
DICE Embeddings

Release 0.1.3.2

Caglar Demir

Nov 20, 2025

Contents:

1	Dicee Manual	2
2	Installation	3
2.1	Installation from Source	3
3	Download Knowledge Graphs	3
4	Knowledge Graph Embedding Models	3
5	How to Train	3
6	Creating an Embedding Vector Database	5
6.1	Learning Embeddings	5
6.2	Loading Embeddings into Qdrant Vector Database	6
6.3	Launching Webservice	6
7	Answering Complex Queries	6
8	Predicting Missing Links	8
9	Downloading Pretrained Models	8
10	How to Deploy	8
11	Docker	8
12	Coverage Report	8
13	How to cite	10
14	dicee	12
14.1	Submodules	12
14.2	Attributes	183
14.3	Classes	184
14.4	Functions	185
14.5	Package Contents	186
	Python Module Index	233

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

1 Dicee Manual

Version: dicee 0.2.0

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

Publisher and maintainer: Caglar Demir²

Contact: caglar.demir@upb.de

License: OSI Approved :: MIT License

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

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

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

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

Why Pandas⁶ & Co. ? A large knowledge graph can be read and preprocessed (e.g. removing literals) by pandas, modin, or polars in parallel. Through polars, a knowledge graph having more than 1 billion triples can be read in parallel fashion. Importantly, using these frameworks allow us to perform all necessary computations on a single CPU as well as a cluster of computers.

Why PyTorch⁷ & Co. ? PyTorch is one of the most popular machine learning frameworks available at the time of writing. PytorchLightning facilitates scaling the training procedure of PyTorch without boilerplate. In our framework, we combine **PyTorch**⁸ & **PytorchLightning**⁹. Users can choose the trainer class (e.g., DDP by Pytorch) to train large knowledge graph embedding models with billions of parameters. PytorchLightning allows us to use state-of-the-art model parallelism techniques (e.g. Fully Sharded Training, FairScale, or DeepSpeed) without extra effort. With our framework, practitioners can directly use PytorchLightning for model parallelism to train gigantic embedding models.

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

where the data is in the following form

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

A KGE model can also be trained from the command line

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

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

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

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

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

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

Similarly, models can be easily trained with torchrun

```

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

```

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

```

torchrun --nnodes 2 --nproc_per_node=gpu --node_rank 0 --rdzv_id 455 --rdzv_backend_
→c10d --rdzv_endpoint=nebula dicee/scripts/run.py --trainer torchDDP --dataset_dir_
→KGs/UMLS
torchrun --nnodes 2 --nproc_per_node=gpu --node_rank 1 --rdzv_id 455 --rdzv_backend_
→c10d --rdzv_endpoint=nebula dicee/scripts/run.py --trainer torchDDP --dataset_dir_
→KGs/UMLS

```

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

```

dicee --path_single_kg "KGs/Family/family-benchmark_rich_background.owl" --model Keci_
→--path_to_store_single_run KeciFamilyRun --backend rdflib

```

where the data is in the following form

```

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

```

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

```

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

```

For more, please refer to examples.

6 Creating an Embedding Vector Database

6.1 Learning Embeddings

```

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

```

6.2 Loading Embeddings into Qdrant Vector Database

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

6.3 Launching Webservice

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

Retrieve and Search

Get embedding of germany

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

Get most similar things to europe

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

7 Answering Complex Queries

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

(continues on next page)

(continued from previous page)

```
pre_trained_kge = KGE(path=result['path_experiment_folder'])
# (3) Single-hop query answering
# Query: ?E : \exist E.hasSibling(E, F9M167)
# Question: Who are the siblings of F9M167?
# Answer: [F9M157, F9F141], as (F9M167, hasSibling, F9M157) and (F9M167, hasSibling, ↵
↵F9F141)
predictions = pre_trained_kge.answer_multi_hop_query(query_type="1p",
                                                       query=('http://www.benchmark.org/
↵family#F9M167',
                                                         ('http://www.benchmark.
↵org/family#hasSibling',)),
                                                       tnorm="min", k=3)
top_entities = [topk_entity for topk_entity, query_score in predictions]
assert "http://www.benchmark.org/family#F9F141" in top_entities
assert "http://www.benchmark.org/family#F9M157" in top_entities
# (2) Two-hop query answering
# Query: ?D : \exist E.Married(D, E) \land hasSibling(E, F9M167)
# Question: To whom a sibling of F9M167 is married to?
# Answer: [F9F158, F9M142] as (F9M157 #married F9F158) and (F9F141 #married F9M142)
predictions = pre_trained_kge.answer_multi_hop_query(query_type="2p",
                                                       query=("http://www.benchmark.org/
↵family#F9M167",
                                                         ("http://www.benchmark.
↵org/family#hasSibling",
                                                         "http://www.benchmark.
↵org/family#married")),
                                                       tnorm="min", k=3)
top_entities = [topk_entity for topk_entity, query_score in predictions]
assert "http://www.benchmark.org/family#F9M142" in top_entities
assert "http://www.benchmark.org/family#F9F158" in top_entities
# (3) Three-hop query answering
# Query: ?T : \exist D.type(D,T) \land Married(D,E) \land hasSibling(E, F9M167)
# Question: What are the type of people who are married to a sibling of F9M167?
# (3) Answer: [Person, Male, Father] since F9M157 is [Brother Father Grandfather ↵
↵Male] and F9M142 is [Male Grandfather Father]
predictions = pre_trained_kge.answer_multi_hop_query(query_type="3p", query=("http://
↵www.benchmark.org/family#F9M167",
                                                         ("http://
↵www.benchmark.org/family#hasSibling",
                                                         "http://
↵www.benchmark.org/family#married",
                                                         "http://
↵www.w3.org/1999/02/22-rdf-syntax-ns#type")),
                                                       tnorm="min", k=5)
top_entities = [topk_entity for topk_entity, query_score in predictions]
print(top_entities)
assert "http://www.benchmark.org/family#Person" in top_entities
assert "http://www.benchmark.org/family#Father" in top_entities
assert "http://www.benchmark.org/family#Male" in top_entities
```

For more, please refer to examples/multi_hop_query_answering.

8 Predicting Missing Links

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

9 Downloading Pretrained Models

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

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

10 How to Deploy

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

11 Docker

To build the Docker image:

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

To test the Docker image:

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

12 Coverage Report

The coverage report is generated using `coverage.py`¹²:

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

(continues on next page)

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

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

(continued from previous page)

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

(continues on next page)

(continued from previous page)

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

TOTAL	6948	3169	54%	

13 How to cite

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

```
# Keci
@inproceedings{demir2023clifford,
  title={Clifford Embeddings--A Generalized Approach for Embedding in Normed Algebras}
  ↪,
```

(continues on next page)

```

    author={Demir, Caglar and Ngonga Ngomo, Axel-Cyrille},
    booktitle={Joint European Conference on Machine Learning and Knowledge Discovery in
↪Databases},
    pages={567--582},
    year={2023},
    organization={Springer}
}
# LitCQD
@inproceedings{demir2023litcq,
    title={LitCQD: Multi-Hop Reasoning in Incomplete Knowledge Graphs with Numeric
↪Literals},
    author={Demir, Caglar and Wiebesiek, Michel and Lu, Renzhong and Ngonga Ngomo, Axel-
↪Cyrille and Heindorf, Stefan},
    booktitle={Joint European Conference on Machine Learning and Knowledge Discovery in
↪Databases},
    pages={617--633},
    year={2023},
    organization={Springer}
}
# DICE Embedding Framework
@article{demir2022hardware,
    title={Hardware-agnostic computation for large-scale knowledge graph embeddings},
    author={Demir, Caglar and Ngomo, Axel-Cyrille Ngonga},
    journal={Software Impacts},
    year={2022},
    publisher={Elsevier}
}
# KronE
@inproceedings{demir2022kronecker,
    title={Kronecker decomposition for knowledge graph embeddings},
    author={Demir, Caglar and Lienen, Julian and Ngonga Ngomo, Axel-Cyrille},
    booktitle={Proceedings of the 33rd ACM Conference on Hypertext and Social Media},
    pages={1--10},
    year={2022}
}
# QMult, OMult, ConvQ, ConvO
@InProceedings{pmlr-v157-demir21a,
    title = {Convolutional Hypercomplex Embeddings for Link Prediction},
    author = {Demir, Caglar and Moussallem, Diego and Heindorf, Stefan and Ngonga
↪Ngomo, Axel-Cyrille},
    booktitle = {Proceedings of The 13th Asian Conference on Machine Learning},
    pages = {656--671},
    year = {2021},
    editor = {Balasubramanian, Vineeth N. and Tsang, Ivor},
    volume = {157},
    series = {Proceedings of Machine Learning Research},
    month = {17--19 Nov},
    publisher = {PMLR},
    pdf = {https://proceedings.mlr.press/v157/demir21a/demir21a.pdf},
    url = {https://proceedings.mlr.press/v157/demir21a.html},
}
# ConEx

```

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

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

14 dicee

14.1 Submodules

dicee.__main__

dicee.abstracts

Classes

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

Module Contents

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

Abstract class for Trainer class for knowledge graph embedding models

Parameter

args
[str] ?

callbacks: list
?

```

attributes

callbacks

is_global_zero = True

global_rank = 0

local_rank = 0

strategy = None

on_fit_start (*args, **kwargs)
    A function to call callbacks before the training starts.

