (dicee.models.dualE.DualE attribute), 31
name (dicee.models.FMult attribute), 93
name (dicee.models.FMult2 attribute), 94
name (dicee.models.function_space.FMult attribute), 32
name (dicee.models.function_space.FMult2 attribute), 33
```

```
name (dicee.models.function_space.GFMult attribute), 32
name (dicee.models.function_space.LFMult attribute), 34
name (dicee.models.function_space.LFMult1 attribute), 33
name (dicee.models.GFMult attribute), 94
name (dicee.models.Keci attribute), 81
name (dicee.models.KeciBase attribute), 84
name (dicee.models.LFMult attribute), 95
name (dicee.models.LFMult1 attribute), 95
name (dicee.models.octonion.AConvO attribute), 38
name (dicee.models.octonion.ConvO attribute), 37
name (dicee.models.octonion.OMult attribute), 36
name (dicee.models.OMult attribute), 79
name (dicee.models.Pyke attribute), 64
name (dicee.models.pykeen_models.PykeenKGE attribute), 38
name (dicee.models.PykeenKGE attribute), 90
name (dicee.models.QMult attribute), 73
name (dicee.models.quaternion.AConvQ attribute), 42
name (dicee.models.quaternion.ConvQ attribute), 41
name (dicee.models.quaternion.QMult attribute), 40
name (dicee.models.real.DistMult attribute), 43
name (dicee.models.real.Pyke attribute), 44
name (dicee.models.real.Shallom attribute). 43
name (dicee.models.real.TransE attribute), 43
name (dicee.models.Shallom attribute), 64
name (dicee.models.TransE attribute), 64
name (dicee.models.transformers.BytE attribute), 45
name (dicee.OMult attribute), 176
name (dicee.Pyke attribute), 160
name (dicee.PykeenKGE attribute), 177
name (dicee.QMult attribute), 174
name (dicee.Shallom attribute), 176
name (dicee. TransE attribute), 164
Namespace (class in dicee.config), 126
neg_ratio (dicee.BPE_NegativeSamplingDataset attribute), 192
neg_ratio (dicee.config.Namespace attribute), 128
neg_ratio (dicee.dataset_classes.BPE_NegativeSamplingDataset attribute), 130
neg ratio (dicee.dataset classes.KvsSampleDataset attribute), 136
neg_ratio (dicee.KvsSampleDataset attribute), 197
neg_sample_ratio (dicee.CVDataModule attribute), 199
neg_sample_ratio (dicee.dataset_classes.CVDataModule attribute), 138
neg_sample_ratio (dicee.dataset_classes.NegSampleDataset attribute), 136
neg_sample_ratio (dicee.dataset_classes.OnevsSample attribute), 134, 135
neg_sample_ratio (dicee.dataset_classes.TriplePredictionDataset attribute), 137
neg_sample_ratio (dicee.NegSampleDataset attribute), 198
neg_sample_ratio (dicee.OnevsSample attribute), 196
neg_sample_ratio (dicee.TriplePredictionDataset attribute), 199
negnorm() (dicee.KGE method), 188
negnorm() (dicee.knowledge_graph_embeddings.KGE method), 150
NegSampleDataset (class in dicee), 198
NegSampleDataset (class in dicee.dataset_classes), 136
neural_searcher (in module dicee.scripts.serve), 106
NeuralSearcher (class in dicee.scripts.serve), 106
NodeTrainer (class in dicee.trainer.torch_trainer_ddp), 110
norm_fc1 (dicee.AConEx attribute), 170
norm_fc1 (dicee.AConvO attribute), 170
norm_fc1 (dicee.ConEx attribute), 173
norm_fc1 (dicee.ConvO attribute), 173
norm_fc1 (dicee.models.AConEx attribute), 68
norm_fc1 (dicee.models.AConvO attribute), 80
norm_fc1 (dicee.models.complex.AConEx attribute), 29
norm fc1 (dicee.models.complex.ConEx attribute), 28
norm_fc1 (dicee.models.ConEx attribute), 67
norm_fc1 (dicee.models.ConvO attribute), 80
norm_fc1 (dicee.models.octonion.AConvO attribute), 38
norm_fc1 (dicee.models.octonion.ConvO attribute), 37
normalization (dicee.analyse_experiments.Experiment attribute), 119
normalization (dicee.config.Namespace attribute), 128
normalize_head_entity_embeddings (dicee.BaseKGE attribute), 182
```

```
normalize_head_entity_embeddings (dicee.models.base_model.BaseKGE attribute), 19
normalize_head_entity_embeddings (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
normalize_relation_embeddings (dicee.BaseKGE attribute), 182
normalize_relation_embeddings (dicee.models.base_model.BaseKGE attribute), 19
normalize_relation_embeddings (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
normalize_tail_entity_embeddings (dicee.BaseKGE attribute), 182
normalize_tail_entity_embeddings (dicee.models.base_model.BaseKGE attribute), 19
normalize_tail_entity_embeddings (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
normalizer_class (dicee.BaseKGE attribute), 181
normalizer_class (dicee.models.base_model.BaseKGE attribute), 19
normalizer_class (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
num_bpe_entities (dicee.BPE_NegativeSamplingDataset attribute), 192
num_bpe_entities (dicee.dataset_classes.BPE_NegativeSamplingDataset attribute), 130
num_bpe_entities (dicee.knowledge_graph.KG attribute), 146
num_core (dicee.config.Namespace attribute), 128
num_datapoints (dicee.BPE_NegativeSamplingDataset attribute), 192
num_datapoints (dicee.dataset_classes.BPE_NegativeSamplingDataset attribute), 130
num_datapoints (dicee.dataset_classes.MultiLabelDataset attribute), 131
num_datapoints (dicee.MultiLabelDataset attribute), 192
num_ent (dicee.DualE attribute), 168
num_ent (dicee.models.DualE attribute), 97
num ent (dicee.models.dualE.DualE attribute), 31
num_entities (dicee.BaseKGE attribute), 181
num_entities (dicee.CVDataModule attribute), 199
num_entities (dicee.dataset_classes.CVDataModule attribute), 138
num_entities (dicee.dataset_classes.KvsSampleDataset attribute), 136
\verb|num_entities| (\textit{dicee.dataset\_classes.NegSampleDataset attribute}), 136
num_entities (dicee.dataset_classes.OnevsSample attribute), 134
\verb|num_entities| (\textit{dicee.dataset\_classes.TriplePredictionDataset attribute}), 137
num_entities (dicee.evaluator.Evaluator attribute), 142
num_entities (dicee.knowledge_graph.KG attribute), 146
num entities (dicee.KvsSampleDataset attribute), 197
num_entities (dicee.models.base_model.BaseKGE attribute), 18
num_entities (dicee.models.BaseKGE attribute), 58, 61, 65, 70, 75, 88, 92
num_entities (dicee.NegSampleDataset attribute), 198
num_entities (dicee.OnevsSample attribute), 196
num entities (dicee. Triple Prediction Dataset attribute), 199
num_epochs (dicee.abstracts.AbstractPPECallback attribute), 117
num_epochs (dicee.analyse_experiments.Experiment attribute), 119
num_epochs (dicee.callbacks.ASWA attribute), 123
num_epochs (dicee.config.Namespace attribute), 127
num_epochs (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 110
num_folds_for_cv (dicee.config.Namespace attribute), 128
\verb|num_of_data_points| (\textit{dicee.dataset\_classes.MultiClassClassificationDataset\ attribute}), 131
num_of_data_points (dicee.MultiClassClassificationDataset attribute), 193
num_of_epochs (dicee.callbacks.PseudoLabellingCallback attribute), 123
num_of_output_channels (dicee.BaseKGE attribute), 181
num_of_output_channels (dicee.config.Namespace attribute), 128
num_of_output_channels (dicee.models.base_model.BaseKGE attribute), 19
num_of_output_channels (dicee.models.BaseKGE attribute), 59, 62, 65, 70, 76, 89, 92
num_params (dicee.analyse_experiments.Experiment attribute), 119
num_relations (dicee.BaseKGE attribute), 181
num_relations (dicee.CVDataModule attribute), 199
num_relations (dicee.dataset_classes.CVDataModule attribute), 138
num_relations (dicee.dataset_classes.NegSampleDataset attribute), 136
num_relations (dicee.dataset_classes.OnevsSample attribute), 134
num_relations (dicee.dataset_classes.TriplePredictionDataset attribute), 137
num_relations (dicee.evaluator.Evaluator attribute), 142
num_relations (dicee.knowledge_graph.KG attribute), 146
num_relations (dicee.models.base_model.BaseKGE attribute), 18
num relations (dicee.models.BaseKGE attribute), 58, 61, 65, 70, 76, 88, 92
num_relations (dicee.NegSampleDataset attribute), 198
num_relations (dicee. Onevs Sample attribute), 196
num_relations (dicee. TriplePredictionDataset attribute), 199
num_sample (dicee.models.FMult attribute), 93
num_sample (dicee.models.function_space.FMult attribute), 32
num_sample (dicee.models.function_space.GFMult attribute), 32
num_sample (dicee.models.GFMult attribute), 94
```

```
num tokens (dicee.BaseKGE attribute), 181
num_tokens (dicee.knowledge_graph.KG attribute), 146
num_tokens (dicee.models.base_model.BaseKGE attribute), 18
num_tokens (dicee.models.BaseKGE attribute), 58, 62, 65, 70, 76, 88, 92
num_workers (dicee.CVDataModule attribute), 199
num_workers (dicee.dataset_classes.CVDataModule attribute), 138
numpy_data_type_changer() (in module dicee), 183
numpy_data_type_changer() (in module dicee.static_funcs), 155
octonion_mul() (in module dicee.models), 78
octonion_mul() (in module dicee.models.octonion), 35
octonion_mul_norm() (in module dicee.models), 78
octonion_mul_norm() (in module dicee.models.octonion), 35
octonion_normalizer() (dicee.AConvO static method), 170
octonion_normalizer() (dicee.ConvO static method), 173
octonion_normalizer() (dicee.models.AConvO static method), 80
octonion_normalizer() (dicee.models.ConvO static method), 80
octonion_normalizer() (dicee.models.octonion.AConvO static method), 38
octonion_normalizer() (dicee.models.octonion.ConvO static method), 37
\verb|octonion_normalizer()| \textit{(dicee.models.octonion.OMult static method)}, 36
octonion_normalizer() (dicee.models.OMult static method), 79
octonion_normalizer() (dicee.OMult static method), 176
OMult (class in dicee), 175
OMult (class in dicee.models), 78
OMult (class in dicee.models.octonion), 35
on_epoch_end() (dicee.callbacks.KGESaveCallback method), 123
on_epoch_end() (dicee.callbacks.PseudoLabellingCallback method), 123
on_fit_end() (dicee.abstracts.AbstractCallback method), 117
on_fit_end() (dicee.abstracts.AbstractPPECallback method), 118
on_fit_end() (dicee.abstracts.AbstractTrainer method), 113
on_fit_end() (dicee.callbacks.AccumulateEpochLossCallback method), 120
on\_fit\_end() (dicee.callbacks.ASWA method), 123
on_fit_end() (dicee.callbacks.Eval method), 125
on_fit_end() (dicee.callbacks.KGESaveCallback method), 122
on_fit_end() (dicee.callbacks.PrintCallback method), 121