```

Parameter

```

    args

    kwargs

    rtype
        None

on_fit_end (*args, **kwargs)
    A function to call callbacks at the end of the training.

```

Parameter

```

    args

    kwargs

    rtype
        None

on_train_epoch_start (*args, **kwargs)
    A function to call callbacks at the start of an epoch.

```

Parameter

```

    args

    kwargs

    rtype
        None

on_train_epoch_end (*args, **kwargs)
    A function to call callbacks at the end of an epoch.

```

Parameter

```

    args

    kwargs

    rtype
        None

```

on_train_batch_end (*args, **kwargs)

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

Parameter

args

kwargs

rtype

None

static save_checkpoint (full_path: str, model) → None

A static function to save a model into disk

Parameter

full_path : str

model:

rtype

None

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

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

Parameter

path_of_pretrained_model_dir

[str] ?

construct_ensemble: boolean

?

model_name: str apply_semantic_constraint : boolean

construct_ensemble = False

apply_semantic_constraint = False

configs

get_eval_report () → dict

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

Parameters

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

Return type

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

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

Parameters

triples

set_model_train_mode () → None
Setting the model into training mode

Parameter

set_model_eval_mode () → None
Setting the model into eval mode

Parameter

property name

sample_entity (*n: int*) → List[str]

sample_relation (*n: int*) → List[str]

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

save () → None

get_entity_index (*x: str*)

get_relation_index (*x: str*)

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

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

pytorch tensor of triple score

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

get_entity_embeddings (*items: List[str]*)

Return embedding of an entity given its string representation

Parameter

items:

entities

get_relation_embeddings (*items: List[str]*)

Return embedding of a relation given its string representation

Parameter

items:

relations

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

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

parameters ()

class dicee.abstracts.**InteractiveQueryDecomposition**

t_norm (*tens_1: torch.Tensor, tens_2: torch.Tensor, tnorm: str = 'min'*) → torch.Tensor

tensor_t_norm (*subquery_scores: torch.FloatTensor, tnorm: str = 'min'*) → torch.FloatTensor

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

t_conorm (*tens_1: torch.Tensor, tens_2: torch.Tensor, tconorm: str = 'min'*) → torch.Tensor

negnorm (*tens_1: torch.Tensor, lambda_: float, neg_norm: str = 'standard'*) → torch.Tensor

class dicee.abstracts.**AbstractCallback**

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

Abstract class for Callback class for knowledge graph embedding models

Parameter

on_init_start (*args, **kwargs)

Parameter

trainer:

model:

rtype

None

on_init_end (*args, **kwargs)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_fit_start (trainer, model)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_train_epoch_end (*trainer, model*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype

None

on_train_batch_end (**args, **kwargs*)

Call at the end of each mini-batch during the training.

Parameter

trainer:

model:

rtype

None

on_fit_end (**args, **kwargs*)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

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

Bases: [*AbstractCallback*](#)

Abstract class for Callback class for knowledge graph embedding models

Parameter

num_epochs

path

sample_counter = 0

epoch_count = 0

alphas = None

on_fit_start (*trainer, model*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_fit_end (*trainer, model*)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

store_ensemble (*param_ensemble*) → None

class dicee.abstracts.**BaseInteractiveTrainKGE**

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

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

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

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

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

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

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

Trains the Literal Embeddings model using literal data.

Parameters

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

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

dicee.analyse_experiments

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

Classes

Experiment

Functions

get_default_arguments()

analyse(args)

Module Contents

`dicee.analyse_experiments.get_default_arguments()`

class `dicee.analyse_experiments.Experiment`

```

    model_name = []
    callbacks = []
    embedding_dim = []
    num_params = []
    num_epochs = []
    batch_size = []
    lr = []
    byte_pair_encoding = []
    aswa = []
    path_dataset_folder = []

```

```

full_storage_path = []

pq = []

train_mrr = []

train_h1 = []

train_h3 = []

train_h10 = []

val_mrr = []

val_h1 = []

val_h3 = []

val_h10 = []

test_mrr = []

test_h1 = []

test_h3 = []

test_h10 = []

runtime = []

normalization = []

scoring_technique = []

save_experiment(x)

to_df()

dicee.analyse_experiments.analyse(args)

```

`dicee.callbacks`

Classes

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

Functions

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

Module Contents

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

Bases: `dicee.abstracts.AbstractCallback`

Abstract class for Callback class for knowledge graph embedding models

Parameter

path

on_fit_end (*trainer, model*) → None

Store epoch loss

Parameter

trainer:

model:

rtype

None

class dicee.callbacks.**PrintCallback**

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

start_time

on_fit_start (*trainer, pl_module*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_fit_end (*trainer, pl_module*)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

on_train_batch_end (**args, **kwargs*)

Call at the end of each mini-batch during the training.

Parameter

trainer:

model:

rtype

None

on_train_epoch_end (**args, **kwargs*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype

None

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

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

every_x_epoch

max_epochs

epoch_counter = 0

path

on_train_batch_end(**args, **kwargs*)

Call at the end of each mini-batch during the training.

Parameter

trainer:

model:

rtype

None

on_fit_start(*trainer, pl_module*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_train_epoch_end(**args, **kwargs*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype

None

on_fit_end(**args, **kwargs*)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

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

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

Bases: `dicee.abstracts.AbstractCallback`

Abstract class for Callback class for knowledge graph embedding models

Parameter

data_module

kg

num_of_epochs = 0

unlabelled_size

batch_size

create_random_data ()

on_epoch_end (*trainer, model*)

`dicee.callbacks.estimate_q` (*eps*)

estimate rate of convergence q from sequence esp

`dicee.callbacks.compute_convergence` (*seq, i*)

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

Bases: `dicee.abstracts.AbstractCallback`

Adaptive stochastic weight averaging ASWE keeps track of the validation performance and update s the ensemble model accordingly.

path

num_epochs

initial_eval_setting = None

epoch_count = 0

alphas = []

val_aswa = -1

on_fit_end (*trainer, model*)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

static compute_mrr (*trainer, model*) → float

get_aswa_state_dict (*model*)

decide (*running_model_state_dict, ensemble_state_dict, val_running_model, mrr_updated_ensemble_model*)

Perform Hard Update, software or rejection

Parameters

- **running_model_state_dict**
- **ensemble_state_dict**
- **val_running_model**
- **mrr_updated_ensemble_model**

on_train_epoch_end (*trainer, model*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype

None

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

Bases: `dicee.abstracts.AbstractCallback`

Abstract class for Callback class for knowledge graph embedding models

Parameter

path

reports = []

epoch_ratio = None

epoch_counter = 0

on_fit_start (*trainer, model*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_fit_end (*trainer, model*)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

on_train_epoch_end (*trainer, model*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype

None

on_train_batch_end (**args, **kwargs*)

Call at the end of each mini-batch during the training.

Parameter

trainer:

model:

rtype

None

class dicee.callbacks.**KronE**

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

f = None

static batch_kronecker_product (*a, b*)

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

get_kronecker_triple_representation (*indexed_triple: torch.LongTensor*)

Get kronecker embeddings

on_fit_start (*trainer, model*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

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

Bases: `diccee.abstracts.AbstractCallback`

A callback for a three-Level Perturbation

Input Perturbation: During training an input *x* is perturbed by randomly replacing its element. In the context of knowledge graph embedding models, *x* can denote a triple, a tuple of an entity and a relation, or a tuple of two entities. A perturbation means that a component of *x* is randomly replaced by an entity or a relation.

Parameter Perturbation:

Output Perturbation:

level = 'input'

ratio = 0.0

method = None

scaler = None

frequency = None

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

Called when the train batch begins.

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

Bases: `diccee.abstracts.AbstractCallback`

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

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

experiment_dir

max_epochs

epoch_counter = 0

save_model_every_n_epoch = True

reports

```

n_epochs_eval_model = 'val_test'

default_eval_model = None

eval_epochs

on_fit_end(trainer, model)
    Called at the end of training. Saves final evaluation report.

on_train_epoch_end(trainer, model)
    Called at the end of each training epoch. Performs evaluation and checkpointing if scheduled.

```

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

Bases: `dicee.abstracts.AbstractCallback`

Callback for managing learning rate scheduling and model snapshots.

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

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

```

total_epochs

experiment_dir

snapshot_dir

batches_per_epoch = None

total_steps = None

cycle_length = None

warmup_steps = None

lr_lambda = None

scheduler = None

step_count = 0

snapshot_loss

on_train_start(trainer, model)
    Initialize training parameters and LR scheduler at start of training.

on_train_batch_end(trainer, model, outputs, batch, batch_idx)
    Step the LR scheduler and save model snapshot if needed after each batch.

on_fit_end(trainer, model)
    Call at the end of the training.