on_fit_start() (dicee.abstracts.AbstractCallback method), 116
\verb"on_fit_start"() \textit{ (dicee.abstracts.AbstractPPECallback method)}, 118
on_fit_start() (dicee.abstracts.AbstractTrainer method), 113
on_fit_start() (dicee.callbacks.Eval method), 124
on_fit_start() (dicee.callbacks.KGESaveCallback method), 122
on_fit_start() (dicee.callbacks.KronE method), 126
on_fit_start() (dicee.callbacks.PrintCallback method), 121
on_init_end() (dicee.abstracts.AbstractCallback method), 116
\verb"on_init_start"() \textit{ (dicee.abstracts.AbstractCallback method)}, 116
on_train_batch_end() (dicee.abstracts.AbstractCallback method), 117
on_train_batch_end() (dicee.abstracts.AbstractTrainer method), 114
on_train_batch_end() (dicee.callbacks.Eval method), 125
on_train_batch_end() (dicee.callbacks.KGESaveCallback method), 122
\verb"on_train_batch_end()" (\textit{dicee.callbacks.PrintCallback method}), 121
on_train_batch_start() (dicee.callbacks.Perturb method), 126
on_train_epoch_end() (dicee.abstracts.AbstractCallback method), 117
on_train_epoch_end() (dicee.abstracts.AbstractTrainer method), 113
on_train_epoch_end() (dicee.callbacks.ASWA method), 124
on_train_epoch_end() (dicee.callbacks.Eval method), 125
on_train_epoch_end() (dicee.callbacks.KGESaveCallback method), 122
on_train_epoch_end() (dicee.callbacks.PrintCallback method), 121
on_train_epoch_end() (dicee.models.base_model.BaseKGELightning method), 14
on_train_epoch_end() (dicee.models.BaseKGELightning method), 54
OnevsAllDataset (class in dicee), 193
OnevsAllDataset (class in dicee.dataset_classes), 131
OnevsSample (class in dicee), 195
OnevsSample (class in dicee.dataset_classes), 133
optim (dicee.config.Namespace attribute), 127
optimizer (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 110
optimizer (dicee.trainer.torch_trainer.TorchTrainer attribute), 109
optimizer_name (dicee.BaseKGE attribute), 181
```

```
optimizer name (dicee.models.base model.BaseKGE attribute), 19
optimizer_name (dicee.models.BaseKGE attribute), 58, 62, 65, 70, 76, 88, 92
ordered_bpe_entities (dicee.BPE_NegativeSamplingDataset attribute), 192
ordered_bpe_entities (dicee.dataset_classes.BPE_NegativeSamplingDataset attribute), 130
ordered_bpe_entities (dicee.knowledge_graph.KG attribute), 147
ordered_shaped_bpe_tokens (dicee.knowledge_graph.KG attribute), 146
p (dicee.config.Namespace attribute), 128
p (dicee.DeCaL attribute), 165
p (dicee.Keci attribute), 161
p (dicee.models.clifford.DeCaL attribute), 25
p (dicee.models.clifford.Keci attribute), 22
p (dicee.models.DeCaL attribute), 85
p (dicee.models.Keci attribute), 81
padding (dicee.knowledge_graph.KG attribute), 146
pandas_dataframe_indexer() (in module dicee.read_preprocess_save_load_kg.util), 101
param_init (dicee.BaseKGE attribute), 182
param_init (dicee.models.base_model.BaseKGE attribute), 19
param_init (dicee.models.BaseKGE attribute), 59, 62, 66, 70, 76, 89, 92
parameters () (dicee.abstracts.BaseInteractiveKGE method), 116
path (dicee.abstracts.AbstractPPECallback attribute), 117
path (dicee.callbacks.AccumulateEpochLossCallback attribute), 120
path (dicee.callbacks.ASWA attribute), 123
path (dicee.callbacks.Eval attribute), 124
path (dicee.callbacks.KGESaveCallback attribute), 122
path_dataset_folder (dicee.analyse_experiments.Experiment attribute), 119
path_for_deserialization (dicee.knowledge_graph.KG attribute), 146
path_for_serialization (dicee.knowledge_graph.KG attribute), 146
path_single_kg (dicee.config.Namespace attribute), 127
path_single_kg (dicee.knowledge_graph.KG attribute), 146
path_to_store_single_run (dicee.config.Namespace attribute), 127
Perturb (class in dicee.callbacks), 126
polars_dataframe_indexer() (in module dicee.read_preprocess_save_load_kg.util), 100
poly_NN() (dicee.LFMult method), 177
poly_NN() (dicee.models.function_space.LFMult method), 34
poly_NN() (dicee.models.LFMult method), 96
polynomial() (dicee.LFMult method), 177
polynomial() (dicee.models.function_space.LFMult method), 35
polynomial() (dicee.models.LFMult method), 96
pop() (dicee.LFMult method), 177
pop () (dicee.models.function_space.LFMult method), 35
pop() (dicee.models.LFMult method), 96
pq (dicee.analyse_experiments.Experiment attribute), 119
predict () (dicee.KGE method), 187
predict() (dicee.knowledge_graph_embeddings.KGE method), 149
predict_dataloader() (dicee.models.base_model.BaseKGELightning method), 15
predict_dataloader() (dicee.models.BaseKGELightning method), 55
predict_missing_head_entity() (dicee.KGE method), 186
predict_missing_head_entity() (dicee.knowledge_graph_embeddings.KGE method), 147
predict_missing_relations() (dicee.KGE method), 187
predict_missing_relations() (dicee.knowledge_graph_embeddings.KGE method), 148
predict_missing_tail_entity() (dicee.KGE method), 187
predict_missing_tail_entity() (dicee.knowledge_graph_embeddings.KGE method), 148
predict_topk() (dicee.KGE method), 187
predict_topk() (dicee.knowledge_graph_embeddings.KGE method), 149
prepare_data() (dicee.CVDataModule method), 201
prepare_data() (dicee.dataset_classes.CVDataModule method), 140
preprocess_with_byte_pair_encoding() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 104
preprocess_with_byte_pair_encoding() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 98
preprocess_with_byte_pair_encoding_with_padding() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 104
preprocess_with_byte_pair_encoding_with_padding() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 98
preprocess_with_pandas() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 104
preprocess_with_pandas() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 98
preprocess_with_polars() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 104
preprocess_with_polars() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 98
```

preprocesses_input_args() (in module dicee.static_preprocess_funcs), 157

```
PreprocessKG (class in dicee.read_preprocess_save_load_kg), 103
PreprocessKG (class in dicee.read_preprocess_save_load_kg.preprocess), 98
previous args (dicee.executer.ContinuousExecute attribute), 145
PrintCallback (class in dicee.callbacks), 120
process (dicee.trainer.torch_trainer.TorchTrainer attribute), 109
PseudoLabellingCallback (class in dicee.callbacks), 123
ptdtype (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 111
Pyke (class in dicee), 160
Pyke (class in dicee.models), 64
Pyke (class in dicee.models.real), 43
pykeen_model_kwargs (dicee.config.Namespace attribute), 128
PykeenKGE (class in dicee), 177
PykeenKGE (class in dicee.models), 90
PykeenKGE (class in dicee.models.pykeen_models), 38
Q
q (dicee.config.Namespace attribute), 128
q (dicee.DeCaL attribute), 165
q (dicee.Keci attribute), 161
q (dicee.models.clifford.DeCaL attribute), 25
q (dicee.models.clifford.Keci attribute), 22
q (dicee.models.DeCaL attribute), 85
\neq (\textit{dicee.models.Keci attribute}),\,81
qdrant_client (dicee.scripts.serve.NeuralSearcher attribute), 106
QMult (class in dicee), 173
QMult (class in dicee.models), 72
QMult (class in dicee.models.quaternion), 40
quaternion_mul() (in module dicee.models), 69
quaternion_mul() (in module dicee.models.static_funcs), 44
quaternion_mul_with_unit_norm() (in module dicee.models), 72
quaternion_mul_with_unit_norm() (in module dicee.models.quaternion), 40
quaternion_multiplication_followed_by_inner_product() (dicee.models.QMult method), 73
\verb|quaternion_multiplication_followed_by_inner_product()| \textit{(dicee.models.quaternion.QMult method)}, 40
quaternion_multiplication_followed_by_inner_product() (dicee.QMult method), 174
quaternion_normalizer() (dicee.models.QMult static method), 73
quaternion_normalizer() (dicee.models.quaternion.QMult static method), 40
quaternion_normalizer() (dicee.QMult static method), 174
query_name_to_struct (dicee.query_generator.QueryGenerator attribute), 152
query_name_to_struct (dicee.QueryGenerator attribute), 203
QueryGenerator (class in dicee), 202
QueryGenerator (class in dicee.query_generator), 151
r (dicee.DeCaL attribute), 165
r (dicee. Keci attribute), 161
r (dicee.models.clifford.DeCaL attribute), 25
r (dicee.models.clifford.Keci attribute), 22
r (dicee.models.DeCaL attribute), 85
r (dicee.models.Keci attribute), 81
random_prediction() (in module dicee), 184
random_prediction() (in module dicee.static_funcs), 155
random_seed (dicee.config.Namespace attribute), 128
ratio (dicee.callbacks.Perturb attribute), 126
re (dicee.DeCaL attribute), 165
re (dicee.models.clifford.DeCaL attribute), 25
re (dicee.models.DeCaL attribute), 85
re_vocab (dicee.evaluator.Evaluator attribute), 142
read_from_disk() (in module dicee.read_preprocess_save_load_kg.util), 102
read_from_triple_store() (in module dicee.read_preprocess_save_load_kg.util), 102
read_only_few (dicee.config.Namespace attribute), 128
\verb"read_only_few" (\textit{dicee.knowledge\_graph.KG attribute}), 146
read_or_load_kg() (in module dicee), 184
read_or_load_kg() (in module dicee.static_funcs), 155
read_with_pandas() (in module dicee.read_preprocess_save_load_kg.util), 102
read_with_polars() (in module dicee.read_preprocess_save_load_kg.util), 102
ReadFromDisk (class in dicee.read_preprocess_save_load_kg), 104
ReadFromDisk (class in dicee.read_preprocess_save_load_kg.read_from_disk), 98
```

```
rel2id (dicee.query generator.QueryGenerator attribute), 151
rel2id (dicee.QueryGenerator attribute), 203
relation embeddings (dicee. AConvO attribute), 171
relation_embeddings (dicee.ConvQ attribute), 171
relation_embeddings (dicee.DeCaL attribute), 165
relation_embeddings (dicee.DualE attribute), 168
relation_embeddings (dicee.LFMult attribute), 176
relation embeddings (dicee.models.AConvO attribute), 74
relation_embeddings (dicee.models.clifford.DeCaL attribute), 25
relation_embeddings (dicee.models.ConvQ attribute).74
relation_embeddings (dicee.models.DeCaL attribute), 85
{\tt relation\_embeddings}~(\textit{dicee.models.DualE attribute}),\,97
relation_embeddings (dicee.models.dualE.DualE attribute), 31