```

Parameter

trainer:

model:

rtype

None

```

class dicee.callbacks.SWA (swa_start_epoch, swa_c_epochs: int = 1, lr_init: float = 0.1,
                          swa_lr: float = 0.05, max_epochs: int = None)
    Bases: dicee.abstracts.AbstractCallback
    Stochastic Weight Averaging callbacks.

    swa_start_epoch

    swa_c_epochs = 1

    swa_lr = 0.05

    lr_init = 0.1

    max_epochs = None

    swa_model = None

    swa_n = 0

    current_epoch = -1

    static moving_average (swa_model, running_model, alpha)
        Update SWA model with moving average of current model.

    on_train_epoch_start (trainer, model)
        Update learning rate according to SWA schedule.

    on_train_epoch_end (trainer, model)
        Apply SWA averaging if conditions are met.

    on_fit_end (trainer, model)
        Replace main model with SWA model at the end of training.

```

dicee.config

Classes

<i>Namespace</i>	Simple object for storing attributes.
------------------	---------------------------------------

Module Contents

```

class dicee.config.Namespace (**kwargs)
    Bases: argparse.Namespace
    Simple object for storing attributes.
    Implements equality by attribute names and values, and provides a simple string representation.

    dataset_dir: str = None
        The path of a folder containing train.txt, and/or valid.txt and/or test.txt

    save_embeddings_as_csv: bool = False
        Embeddings of entities and relations are stored into CSV files to facilitate easy usage.

    storage_path: str = 'Experiments'
        A directory named with time of execution under -storage_path that contains related data about embeddings.

```

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

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

sparql_endpoint = None
An endpoint of a triple store.

model: str = 'Keci'
KGE model

optim: str = 'Adam'
Optimizer

embedding_dim: int = 64
Size of continuous vector representation of an entity/relation

num_epochs: int = 150
Number of pass over the training data

batch_size: int = 1024
Mini-batch size if it is None, an automatic batch finder technique applied

lr: float = 0.1
Learning rate

add_noise_rate: float = None
The ratio of added random triples into training dataset

gpus = None
Number GPUs to be used during training

callbacks
10}}

Type
Callbacks, e.g., {"PPE"

Type
{ "last_percent_to_consider"

backend: str = 'pandas'
Backend to read, process, and index input knowledge graph. pandas, polars and rdflib available

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

trainer: str = 'torchCPUTrainer'
Trainer for knowledge graph embedding model

scoring_technique: str = 'KvsAll'
Scoring technique for knowledge graph embedding models

neg_ratio: int = 0
Negative ratio for a true triple in NegSample training_technique

weight_decay: float = 0.0
Weight decay for all trainable params

normalization: **str** = 'None'
 LayerNorm, BatchNorm1d, or None

init_param: **str** = None
 xavier_normal or None

gradient_accumulation_steps: **int** = 0
 Not tested e

num_folds_for_cv: **int** = 0
 Number of folds for CV

eval_model: **str** = 'train_val_test'
 ["None", "train", "train_val", "train_val_test", "test"]

Type
 Evaluate trained model choices

save_model_at_every_epoch: **int** = None
 Not tested

label_smoothing_rate: **float** = 0.0

num_core: **int** = 0
 Number of CPUs to be used in the mini-batch loading process

random_seed: **int** = 0
 Random Seed

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

read_only_few: **int** = None
 Read only first few triples

pykeen_model_kwargs
 Additional keyword arguments for pykeen models

kernel_size: **int** = 3
 Size of a square kernel in a convolution operation

num_of_output_channels: **int** = 32
 Number of slices in the generated feature map by convolution.

p: **int** = 0
 P parameter of Clifford Embeddings

q: **int** = 1
 Q parameter of Clifford Embeddings

input_dropout_rate: **float** = 0.0
 Dropout rate on embeddings of input triples

hidden_dropout_rate: **float** = 0.0
 Dropout rate on hidden representations of input triples

feature_map_dropout_rate: **float** = 0.0
 Dropout rate on a feature map generated by a convolution operation

byte_pair_encoding: bool = False

Byte pair encoding

Type

WIP

adaptive_swa: bool = False

Adaptive stochastic weight averaging

swa: bool = False

Stochastic weight averaging

block_size: int = None

block size of LLM

continual_learning = None

Path of a pretrained model size of LLM

auto_batch_finding = False

A flag for using auto batch finding

eval_every_n_epochs: int = 0

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

save_every_n_epochs: bool = False

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

eval_at_epochs: list = None

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

n_epochs_eval_model: str = 'val_test'

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

adaptive_lr

“cca”}

Type

Adaptive learning rate parameters, e.g., ‘{“scheduler_name”

swa_start_epoch: int = None

Epoch at which to start applying stochastic weight averaging.

__iter__()

dicee.dataset_classes

Classes

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

Functions

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

Module Contents

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

Reload the files from disk to construct the Pytorch dataset

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

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

Bases: `torch.utils.data.Dataset`

An abstract class representing a Dataset.

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

Note

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

```
train_set
ordered_bpe_entities
num_bpe_entities
neg_ratio
num_datapoints
__len__()
__getitem__(idx)
collate_fn(batch_shaped_bpe_triples: List[Tuple[torch.Tensor, torch.Tensor]])
```

```
class dicee.dataset_classes.MultiLabelDataset (train_set: torch.LongTensor,
        train_indices_target: torch.LongTensor, target_dim: int,
        torch_ordered_shaped_bpe_entities: torch.LongTensor)
```

Bases: `torch.utils.data.Dataset`

An abstract class representing a `Dataset`.

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

Note

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

```
train_set
train_indices_target
target_dim
num_datapoints
torch_ordered_shaped_bpe_entities
collate_fn = None
__len__()
__getitem__(idx)
```

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

Bases: torch.utils.data.Dataset

Dataset for the 1vsALL training strategy

Parameters

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

Return type

torch.utils.data.Dataset

train_data

block_size = 8

num_of_data_points

collate_fn = None

__len__()

__getitem__(*idx*)

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

Bases: torch.utils.data.Dataset

Dataset for the 1vsALL training strategy

Parameters

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

Return type

torch.utils.data.Dataset

train_data

target_dim

collate_fn = None

__len__()

__getitem__(*idx*)

```
class dicee.dataset_classes.KvsAll (train_set_idx: numpy.ndarray, entity_idxes, relation_idxes, form,  

store=None, label_smoothing_rate: float = 0.0)
```

Bases: torch.utils.data.Dataset

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

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

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

Note

TODO

train_set_idx

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

entity_idxes

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

relation_idxes

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

self : torch.utils.data.Dataset

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

train_data = None

train_target = None

label_smoothing_rate

collate_fn = None

__len__()

__getitem__(idx)

```
class dicee.dataset_classes.AllvsAll (train_set_idx: numpy.ndarray, entity_idxes, relation_idxes,  

label_smoothing_rate=0.0)
```

Bases: torch.utils.data.Dataset

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

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

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

Note

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

train_set_idx

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

entity_idx

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

relation_idx

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

self : torch.utils.data.Dataset

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

train_data = None

train_target = None

label_smoothing_rate

collate_fn = None

target_dim

__len__()

__getitem__(idx)

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

Bases: torch.utils.data.Dataset

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

Parameters

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

train_data

The input data converted into a PyTorch tensor.

Type

torch.Tensor

num_entities

Number of entities in the dataset.

Type

int

num_relations

Number of relations in the dataset.

Type

int

neg_sample_ratio

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

Type

int

label_smoothing_rate

The smoothing factor applied to the labels.

Type

torch.Tensor

collate_fn

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

Type

function, optional

train_data

num_entities

num_relations

neg_sample_ratio = None

label_smoothing_rate

collate_fn = None

__len__()

Returns the number of samples in the dataset.