relation_embeddings (dicee.models.FMult attribute), 93
relation_embeddings (dicee.models.FMult2 attribute), 95
relation_embeddings (dicee.models.function_space.FMult attribute), 32
relation_embeddings (dicee.models.function_space.FMult2 attribute), 33
relation_embeddings (dicee.models.function_space.GFMult attribute), 32
relation_embeddings (dicee.models.function_space.LFMult attribute), 34
relation_embeddings (dicee.models.function_space.LFMult1 attribute), 33
relation_embeddings (dicee.models.GFMult attribute), 94
relation_embeddings (dicee.models.LFMult attribute), 95
relation_embeddings (dicee.models.LFMult1 attribute), 95
relation_embeddings (dicee.models.pykeen_models.PykeenKGE attribute), 39
relation_embeddings (dicee.models.PykeenKGE attribute), 90
relation_embeddings (dicee.models.quaternion.AConvQ attribute), 42
relation_embeddings (dicee.models.quaternion.ConvQ attribute), 41
relation_embeddings (dicee.PykeenKGE attribute), 178
relation_to_idx (dicee.knowledge_graph.KG attribute), 146
relations_str (dicee.knowledge_graph.KG property), 147
reload_dataset() (in module dicee), 191
reload dataset () (in module dicee.dataset classes), 130
report (dicee.DICE_Trainer attribute), 185
report (dicee.evaluator.Evaluator attribute), 142
report (dicee. Execute attribute), 190
report (dicee.executer.Execute attribute), 144
report (dicee.trainer.DICE Trainer attribute), 111
report (dicee.trainer.dice_trainer.DICE_Trainer attribute), 107
reports (dicee.callbacks.Eval attribute), 124
requires_grad_for_interactions (dicee.Keci attribute), 162
requires_grad_for_interactions (dicee.KeciBase attribute), 161
requires_grad_for_interactions (dicee.models.clifford.Keci attribute), 22
requires_grad_for_interactions (dicee.models.clifford.KeciBase attribute), 24
requires_grad_for_interactions (dicee.models.Keci attribute), 81
requires_grad_for_interactions (dicee.models.KeciBase attribute), 84
resid_dropout (dicee.models.transformers.CausalSelfAttention attribute), 48
residual_convolution() (dicee.AConEx method), 170
residual_convolution() (dicee.AConvO method), 170
residual_convolution() (dicee.AConvQ method), 171
residual_convolution() (dicee.ConEx method), 173
residual_convolution() (dicee.ConvO method), 173
residual_convolution() (dicee.ConvQ method), 172
residual_convolution() (dicee.models.AConEx method), 68
residual_convolution() (dicee.models.AConvO method), 80
residual_convolution() (dicee.models.AConvQ method), 75
residual_convolution() (dicee.models.complex.AConEx method), 29
residual_convolution() (dicee.models.complex.ConEx method), 28
residual_convolution() (dicee.models.ConEx method), 67
residual_convolution() (dicee.models.ConvO method), 80
residual_convolution() (dicee.models.ConvQ method), 74
residual convolution() (dicee.models.octonion.AConvO method), 38
residual_convolution() (dicee.models.octonion.ConvO method), 37
residual_convolution() (dicee.models.quaternion.AConvQ method), 42
residual_convolution() (dicee.models.quaternion.ConvQ method), 42
retrieve_embeddings() (in module dicee.scripts.serve), 106
return_multi_hop_query_results() (dicee.KGE method), 188
return_multi_hop_query_results() (dicee.knowledge_graph_embeddings.KGE method), 150
root () (in module dicee.scripts.serve), 106
```

```
roots (dicee.models.FMult attribute), 94
roots (dicee.models.function_space.FMult attribute), 32
roots (dicee.models.function_space.GFMult attribute), 32
roots (dicee.models.GFMult attribute), 94
\verb"runtime" (\textit{dicee.analyse\_experiments.Experiment attribute}), 119
S
sample_counter (dicee.abstracts.AbstractPPECallback attribute), 118
sample entity() (dicee.abstracts.BaseInteractiveKGE method), 115
sample_relation() (dicee.abstracts.BaseInteractiveKGE method), 115
sample_triples_ratio (dicee.config.Namespace attribute), 128
sample_triples_ratio (dicee.knowledge_graph.KG attribute), 146
sanity_checking_with_arguments() (in module dicee.sanity_checkers), 153
save() (dicee.abstracts.BaseInteractiveKGE method), 115
save() (dicee.read_preprocess_save_load_kg.LoadSaveToDisk method), 104
save() (dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk method), 99
save_checkpoint() (dicee.abstracts.AbstractTrainer static method), 114
save_checkpoint_model() (in module dicee), 183
save_checkpoint_model() (in module dicee.static_funcs), 155
save_embeddings() (in module dicee), 184
save_embeddings() (in module dicee.static_funcs), 155
save_embeddings_as_csv (dicee.config.Namespace attribute), 127
save_experiment() (dicee.analyse_experiments.Experiment method), 119
save_model_at_every_epoch (dicee.config.Namespace attribute), 128
save_numpy_ndarray() (in module dicee), 183
save_numpy_ndarray() (in module dicee.read_preprocess_save_load_kg.util), 103
save_numpy_ndarray() (in module dicee.static_funcs), 155
save_pickle() (in module dicee), 183
save_pickle() (in module dicee.read_preprocess_save_load_kg.util), 103
save_pickle() (in module dicee.static_funcs), 154
save_queries() (dicee.query_generator.QueryGenerator method), 152