__getitem__(idx)

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

Parameters

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

Returns

A tuple consisting of:

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

Return type
tuple

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

Bases: torch.utils.data.Dataset

KvsSample a Dataset:

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

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

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

new_y! << **|E|** new_y contains all 1's if sum(y)< neg_sample_ratio new_y contains

train_set_idx
Indexed triples for the training.

entity_idxes
mapping.

relation_idxes
mapping.

form
?

store
?

label_smoothing_rate
?

torch.utils.data.Dataset

train_data = None

train_target = None

neg_ratio = None

num_entities

label_smoothing_rate

collate_fn = None

max_num_of_classes

__len__()

__getitem__ (idx)

```
class dicee.dataset_classes.NegSampleDataset (train_set: numpy.ndarray, num_entities: int,
      num_relations: int, neg_sample_ratio: int = 1)
```

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

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

Note

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

neg_sample_ratio

train_triples

length

num_entities

num_relations

labels

train_set = []

__len__()

__getitem__(idx)

```
class dicee.dataset_classes.TriplePredictionDataset(train_set: numpy.ndarray,
                                                    num_entities: int, num_relations: int, neg_sample_ratio: int = 1, label_smoothing_rate: float = 0.0)
```

Bases: `torch.utils.data.Dataset`

Triple Dataset

D:= {(x)_i}_i ^N, where

. x:(h,r, t) in KG is a unique h in E and a relation r in R and . collect_fn => Generates negative triples

collect_fn:

orall (h,r,t) in G obtain, create negative triples{(h,r,x),(r,t),(h,m,t)}

y:labels are represented in torch.float16

train_set_idx

Indexed triples for the training.

entity_idxxs

mapping.

relation_idxxs

mapping.

form

?

store

?

```

    label_smoothing_rate
    collate_fn: batch:List[torch.IntTensor] Returns —— torch.utils.data.Dataset
label_smoothing_rate
neg_sample_ratio
train_set
length
num_entities
num_relations
__len__ ()
__getitem__ (idx)
collate_fn (batch: List[torch.Tensor])
class dicee.dataset_classes.CVDDataModule (train_set_idx: numpy.ndarray, num_entities,
    num_relations, neg_sample_ratio, batch_size, num_workers)
Bases: pytorch_lightning.LightningDataModule
Create a Dataset for cross validation

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

Return type
    ?

train_set_idx
num_entities
num_relations
neg_sample_ratio
batch_size
num_workers

```

`train_dataloader()` → `torch.utils.data.DataLoader`

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:param-ref:~pytorch_lightning.trainer.trainer.Trainer.reload_dataloaders_every_n_epochs** to a positive integer.

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

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

Note

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

`setup(*args, **kwargs)`

Called at the beginning of `fit` (train + validate), `validate`, `test`, or `predict`. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

Parameters

stage – either 'fit', 'validate', 'test', or 'predict'

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

        # don't do this
        self.something = else

    def setup(self, stage):
        data = load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

`transfer_batch_to_device(*args, **kwargs)`

Override this hook if your `DataLoader` returns tensors wrapped in a custom data structure.

The data types listed below (and any arbitrary nesting of them) are supported out of the box:

- `torch.Tensor` or anything that implements `.to(...)`
- `list`
- `dict`
- `tuple`

For anything else, you need to define how the data is moved to the target device (CPU, GPU, TPU, ...).

Note

This hook should only transfer the data and not modify it, nor should it move the data to any other device than the one passed in as argument (unless you know what you are doing). To check the current state of execution of this hook you can use `self.trainer.training/testing/validating/predicting` so that you can add different logic as per your requirement.

Parameters

- **batch** – A batch of data that needs to be transferred to a new device.
- **device** – The target device as defined in PyTorch.
- **dataloader_idx** – The index of the dataloader to which the batch belongs.

Returns

A reference to the data on the new device.

Example:

```
def transfer_batch_to_device(self, batch, device, dataloader_idx):
    if isinstance(batch, CustomBatch):
        # move all tensors in your custom data structure to the device
        batch.samples = batch.samples.to(device)
        batch.targets = batch.targets.to(device)
    elif dataloader_idx == 0:
        # skip device transfer for the first dataloader or anything you wish
        pass
    else:
        batch = super().transfer_batch_to_device(batch, device, dataloader_
↪idx)
    return batch
```

See also

- `move_data_to_device()`
- `apply_to_collection()`

`prepare_data(*args, **kwargs)`

Use this to download and prepare data. Downloading and saving data with multiple processes (distributed settings) will result in corrupted data. Lightning ensures this method is called only within a single process, so you can safely add your downloading logic within.

 **Warning**

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

Example:

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

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

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

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

Example:

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

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

This is called before requesting the dataloaders:

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

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

Bases: `torch.utils.data.Dataset`

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

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

`train_file_path`

Path to the training data file.

Type

`str`

`normalization`

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

Type

`str`

`normalization_params`

Parameters used for normalization.

Type

`dict`

`sampling_ratio`

Fraction of the training set to use for ablations.

Type

`float`

`entity_to_idx`

Mapping of entities to their indices.

Type

`dict`

`num_entities`

Total number of entities.

Type

`int`

`data_property_to_idx`

Mapping of data properties to their indices.

Type

`dict`

`num_data_properties`

Total number of data properties.

Type

`int`

`loader_backend`

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

Type

`str`

```

train_file_path

loader_backend = 'pandas'

normalization_type = 'z-norm'

normalization_params

sampling_ratio = None

entity_to_idx = None

num_entities

__getitem__(index)

__len__()

static load_and_validate_literal_data (file_path: str = None, loader_backend: str = 'pandas')
    → pandas.DataFrame
    Loads and validates the literal data file. :param file_path: Path to the literal data file. :type file_path: str

Returns
    DataFrame containing the loaded and validated data.

Return type
    pd.DataFrame

static denormalize (preds_norm, attributes, normalization_params) → numpy.ndarray
    Denormalizes the predictions based on the normalization type.

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

Returns
    Denormalized predictions.

Return type
    np.ndarray

```

dicee.eval_static_funcs

Functions

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

Module Contents

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

Parameters

- **model**
- **triples**
- **er_vocab**
- **re_vocab**

```
dicee.eval_static_funcs.evaluate_link_prediction_performance_with_reciprocals(  
    model: dicee.knowledge_graph_embeddings.KGE, triples, er_vocab: Dict[Tuple, List])
```

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

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

Parameters

- **model**
- **triples**
- **within_entities**
- **er_vocab**
- **re_vocab**

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

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

Evaluates the trained literal prediction model on a test file.

Parameters

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

Returns

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

Return type

pd.DataFrame

Raises

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

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

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

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

Parameters

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

Returns

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

`dicee.evaluator`

Classes

<i>Evaluator</i>	Evaluator class to evaluate KGE models in various downstream tasks
------------------	--

Module Contents

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

Evaluator class to evaluate KGE models in various downstream tasks

Arguments

re_vocab = None

er_vocab = None

ee_vocab = None

func_triple_to_bpe_representation = None

is_continual_training = None

num_entities = None

num_relations = None

args

report

during_training = False

vocab_preparation (*dataset*) → None

A function to wait future objects for the attributes of executor

Return type

None

eval (*dataset: dicee.knowledge_graph.KG, trained_model, form_of_labelling, during_training=False*)
→ None

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

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

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

Evaluate model after reciprocal triples are added

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

Evaluate model after reciprocal triples are added

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

Evaluate model after reciprocal triples are added

evaluate_lp_k_vs_all (*model, triple_idx, info=None, form_of_labelling=None*)

Filtered link prediction evaluation. :param model: :param triple_idx: test triples :param info: :param form_of_labelling: :return:

evaluate_lp_with_byte (*model, triples: List[List[str]], info=None*)

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

Parameters

- **model**
- **triples** (*List of lists*)
- **info**
- **form_of_labelling**

evaluate_lp (*model, triple_idx, info: str*)

dummy_eval (*trained_model, form_of_labelling: str*)

eval_with_data (*dataset, trained_model, triple_idx: numpy.ndarray, form_of_labelling: str*)

dicee.executor

Classes

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

Module Contents

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

A class for Training, Retraining and Evaluation a model.

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

distributed

args

is_continual_training = **False**

trainer = **None**

trained_model = **None**

knowledge_graph = **None**

report

evaluator = **None**

start_time = **None**

is_rank_zero() → bool

cleanup()

setup_executor() → None

create_and_store_kg()

load_from_memmap()

save_trained_model() → None

Save a knowledge graph embedding model

- (1) Send model to eval mode and cpu.
- (2) Store the memory footprint of the model.
- (3) Save the model into disk.
- (4) Update the stats of KG again ?

Parameter

rtype

None

end (*form_of_labelling: str*) → dict

End training

- (1) Store trained model.
- (2) Report runtimes.
- (3) Eval model if required.

Parameter

rtype

A dict containing information about the training and/or evaluation

write_report () → None

Report training related information in a report.json file

start () → dict

Start training

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

Parameter

rtype

A dict containing information about the training and/or evaluation

class `dicee.executer.ContinuousExecute` (*args*)

Bases: *Execute*

A subclass of Execute Class for retraining

(1) Loading & Preprocessing & Serializing input data.

(2) Training & Validation & Testing

(3) Storing all necessary info

During the continual learning we can only modify * **num_epochs** * parameter. Trained model stored in the same folder as the seed model for the training. Trained model is noted with the current time.

continual_start () → dict

Start Continual Training

(1) Initialize training.

(2) Start continual training.

(3) Save trained model.

Parameter

rtype

A dict containing information about the training and/or evaluation

dicee.knowledge_graph

Classes

KG

Knowledge Graph

Module Contents

```

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

Knowledge Graph

```

dataset_dir = None

sparql_endpoint = None

path_single_kg = None

byte_pair_encoding = False

ordered_shaped_bpe_tokens = None

add_noise_rate = None

num_entities = None

num_relations = None

path_for_deserialization = None

add_reciprocal = None

eval_model = None

read_only_few = None

sample_triples_ratio = None

path_for_serialization = None

entity_to_idx = None

relation_to_idx = None

backend = 'pandas'

training_technique = None

idx_entity_to_bpe_shaped

enc

num_tokens

num_bpe_entities = None

padding = False

dummy_id

max_length_subword_tokens = None

train_set_target = None
```

```

target_dim = None

train_target_indices = None

ordered_bpe_entities = None

separator = None

description_of_input = None

describe() → None

property_entities_str: List

property_relations_str: List

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

__iter__()

__len__()

func_triple_to_bpe_representation(triple: List[str])

```

dicee.knowledge_graph_embeddings

Classes

<i>KGE</i>	Knowledge Graph Embedding Class for interactive usage of pre-trained models
------------	---

Module Contents

```

class dicee.knowledge_graph_embeddings.KGE(path=None, url=None, construct_ensemble=False,
    model_name=None)
    Bases: dicee.abstracts.BaseInteractiveKGE, dicee.abstracts.InteractiveQueryDecomposition, dicee.abstracts.BaseInteractiveTrainKGE
    Knowledge Graph Embedding Class for interactive usage of pre-trained models
    __str__()
    to(device: str) → None

    get_transductive_entity_embeddings(indices: torch.LongTensor | List[str], as_pytorch=False,
        as_numpy=False, as_list=True) → torch.FloatTensor | numpy.ndarray | List[float]

    create_vector_database(collection_name: str, distance: str, location: str = 'localhost',
        port: int = 6333)

    generate(h="", r="")

    eval_lp_performance(dataset=List[Tuple[str, str, str]], filtered=True)

    predict_missing_head_entity(relation: List[str] | str, tail_entity: List[str] | str, within=None,
        batch_size=2, topk=1, return_indices=False) → Tuple
        Given a relation and a tail entity, return top k ranked head entity.
         $\operatorname{argmax}_{\{e \in E\}} f(e, r, t)$ , where  $r \in R$ ,  $t \in E$ .

```

Parameter

relation: Union[List[str], str]

String representation of selected relations.

tail_entity: Union[List[str], str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_relations (*head_entity: List[str] | str, tail_entity: List[str] | str, within=None, batch_size=2, topk=1, return_indices=False*) → Tuple

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

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

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_tail_entity (*head_entity: List[str] | str, relation: List[str] | str, within: List[str] = None, batch_size=2, topk=1, return_indices=False*) → torch.FloatTensor

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

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

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

scores

predict (*, *h*: List[str] | str = None, *r*: List[str] | str = None, *t*: List[str] | str = None, *within*=None, *logits*=True) → torch.FloatTensor

Parameters

- **logits**
- **h**
- **r**
- **t**
- **within**

predict_topk (*, *h*: str | List[str] = None, *r*: str | List[str] = None, *t*: str | List[str] = None, *topk*: int = 10, *within*: List[str] = None, *batch_size*: int = 1024)

Predict missing item in a given triple.

Returns

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

triple_score (*h*: List[str] | str = None, *r*: List[str] | str = None, *t*: List[str] | str = None, *logits*=False) → torch.FloatTensor

Predict triple score

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

Returns: Tuple

pytorch tensor of triple score

return_multi_hop_query_results (*aggregated_query_for_all_entities*, *k*: int, *only_scores*)

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

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

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

Find an answer set for EPFO queries including negation and disjunction

Parameter

query_type: str The type of the query, e.g., “2p”.

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

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

tnorm: str The t-norm operator.

neg_norm: str The negation norm.

lambda_: float lambda parameter for sugeno and yager negation norms

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

returns

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

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

Find missing triples

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

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

Return (e,r,x)

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

confidence: float

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

topk: int

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

at_most: int

Stop after finding at_most missing triples

{(e,r,x) | f(e,r,x) > confidence land (e,r,x)}

otin G

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

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

Predicts literal values for given entities and attributes.

Parameters

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

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

Returns

Predictions for the given entities and attributes.

Return type

numpy ndarray

dicee.models

Submodules

dicee.models.adopt

ADOPT Optimizer Implementation.

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

ADOPT Overview:

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

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

Algorithm Comparison:

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

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

Classes:

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

Functions:

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

Performance:

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

Example:

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

References:

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

Notes:

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

Classes

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

Functions

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

Module Contents

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

Bases: `torch.optim.optimizer.Optimizer`

ADOPT Optimizer.

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

The algorithm performs the following key operations: 1. Normalizes gradients by the square root of the second moment estimate 2. Applies optional gradient clipping based on the training step 3. Updates parameters using

momentum-smoothed normalized gradients 4. Supports decoupled weight decay (AdamW-style) or L2 regularization

Mathematical formulation:

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

where:

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

Reference:

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

Parameters

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

Raises

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

Example

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

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

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

Note

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

clip_lambda

__setstate__(state)

Restore optimizer state from a checkpoint.

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

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

Parameters

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

Note

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

step (*closure=None*)

Perform a single optimization step.

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

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

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

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

Parameters

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

Returns

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

Return type

Optional[Tensor]

Example

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

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

Note

- Call zero_grad() before computing gradients for the next step

- CUDA graph capture is checked for safety when capturable=True
- The method is thread-safe for different parameter groups

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

Functional API that performs ADOPT algorithm computation.

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

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

Algorithm overview (ADOPT):

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

Mathematical formulation:

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

where:

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

Automatic mode selection:

When foreach and fused are both None (default), the function automatically selects the best implementation based on:

- Parameter types and devices
- Whether differentiable mode is enabled
- Learning rate type (float vs Tensor)
- Capturable mode requirements

param params

Parameters to optimize.

type params

List[Tensor]

param grads

Gradients for each parameter.

type grads

List[Tensor]

param exp_avgs

First moment estimates (momentum).

type exp_avgs

List[Tensor]

param exp_avg_sqs

Second moment estimates (variance).

type exp_avg_sqs

List[Tensor]

param state_steps

Step counters (must be singleton tensors).

type state_steps

List[Tensor]

param foreach

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

type foreach

Optional[bool]

param capturable

If True, ensure CUDA graph capture safety.

type capturable

bool

param differentiable

If True, allow gradients through optimization step.

type differentiable

bool

param fused

If True, use fused kernels (not implemented).

type fused

Optional[bool]

param grad_scale

Gradient scaler for AMP training.

type grad_scale
Optional[Tensor]

param found_inf
Flag for inf/nan detection in AMP.

type found_inf
Optional[Tensor]

param has_complex
Whether any parameters are complex-valued.

type has_complex
bool

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

type beta1
float

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

type beta2
float

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

type lr
Union[float, Tensor]

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

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

param weight_decay
Weight decay coefficient (L2 penalty).

type weight_decay
float

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

type decouple
bool

param eps
Small constant for numerical stability in normalization.

type eps
float

param maximize
If True, maximize objective instead of minimize.

type maximize
bool

raises RuntimeError

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

raises RuntimeError

If state_steps contains non-tensor elements.

raises RuntimeError

If fused=True (not yet implemented).

raises RuntimeError

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

Example

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

Note

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

See also

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

dicee.models.base_model

Classes

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

Module Contents

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

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

training_step_outputs = []

mem_of_model() → Dict

Size of model in MB and number of params

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

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

Parameters

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

Returns

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

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

Example:

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

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

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

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

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

Note

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

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

Parameters

- **yhat_batch**

- `y_batch`

`on_train_epoch_end(*args, **kwargs)`

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

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

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

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

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

`test_epoch_end(outputs: List[Any])`

`test_dataloader() → None`

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

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

Note

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

Note

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

`val_dataloader()` → None

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

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

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

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

Note

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

Note

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

`predict_dataloader()` → None

An iterable or collection of iterables specifying prediction samples.

For more information about multiple dataloaders, see this section.

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

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

Note

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

Returns

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

`train_dataloader()` → None

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

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

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

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

Note

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

`configure_optimizers` (*parameters=None*)

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

Returns

Any of these 6 options.

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

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

```

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

```

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

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

Note

Some things to know:

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

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

```

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

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

forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors

    Parameters
    -----

init_params_with_sanity_checking()

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

    Parameters

        •  $\mathbf{x}$ 

        •  $\mathbf{y\_idx}$ 

        • ordered_bpe_entities

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

    Parameters

         $\mathbf{x}$ 

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

```

```

get_triple_representation(idx_hrt)

get_head_relation_representation(indexed_triple)

get_sentence_representation(x: torch.LongTensor)

```

Parameters

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

```

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

```

Parameters

x ($B \times 2 \times T$)

```

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

```

```

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

```

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

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

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

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

```

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

Note

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

Variables

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

```

args = None

__call__(x)

static forward(x)

```

dicee.models.clifford

Classes

<i>Keci</i>	Base class for all neural network modules.
<i>CKeci</i>	Without learning dimension scaling
<i>DeCaL</i>	Base class for all neural network modules.