save_queries() (dicee.QueryGenerator method), 203
{\tt save\_queries\_and\_answers} \ () \ \textit{(dicee.query\_generator.QueryGenerator static method)}, 152
save_queries_and_answers() (dicee.QueryGenerator static method), 204
save_trained_model() (dicee.Execute method), 190
save_trained_model() (dicee.executer.Execute method), 144
scalar_batch_NN() (dicee.LFMult method), 177
\verb|scalar_batch_NN()| \textit{ (dicee.models.function\_space.LFMult method)}, 34
scalar_batch_NN() (dicee.models.LFMult method), 96
scaler (dicee.callbacks.Perturb attribute), 126
scaler (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 111
score () (dicee.ComplEx static method), 169
score () (dicee.DistMult method), 161
score () (dicee. Keci method), 164
score () (dicee.models.clifford.Keci method), 24
score() (dicee.models.ComplEx static method), 69
score () (dicee.models.complex.ComplEx static method), 30
score () (dicee.models.DistMult method), 64
score () (dicee.models.Keci method), 84
score () (dicee.models.octonion.OMult method), 36
score () (dicee.models.OMult method), 79
score () (dicee.models.QMult method), 73
score() (dicee.models.quaternion.QMult method), 41
score() (dicee.models.real.DistMult method), 43
score() (dicee.models.real.TransE method), 43
score() (dicee.models.TransE method), 64
score () (dicee.OMult method), 176
score () (dicee.QMult method), 175
score () (dicee. TransE method), 164
score_func (dicee.models.FMult2 attribute), 94
score_func (dicee.models.function_space.FMult2 attribute), 33
scoring_technique (dicee.analyse_experiments.Experiment attribute), 119
scoring_technique (dicee.config.Namespace attribute), 127
search() (dicee.scripts.serve.NeuralSearcher method), 106
search_embeddings() (in module dicee.scripts.serve), 106
seed (dicee.query_generator.QueryGenerator attribute), 151
seed (dicee.QueryGenerator attribute), 203
```

```
select model() (in module dicee), 183
select_model() (in module dicee.static_funcs), 154
selected_optimizer (dicee.BaseKGE attribute), 181
selected_optimizer (dicee.models.base_model.BaseKGE attribute), 19
selected_optimizer (dicee.models.BaseKGE attribute), 59, 62, 65, 70, 76, 89, 92
separator (dicee.config.Namespace attribute), 127
separator (dicee.knowledge_graph.KG attribute), 147
sequential_vocabulary_construction() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 104
sequential_vocabulary_construction() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 98
set_global_seed() (dicee.query_generator.QueryGenerator method), 152
set_global_seed() (dicee.QueryGenerator method), 203
\verb|set_model_eval_mode()| \textit{ (dicee. abstracts. Base Interactive KGE method)}, 115
set_model_train_mode() (dicee.abstracts.BaseInteractiveKGE method), 115
setup() (dicee.CVDataModule method), 200
setup() (dicee.dataset_classes.CVDataModule method), 138
setup_executor() (dicee.Execute method), 190
setup_executor() (dicee.executer.Execute method), 144
Shallom (class in dicee), 176
Shallom (class in dicee.models), 64
Shallom (class in dicee.models.real), 43
shallom (dicee.models.real.Shallom attribute), 43
shallom (dicee.models.Shallom attribute), 64
shallom (dicee.Shallom attribute), 176
shallom_width (dicee.models.real.Shallom attribute), 43
shallom_width (dicee.models.Shallom attribute), 64
shallom_width (dicee.Shallom attribute), 176
single_hop_query_answering() (dicee.KGE method), 188
single_hop_query_answering() (dicee.knowledge_graph_embeddings.KGE method), 150
sparql_endpoint (dicee.config.Namespace attribute), 127
spargl_endpoint (dicee.knowledge_graph.KG attribute), 146
start() (dicee.DICE_Trainer method), 185
start () (dicee.Execute method), 191
start () (dicee.executer.Execute method), 145
start() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 103
start() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 98
start() (dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method), 98
start() (dicee.read_preprocess_save_load_kg.ReadFromDisk method), 104
start () (dicee.trainer.DICE_Trainer method), 112
start () (dicee.trainer.dice_trainer.DICE_Trainer method), 108
start_time (dicee.callbacks.PrintCallback attribute), 121
start_time (dicee.Execute attribute), 190
start_time (dicee.executer.Execute attribute), 144
storage_path (dicee.config.Namespace attribute), 127
storage_path (dicee.DICE_Trainer attribute), 185
storage_path (dicee.trainer.DICE_Trainer attribute), 111
storage_path (dicee.trainer.dice_trainer.DICE_Trainer attribute), 107
store (dicee. Allvs All attribute), 195
store (dicee.dataset_classes.AllvsAll attribute), 133
store (dicee.dataset_classes.KvsSampleDataset attribute), 136
store (dicee. KvsSampleDataset attribute), 197
store () (in module dicee), 183
store() (in module dicee.static_funcs), 155
\verb|store_ensemble()| \textit{(dicee.abstracts.AbstractPPECallback method)}, 118
strategy (dicee.abstracts.AbstractTrainer attribute), 113
swa (dicee.config.Namespace attribute), 129
Т
T() (dicee.DualE method), 168
T() (dicee.models.DualE method), 97
T() (dicee.models.dualE.DualE method), 31
t_conorm() (dicee.KGE method), 188
t_conorm() (dicee.knowledge_graph_embeddings.KGE method), 150
t_norm() (dicee.KGE method), 188
t_norm() (dicee.knowledge_graph_embeddings.KGE method), 150
target_dim (dicee.AllvsAll attribute), 195
target_dim (dicee.dataset_classes.AllvsAll attribute), 133
target_dim (dicee.dataset_classes.MultiLabelDataset attribute), 131
```

```
target dim (dicee.dataset classes.OnevsAllDataset attribute), 132
target_dim (dicee.knowledge_graph.KG attribute), 147
target_dim (dicee.MultiLabelDataset attribute), 192
target_dim (dicee.OnevsAllDataset attribute), 193
temperature (dicee. BytE attribute), 179
temperature (dicee.models.transformers.BytE attribute), 45
tensor_t_norm() (dicee.KGE method), 188
tensor_t_norm() (dicee.knowledge_graph_embeddings.KGE method), 150
test_dataloader() (dicee.models.base_model.BaseKGELightning method), 14
test_dataloader() (dicee.models.BaseKGELightning method), 54
test_epoch_end() (dicee.models.base_model.BaseKGELightning method), 14
test_epoch_end() (dicee.models.BaseKGELightning method), 54
test_h1 (dicee.analyse_experiments.Experiment attribute), 119
test_h3 (dicee.analyse_experiments.Experiment attribute), 119
test_h10 (dicee.analyse_experiments.Experiment attribute), 119
test_mrr (dicee.analyse_experiments.Experiment attribute), 119
test_path (dicee.query_generator.QueryGenerator attribute), 151
test_path (dicee.QueryGenerator attribute), 203
timeit() (in module dicee), 183, 191
timeit() (in module dicee.read_preprocess_save_load_kg.util), 102
timeit() (in module dicee.static_funcs), 154
timeit() (in module dicee.static_preprocess_funcs), 157
to() (dicee.KGE method), 186
to() (dicee.knowledge_graph_embeddings.KGE method), 147
to_df() (dicee.analyse_experiments.Experiment method), 119
topk (dicee.BytE attribute), 179
topk (dicee.models.transformers.BytE attribute), 45
torch_ordered_shaped_bpe_entities (dicee.dataset_classes.MultiLabelDataset attribute), 131
torch_ordered_shaped_bpe_entities (dicee.MultiLabelDataset attribute), 192
TorchDDPTrainer (class in dicee.trainer.torch_trainer_ddp), 110
TorchTrainer (class in dicee.trainer.torch_trainer), 108
train() (dicee.KGE method), 190
train() (dicee.knowledge_graph_embeddings.KGE method), 151
train() (dicee.trainer.torch_trainer_ddp.NodeTrainer method), 111
train_data (dicee. Allvs All attribute), 195
train_data (dicee.dataset_classes.AllvsAll attribute), 133
train data (dicee.dataset classes.KvsAll attribute), 132
train_data (dicee.dataset_classes.KvsSampleDataset attribute), 136
train_data (dicee.dataset_classes.MultiClassClassificationDataset attribute), 131
train_data (dicee.dataset_classes.OnevsAllDataset attribute), 132
train_data (dicee.dataset_classes.OnevsSample attribute), 134
train_data (dicee.KvsAll attribute), 194
train_data (dicee.KvsSampleDataset attribute), 197
train_data (dicee.MultiClassClassificationDataset attribute), 193
train_data (dicee.OnevsAllDataset attribute), 193
train_data (dicee.OnevsSample attribute), 195, 196
train_dataloader() (dicee.CVDataModule method), 199
train_dataloader() (dicee.dataset_classes.CVDataModule method), 138
train_dataloader() (dicee.models.base_model.BaseKGELightning method), 16
train_dataloader() (dicee.models.BaseKGELightning method), 56
train_dataloaders (dicee.trainer.torch_trainer.TorchTrainer attribute), 109
train_dataset_loader (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 110
train_h1 (dicee.analyse_experiments.Experiment attribute), 119
train_h3 (dicee.analyse_experiments.Experiment attribute), 119
train_h10 (dicee.analyse_experiments.Experiment attribute), 119
train_indices_target (dicee.dataset_classes.MultiLabelDataset attribute), 131
train_indices_target (dicee.MultiLabelDataset attribute), 192
{\tt train\_k\_vs\_all()} \ (\textit{dicee.KGE method}), \, 189
train_k_vs_all() (dicee.knowledge_graph_embeddings.KGE method), 151
train_mrr (dicee.analyse_experiments.Experiment attribute), 119
train path (dicee.query generator, Query Generator attribute), 151
train_path (dicee.QueryGenerator attribute), 202
train_set (dicee.BPE_NegativeSamplingDataset attribute), 192
train_set (dicee.dataset_classes.BPE_NegativeSamplingDataset attribute), 130
train_set (dicee.dataset_classes.MultiLabelDataset attribute), 131
train_set (dicee.dataset_classes.NegSampleDataset attribute), 136
train_set (dicee.dataset_classes.TriplePredictionDataset attribute), 137
train_set (dicee.MultiLabelDataset attribute), 192
```

```
train_set (dicee.NegSampleDataset attribute), 198
train_set (dicee. TriplePredictionDataset attribute), 199
train_set_idx (dicee.CVDataModule attribute), 199
train_set_idx (dicee.dataset_classes.CVDataModule attribute), 138
train_set_target (dicee.knowledge_graph.KG attribute), 146
train_target (dicee.AllvsAll attribute), 195
train_target (dicee.dataset_classes.AllvsAll attribute), 133
train target (dicee.dataset classes.KvsAll attribute), 132