Module Contents

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

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

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

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

```

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

Note

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

Variables

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

```
name = 'Keci'
```

p

q

r

requires_grad_for_interactions = True

compute_sigma_pp(hp, rp)

Compute $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_{i,r,k} - h_{k,r,i}) e_i e_k$

σ_{pp} captures the interactions between along p bases For instance, let p e₁, e₂, e₃, we compute interactions between e₁ e₂, e₁ e₃, and e₂ e₃ This can be implemented with a nested two for loops

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

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

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

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

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

e₂e₁, e₂e₂, e₂e₃, e₃e₁, e₃e₂, e₃e₃

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

compute_sigma_qq(hq, rq)

Compute $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_{j,r,k} - h_{k,r,j}) e_j e_k$ σ_{qq} captures the interactions between along q bases For instance, let q e₁, e₂, e₃, we compute interactions between e₁ e₂, e₁ e₃, and e₂ e₃ This can be implemented with a nested two for loops

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

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

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

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

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

e₂e₁, e₂e₂, e₂e₃, e₃e₁, e₃e₂, e₃e₃

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

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

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

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

for j in range(q):

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

print(sigma_pq.shape)

apply_coefficients(hp, hq, rp, rq)

Multiplying a base vector with its scalar coefficient

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

Compute our CL multiplication

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

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

eq j

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

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

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

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

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

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

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

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

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

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

Parameter

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

returns

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

forward_k_vs_with_explicit (*x: torch.Tensor*)

k_vs_all_score (*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

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

Kvsall training

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

forward_k_vs_with_explicit and this functions are identical Parameter ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, **IEI**) shape

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

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

Construct a batch of batches 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.clifford.CKeci` ($args$)

Bases: `Keci`

Without learning dimension scaling

name = 'CKeci'

requires_grad_for_interactions = False

class `dicee.models.clifford.DeCaL` ($args$)

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

name = 'DeCaL'

entity_embeddings

relation_embeddings

p

q

r

re

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

Parameter

x: torch.LongTensor with (n,) shape

rtype

torch.FloatTensor with (n) shape

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

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

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

compute_sigmas_single (*list_h_emb, list_r_emb, list_t_emb*)

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

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

and return:

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

compute_sigmas_multivect (*list_h_emb, list_r_emb*)

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

For same bases vectors interaction we have

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

For different base vector interactions, we have

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

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

Kvsall training

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

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

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

Multiplying a base vector with its scalar coefficient

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

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

Parameter

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

returns

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

compute_sigma_pp (*hp, rp*)

Compute .. math:

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

compute_sigma_qq(hq, rq)

 Compute

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

compute_sigma_rr(hk, rk)

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

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

 Compute

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

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

for j in range(q):

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

print(sigma_pq.shape)

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

 Compute

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

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

```

        for j in range(q):
            sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
    print(sigma_pq.shape)
compute_sigma_qr(*, hq, hk, rq, rk)

```

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

```

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

```

dicee.models.complex

Classes

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

Module Contents

```

class dicee.models.complex.ConEx(args)
    Bases: dicee.models.base_model.BaseKGE
    Convolutional ComplEx Knowledge Graph Embeddings
    name = 'ConEx'
    conv2d
    fc_num_input
    fc1
    norm_fc1
    bn_conv2d
    feature_map_dropout
    residual_convolution(C_1: Tuple[torch.Tensor, torch.Tensor],
                        C_2: Tuple[torch.Tensor, torch.Tensor]) → torch.FloatTensor
        Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors
        that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds
        complex-valued embeddings :return:
    forward_k_vs_all(x: torch.Tensor) → torch.FloatTensor

```

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

Parameters

x

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

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional ComplEx Knowledge Graph Embeddings

name = 'AConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

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

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

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

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

Parameters

x

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

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

(continues on next page)

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

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

Note

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

Variables

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

name = 'Complex'

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

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

Parameters

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

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

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

dicee.models.dualE

Classes

DualE

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

Module Contents

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

Bases: `dicee.models.base_model.BaseKGE`

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

name = 'DualE'

entity_embeddings

relation_embeddings

num_ent = None

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

KvsAll scoring function

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

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

Negative Sampling forward pass:

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_k_vs_all(*x*)

KvsAll forward pass

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

T(*x: torch.tensor*) → torch.tensor

Transpose function

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

dicee.models.ensemble

Classes

EnsembleKGE

Module Contents

```
class dicee.models.ensemble.EnsembleKGE (models: list = None, seed_model=None,  
    pretrained_models: List = None)  
  
    name  
  
    train_mode = True  
  
    args  
  
    named_children()  
  
    property example_input_array  
  
    parameters()  
  
    modules()  
  
    __iter__()  
  
    __len__()  
  
    eval()  
  
    to (device)  
  
    state_dict()  
        Return the state dict of the ensemble.  
  
    load_state_dict (state_dict, strict=True)  
        Load the state dict into the ensemble.  
  
    mem_of_model()  
  
    __call__ (x_batch)  
  
    step()  
  
    get_embeddings()  
  
    __str__()
```

dicee.models.function_space

Classes

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

Module Contents

```
class dicee.models.function_space.FMult (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    name = 'FMult'
    entity_embeddings
    relation_embeddings
    k
    num_sample = 50
    gamma
    roots
    weights
    compute_func (weights: torch.FloatTensor, x) → torch.FloatTensor
    chain_func (weights, x: torch.FloatTensor)
    forward_triples (idx_triple: torch.Tensor) → torch.Tensor
```

Parameters

x

```
class dicee.models.function_space.GFMult (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    name = 'GFMult'
    entity_embeddings
    relation_embeddings
    k
    num_sample = 250
    roots
    weights
    compute_func (weights: torch.FloatTensor, x) → torch.FloatTensor
    chain_func (weights, x: torch.FloatTensor)
    forward_triples (idx_triple: torch.Tensor) → torch.Tensor
```

Parameters

x

```

class dicee.models.function_space.FMult2(args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    name = 'FMult2'
    n_layers = 3
    k
    n = 50
    score_func = 'compositional'
    discrete_points
    entity_embeddings
    relation_embeddings
    build_func(Vec)
    build_chain_funcs(list_Vec)
    compute_func(W, b, x) → torch.FloatTensor
    function(list_W, list_b)
    trapezoid(list_W, list_b)
    forward_triples(idx_triple: torch.Tensor) → torch.Tensor

```

Parameters

x

```

class dicee.models.function_space.LFMult1(args)

```

Bases: *dicee.models.base_model.BaseKGE*

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

```

name = 'LFMult1'
entity_embeddings
relation_embeddings
forward_triples(idx_triple)

```

Parameters

x

```

tri_score(h, r, t)
vtp_score(h, r, t)

```

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

Bases: *dicee.models.base_model.BaseKGE*

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

```
name = 'LFMult'
```

```
entity_embeddings
```

```
relation_embeddings
```

```
degree
```

```
m
```

```
x_values
```

```
forward_triples(idx_triple)
```

Parameters

x

```
construct_multi_coeff(x)
```

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

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

```
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:

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

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

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

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

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

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

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

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

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

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

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

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

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

dicee.models.literal

Classes

LiteralEmbeddings

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

Module Contents

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

Bases: `torch.nn.Module`

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

num_of_data_properties

Number of data properties (attributes).

Type

`int`

embedding_dims

Dimension of the embeddings.

Type

`int`

entity_embeddings

Pre-trained entity embeddings.

Type

`torch.tensor`

dropout

Dropout rate for regularization.

Type

`float`

gate_residual

Whether to use gated residual connections.

Type
bool

freeze_entity_embeddings
Whether to freeze the entity embeddings during training.

Type
bool

embedding_dim

num_of_data_properties

hidden_dim

gate_residual = True

freeze_entity_embeddings = True

entity_embeddings

data_property_embeddings

fc

fc_out

dropout

gated_residual_proj

layer_norm

forward (*entity_idx*, *attr_idx*)

Parameters

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

Returns
scalar predictions.