train_target (dicee.dataset_classes.KvsSampleDataset attribute), 136
train_target (dicee.KvsAll attribute), 194
train_target (dicee.KvsSampleDataset attribute), 197
{\tt train\_target\_indices}~(\textit{dicee.knowledge\_graph.KG attribute}),~147
train_triples() (dicee.KGE method), 189
train_triples() (dicee.knowledge_graph_embeddings.KGE method), 151
trained_model (dicee.Execute attribute), 190
trained_model (dicee.executer.Execute attribute), 144
trainer (dicee.config.Namespace attribute), 127
trainer (dicee.DICE_Trainer attribute), 185
trainer (dicee. Execute attribute), 190
trainer (dicee.executer.Execute attribute), 144
trainer (dicee.trainer.DICE_Trainer attribute), 111
trainer (dicee.trainer.dice_trainer.DICE_Trainer attribute), 107
trainer (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 110
training_step (dicee.trainer.torch_trainer.TorchTrainer attribute), 109
training_step() (dicee.BytE method), 179
training_step() (dicee.models.base_model.BaseKGELightning method), 13
training_step() (dicee.models.BaseKGELightning method), 53
training_step() (dicee.models.transformers.BytE method), 46
training_step_outputs (dicee.models.base_model.BaseKGELightning attribute), 13
training_step_outputs (dicee.models.BaseKGELightning attribute), 53
training_technique (dicee.knowledge_graph.KG attribute), 146
TransE (class in dicee), 164
TransE (class in dicee.models), 64
TransE (class in dicee.models.real), 43
transfer_batch_to_device() (dicee.CVDataModule method), 200
transfer_batch_to_device() (dicee.dataset_classes.CVDataModule method), 139
transformer (dicee.BytE attribute), 179
transformer (dicee.models.transformers.BytE attribute), 45
transformer (dicee.models.transformers.GPT attribute), 50
trapezoid() (dicee.models.FMult2 method), 95
trapezoid() (dicee.models.function_space.FMult2 method), 33
tri_score() (dicee.LFMult method), 177
tri_score() (dicee.models.function_space.LFMult method), 34
\verb|tri_score|()| \textit{(dicee.models.function\_space.LFMult1 method)}, 34
tri_score() (dicee.models.LFMult method), 96
tri_score() (dicee.models.LFMult1 method), 95
triple_score() (dicee.KGE method), 188
triple_score() (dicee.knowledge_graph_embeddings.KGE method), 149
TriplePredictionDataset (class in dicee), 198
TriplePredictionDataset (class in dicee.dataset_classes), 136
{\tt tuned\_embedding\_dim}~(\textit{dicee.models.FMult2 attribute}), 94
tuned_embedding_dim (dicee.models.function_space.FMult2 attribute), 33
tuple2list() (dicee.query_generator.QueryGenerator method), 152
tuple2list() (dicee.QueryGenerator method), 203
unlabelled_size (dicee.callbacks.PseudoLabellingCallback attribute), 123
unmap() (dicee.query_generator.QueryGenerator method), 152
unmap () (dicee.QueryGenerator method), 203
unmap_query() (dicee.query_generator.QueryGenerator method), 152
unmap_query() (dicee.QueryGenerator method), 203
val_aswa (dicee.callbacks.ASWA attribute), 123
val_dataloader() (dicee.models.base_model.BaseKGELightning method), 15
val_dataloader() (dicee.models.BaseKGELightning method), 55
val_h1 (dicee.analyse_experiments.Experiment attribute), 119
```

```
val h3 (dicee.analyse experiments.Experiment attribute), 119
val_h10 (dicee.analyse_experiments.Experiment attribute), 119
val_mrr (dicee.analyse_experiments.Experiment attribute), 119
val_path (dicee.query_generator.QueryGenerator attribute), 151
val_path (dicee.QueryGenerator attribute), 203
validate_knowledge_graph() (in module dicee.sanity_checkers), 153
vocab_preparation() (dicee.evaluator.Evaluator method), 142
vocab_size (dicee.models.transformers.GPTConfig attribute), 49
vocab_to_parquet() (in module dicee), 184
vocab_to_parquet() (in module dicee.static_funcs), 155
vtp_score() (dicee.LFMult method), 177
vtp_score() (dicee.models.function_space.LFMult method), 34
vtp_score() (dicee.models.function_space.LFMult1 method), 34
vtp_score() (dicee.models.LFMult method), 96
vtp_score() (dicee.models.LFMult1 method), 95
weight (dicee.BytE attribute), 179
weight (dicee.models.transformers.BytE attribute), 45
weight (dicee.models.transformers.GPT attribute), 50
weight (dicee.models.transformers.LayerNorm attribute), 47
weight_decay (dicee.BaseKGE attribute), 181
\verb|weight_decay| \textit{(dicee.config.Namespace attribute)}, 128
weight_decay (dicee.models.base_model.BaseKGE attribute), 19
weight_decay (dicee.models.BaseKGE attribute), 59, 62, 65, 70, 76, 89, 92
weights (dicee.models.FMult attribute), 94
weights (dicee.models.function_space.FMult attribute), 32
weights (dicee.models.function_space.GFMult attribute), 33
weights (dicee.models.GFMult attribute), 94
\verb|write_links|| (\textit{dicee.query\_generator.QueryGenerator method}), 152
write_links() (dicee.QueryGenerator method), 203
write_report() (dicee.Execute method), 191
write_report() (dicee.executer.Execute method), 145
X
x_values (dicee.LFMult attribute), 176
x_values (dicee.models.function_space.LFMult attribute), 34
x_values (dicee.models.LFMult attribute), 95
```