Return type
Tensor

property device

dicee.models.octonion

Classes

<i>OMult</i>	Base class for all neural network modules.
<i>ConvO</i>	Base class for all neural network modules.
<i>AConvO</i>	Additive Convolutional Octonion Knowledge Graph Em- beddings

Functions

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

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

Module Contents

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

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

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

```
name = '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.,
 $[score(h,r,x)|x \text{ in Entities}] \Rightarrow [0.0,0.1,...,0.8]$, shape=> (1, **Entities**) Given a batch of head entities and
relations => shape (size of batch,l Entities)

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

```
name = 'ConvO'
```

```
conv2d
```

```
fc_num_input
```

```
fc1
```

```
bn_conv2d
```

```
norm_fc1
```

```

feature_map_dropout

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

residual_convolution(O_1, O_2)

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

    Parameters
    x

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

class dicee.models.octonion.AConvO(args: dict)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional Octonion Knowledge Graph Embeddings
    name = 'AConvO'
    conv2d
    fc_num_input
    fc1
    bn_conv2d
    norm_fc1
    feature_map_dropout

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

residual_convolution(O_1, O_2)

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

    Parameters
    x

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

```

dicee.models.pykeen_models

Classes

PykeenKGE

A class for using knowledge graph embedding models implemented in Pykeen

Module Contents

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

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

A class for using knowledge graph embedding models implemented in Pykeen

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

```
model_kwargs
```

```
name
```

```
model
```

```
loss_history = []
```

```
args
```

```
entity_embeddings = None
```

```
relation_embeddings = None
```

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

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

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

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

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

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

```
    else:
```

```
        t = self.entity_embeddings.weight
```

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

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

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

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

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

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

```
abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)
```

dicее.models.quaternion

Classes

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

Functions

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

Module Contents

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

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

Bases: *dicее.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

name = 'QMult'

explicit = True

quaternion_multiplication_followed_by_inner_product (*h, r, t*)

Parameters

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

Returns

Triple scores.

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

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

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

Parameters

x – The vector.

Returns

The normalized vector.

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

k_vs_all_score (*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

Parameters

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

forward_k_vs_all (*x*)

Parameters

x

forward_k_vs_sample (*x, target_entity_idx*)

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

```

class dicee.models.quaternion.ConvQ(args)
    Bases: dicee.models.base_model.BaseKGE
    Convolutional Quaternion Knowledge Graph Embeddings
    name = 'ConvQ'
    entity_embeddings
    relation_embeddings
    conv2d
    fc_num_input
    fc1
    bn_conv1
    bn_conv2
    feature_map_dropout
    residual_convolution(Q_1, Q_2)
    forward_triples(indexed_triple: torch.Tensor) → torch.Tensor

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

class dicee.models.quaternion.AConvQ(args)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional Quaternion Knowledge Graph Embeddings
    name = 'AConvQ'
    entity_embeddings
    relation_embeddings
    conv2d
    fc_num_input
    fc1
    bn_conv1
    bn_conv2
    feature_map_dropout
    residual_convolution(Q_1, Q_2)

```

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

Parameters

x

forward_k_vs_all (*x: torch.Tensor*)

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

dicee.models.real

Classes

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

Module Contents

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

Bases: *dicee.models.base_model.BaseKGE*

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

name = 'DistMult'

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

Parameters

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

forward_k_vs_all (*x: torch.LongTensor*)

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

score (*h, r, t*)

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

Bases: *dicee.models.base_model.BaseKGE*

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

name = 'TransE'

```

margin = 4

score(head_ent_emb, rel_ent_emb, tail_ent_emb)

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

class dicee.models.real.Shallom(args)
    Bases: dicee.models.base_model.BaseKGE
    A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
    name = 'Shallom'

    shallom

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

    forward_k_vs_all(x) → torch.FloatTensor

    forward_triples(x) → torch.FloatTensor

        Parameters
        x

        Returns

class dicee.models.real.Pyke(args)
    Bases: dicee.models.base_model.BaseKGE
    A Physical Embedding Model for Knowledge Graphs
    name = 'Pyke'

    dist_func

    margin = 1.0

    forward_triples(x: torch.LongTensor)

        Parameters
        x

class dicee.models.real.CoKEConfig

    block_size: int = 3

    vocab_size: int = None

    n_layer: int = 6

    n_head: int = 8

    n_embd: int = None

    dropout: float = 0.3

    bias: bool = True

    causal: bool = False

```

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

`name = 'CoKE'`

`config`

`pos_emb`

`mask_emb`

`blocks`

`ln_f`

`coke_dropout`

`forward_k_vs_all (x: torch.Tensor)`

`score (emb_h, emb_r, emb_t)`

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

dicee.models.static_funcs

Functions

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

Module Contents

```
dicee.models.static_funcs.quaternion_mul(*, Q_1, Q_2)
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor]
Perform quaternion multiplication :param Q_1: :param Q_2: :return:
```

dicee.models.transformers

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

Classes

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

Module Contents

```
class dicee.models.transformers.ByteE(*args, **kwargs)
```

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

(continues on next page)

(continued from previous page)

```
self.conv2 = nn.Conv2d(20, 20, 5)

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

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

Note

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

Variables

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

name = 'Byte'

config

temperature = 0.5

topk = 2

transformer

lm_head

loss_function (*yhat_batch, y_batch*)

Parameters

- **yhat_batch**
- **y_batch**

forward (*x: torch.LongTensor*)

Parameters

x (*B by T tensor*)

generate (*idx, max_new_tokens, temperature=1.0, top_k=None*)

Take a conditioning sequence of indices `idx` (LongTensor of shape (b,t)) and complete the sequence `max_new_tokens` times, feeding the predictions back into the model each time. Most likely you'll want to make sure to be in `model.eval()` mode of operation for this.

training_step (*batch, batch_idx=None*)

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

Parameters

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

Returns

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

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

Example:

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

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

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

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

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

Note

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

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

Bases: `torch.nn.Module`

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

weight

bias

forward (*input*)

```
class dicee.models.transformers.SelfAttention(config)
```

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

c_attn

c_proj

attn_dropout

resid_dropout

n_head

n_embd

dropout

causal

flash = **True**

forward (*x*)

```
class dicee.models.transformers.MLP (config)
```

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

c_fc

gelu

c_proj

dropout

forward (*x*)

```
class dicee.models.transformers.Block (config)
```

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

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

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

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

```

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

Note

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

Variables

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

`ln_1`

`attn`

`ln_2`

`mlp`

`forward(x)`

```
class dicee.models.transformers.GPTConfig
```

`block_size: int = 1024`

`vocab_size: int = 50304`

`n_layer: int = 12`

`n_head: int = 12`

`n_embd: int = 768`

`dropout: float = 0.0`

`bias: bool = False`

`causal: bool = True`

```
class dicee.models.transformers.GPT(config)
```

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

config

transformer

lm_head

get_num_params (*non_embedding=True*)

Return the number of parameters in the model. For non-embedding count (default), the position embeddings get subtracted. The token embeddings would too, except due to the parameter sharing these params are actually used as weights in the final layer, so we include them.

forward (*idx, targets=None*)

crop_block_size (*block_size*)

classmethod from_pretrained (*model_type, override_args=None*)

configure_optimizers (*weight_decay, learning_rate, betas, device_type*)

estimate_mfu (*fwdbwd_per_iter, dt*)

estimate model flops utilization (MFU) in units of A100 bfloat16 peak FLOPS

Classes

<i>ADOPT</i>	ADOPT Optimizer.
<i>BaseKGELightning</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>IdentityClass</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>Block</i>	Base class for all neural network modules.
<i>DistMult</i>	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
<i>TransE</i>	Translating Embeddings for Modeling
<i>Shallom</i>	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
<i>Pyke</i>	A Physical Embedding Model for Knowledge Graphs
<i>CoKEConfig</i>	
<i>CoKE</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>ConEx</i>	Convolutional ComplEx Knowledge Graph Embeddings
<i>AConEx</i>	Additive Convolutional ComplEx Knowledge Graph Embeddings
<i>Complex</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>IdentityClass</i>	Base class for all neural network modules.
<i>QMult</i>	Base class for all neural network modules.
<i>ConvQ</i>	Convolutional Quaternion Knowledge Graph Embeddings
<i>AConvQ</i>	Additive Convolutional Quaternion Knowledge Graph Embeddings
<i>BaseKGE</i>	Base class for all neural network modules.
<i>IdentityClass</i>	Base class for all neural network modules.
<i>OMult</i>	Base class for all neural network modules.
<i>ConvO</i>	Base class for all neural network modules.
<i>AConvO</i>	Additive Convolutional Octonion Knowledge Graph Embeddings
<i>Keci</i>	Base class for all neural network modules.
<i>CKeci</i>	Without learning dimension scaling
<i>DeCaL</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>PykeenKGE</i>	A class for using knowledge graph embedding models implemented in Pykeen
<i>BaseKGE</i>	Base class for all neural network modules.
<i>FMult</i>	Learning Knowledge Neural Graphs
<i>GFMult</i>	Learning Knowledge Neural Graphs
<i>FMult2</i>	Learning Knowledge Neural Graphs
<i>LFMult1</i>	Embedding with trigonometric functions. We represent all entities and relations in the complex number space as:
<i>LFMult</i>	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
<i>DualE</i>	Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)

Functions

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

Package Contents

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

Bases: torch.optim.optimizer.Optimizer

ADOPT Optimizer.

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

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

Mathematical formulation:

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

where:

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

Reference:

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

Parameters

- **params** (*ParamsT*) – Iterable of parameters to optimize or dicts defining parameter groups.
- **lr** (*float or Tensor, optional*) – Learning rate. Can be a float or 1-element Tensor. Default: 1e-3

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

Raises

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

Example

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

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

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

Note

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

`clip_lambda`

`__setstate__(state)`

Restore optimizer state from a checkpoint.

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

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

Parameters

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

Note

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

`step(closure=None)`

Perform a single optimization step.

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

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

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

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

Parameters

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

Returns

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

Return type

Optional[Tensor]

Example

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

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

Note

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

```
class dicee.models.BaseKGE Lightning(*args, **kwargs)
```

Bases: `lightning.LightningModule`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

training_step_outputs = []

mem_of_model() → Dict

Size of model in MB and number of params

training_step (*batch, batch_idx=None*)

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

Parameters

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

Returns

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

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

Example:

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

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

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

# Multiple optimizers (e.g.: GANs)
```

(continues on next page)

(continued from previous page)

```
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

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

Note

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

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

Parameters

- **yhat_batch**
- **y_batch**

on_train_epoch_end (*args, **kwargs)

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

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

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

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

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

test_epoch_end (outputs: *List[Any]*)

test_dataloader() → `None`

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

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

Note

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

Note

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

`val_dataloader()` → None

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

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

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

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

Note

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

Note

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

`predict_dataloader()` → None

An iterable or collection of iterables specifying prediction samples.

For more information about multiple dataloaders, see this section.

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

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

Note

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

Returns

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

`train_dataloader()` → None

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

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

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

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

Note

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

configure_optimizers (*parameters=None*)

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

Returns

Any of these 6 options.

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

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

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

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

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

Note

Some things to know:

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

class `dictee.models.BaseKGE` (*args: dict*)

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

args

embedding_dim = None

num_entities = None

```

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

```

Parameters

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

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

byte pair encoded neural link predictors

Parameters

```
init_params_with_sanity_checking()
```

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

Parameters

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

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

Parameters

x

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

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

```
get_triple_representation(idx_hrt)
```

```
get_head_relation_representation(indexed_triple)
```

```
get_sentence_representation(x: torch.LongTensor)
```

Parameters

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

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

Parameters

x ($B \times 2 \times T$)

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

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

(continues on next page)

(continued from previous page)

```
self.conv2 = nn.Conv2d(20, 20, 5)

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

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

Note

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

Variables

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

args = `None`

__call__ (*x*)

static forward (*x*)

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

`loss`

`selected_optimizer = None`

`normalizer_class = None`

`normalize_head_entity_embeddings`

`normalize_relation_embeddings`

`normalize_tail_entity_embeddings`

`hidden_normalizer`

`param_init`

`input_dp_ent_real`

`input_dp_rel_real`

`hidden_dropout`

`loss_history = []`

`byte_pair_encoding`

```

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

    Parameters
        x ( $B \times 2 \times T$ )

forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors

    Parameters
        -----

init_params_with_sanity_checking ()

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

    Parameters
        • x
        • y_idx
        • ordered_bpe_entities

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

    Parameters
        x

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

get_sentence_representation (x: torch.LongTensor)

    Parameters
        • (b ( $x$  shape)
        • 3
        • t)

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

    Parameters
        x ( $B \times 2 \times T$ )

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

```

```
class dicee.models.Block(config)
```

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

ln_1

attn

ln_2

mlp

forward (*x*)

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

Bases: *dicee.models.base_model.BaseKGE*

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

name = 'DistMult'

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

Parameters

- **emb_h**

```

        • emb_r
        • emb_E

forward_k_vs_all (x: torch.LongTensor)

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

score (h, r, t)

class dicee.models.TransE (args)
    Bases: dicee.models.base_model.BaseKGE
    Translating Embeddings for Modeling Multi-relational Data https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf
    name = 'TransE'
    margin = 4
    score (head_ent_emb, rel_ent_emb, tail_ent_emb)
    forward_k_vs_all (x: torch.Tensor) → torch.FloatTensor

class dicee.models.Shallom (args)
    Bases: dicee.models.base_model.BaseKGE
    A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
    name = 'Shallom'
    shallom
    get_embeddings () → Tuple[numpy.ndarray, None]
    forward_k_vs_all (x) → torch.FloatTensor
    forward_triples (x) → torch.FloatTensor

    Parameters
    x

    Returns

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

    Parameters
    x

class dicee.models.CoKEConfig

```

```

block_size: int = 3

vocab_size: int = None

n_layer: int = 6

n_head: int = 8

n_embd: int = None

dropout: float = 0.3

bias: bool = True

causal: bool = False

```

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

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

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

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

```

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

Note

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

Variables

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

`name = 'CoKE'`

`config`

`pos_emb`

```

mask_emb

blocks

ln_f

coke_dropout

forward_k_vs_all (x: torch.Tensor)

score (emb_h, emb_r, emb_t)

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

```

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

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

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

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

```

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

Note

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

Variables

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

args

embedding_dim = None

num_entities = None

num_relations = None

```

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

```

Parameters

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

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

byte pair encoded neural link predictors

Parameters

```

init_params_with_sanity_checking()

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

```

Parameters

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

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

Parameters

x

```

forward_k_vs_all(*args, **kwargs)

forward_k_vs_sample(*args, **kwargs)

get_triple_representation(idx_hrt)

get_head_relation_representation(indexed_triple)

get_sentence_representation(x: torch.LongTensor)

```

Parameters

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

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

Parameters

x ($B \times 2 \times T$)

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

```

class dicee.models.ConEx(args)
    Bases: dicee.models.base_model.BaseKGE
    Convolutional ComplEx Knowledge Graph Embeddings

    name = 'ConEx'

    conv2d

    fc_num_input

    fc1

    norm_fc1

    bn_conv2d

    feature_map_dropout

```

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

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

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

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

Parameters

x

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

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional ComplEx Knowledge Graph Embeddings

```
name = 'AConEx'
```

```
conv2d
```

```
fc_num_input
```

```
fc1
```

```
norm_fc1
```

```
bn_conv2d
```

```
feature_map_dropout
```

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

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

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

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

Parameters

x

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

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

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

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

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

```

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

Note

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

Variables

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

`name = 'Complex'`

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

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

Parameters

- `emb_h`
- `emb_r`
- `emb_E`

`forward_k_vs_all (x: torch.LongTensor) -> torch.FloatTensor`

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

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

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

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

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

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

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

```

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

Note

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

Variables

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

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

`loss`

```

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

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

forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors

    Parameters
        -----

init_params_with_sanity_checking ()

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

    Parameters
        •  $\mathbf{x}$ 
        •  $\mathbf{y\_idx}$ 
        • ordered_bpe_entities

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

    Parameters
         $\mathbf{x}$ 

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

```

`get_head_relation_representation(indexed_triple)`

`get_sentence_representation(x: torch.LongTensor)`

Parameters

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

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

Parameters

x ($B \times 2 \times T$)

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

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

args = None

```
__call__(x)
```

```
static forward(x)
```

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

```
class dicee.models.QMult(args)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

```
name = 'QMult'
```

```
explicit = True
```

```
quaternion_multiplication_followed_by_inner_product(h, r, t)
```

Parameters

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

Returns

Triple scores.

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

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

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

Parameters

x – The vector.

Returns

The normalized vector.

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

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

Parameters

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

forward_k_vs_all (*x*)

Parameters

x

forward_k_vs_sample (*x*, *target_entity_idx*)

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

class dicee.models.ConvQ (*args*)

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings

name = 'ConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

```

feature_map_dropout

residual_convolution(Q_1, Q_2)

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

    Parameters
    x

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

class dicee.models.AConvQ(args)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional Quaternion Knowledge Graph Embeddings
    name = 'AConvQ'
    entity_embeddings
    relation_embeddings
    conv2d
    fc_num_input
    fc1
    bn_conv1
    bn_conv2
    feature_map_dropout
    residual_convolution(Q_1, Q_2)
    forward_triples(indexed_triple: torch.Tensor) → torch.Tensor

    Parameters
    x

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

class dicee.models.BaseKGE(args: dict)
    Bases: BaseKGELightning
    Base class for all neural network modules.
    Your models should also subclass this class.
    Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-
    modules as regular attributes:

```

```

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

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

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

```

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

Note

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

Variables

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

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

`loss`

```

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

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

forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors

    Parameters
    -----

init_params_with_sanity_checking ()

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

    Parameters

        •  $\mathbf{x}$ 

        •  $\mathbf{y\_idx}$ 

        •  $\mathbf{ordered\_bpe\_entities}$ 

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

    Parameters
         $\mathbf{x}$ 

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

```

`get_head_relation_representation(indexed_triple)`

`get_sentence_representation(x: torch.LongTensor)`

Parameters

- **b** (x shape)
- 3
- **t**)

`get_bpe_head_and_relation_representation(x: torch.LongTensor)`
→ Tuple[torch.FloatTensor, torch.FloatTensor]

Parameters

x ($B \times 2 \times T$)

`get_embeddings()` → Tuple[numpy.ndarray, numpy.ndarray]

`class dicee.models.IdentityClass(args=None)`

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

args = None

```
__call__(x)
```

```
static forward(x)
```

```
dicee.models.octonion_mul(*, O_1, O_2)
```

```
dicee.models.octonion_mul_norm(*, O_1, O_2)
```

```
class dicee.models.OMult(args)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

```
name = '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., $[score(h,r,x)|x \text{ in Entities}] \Rightarrow [0.0,0.1,...,0.8]$, shape= $\Rightarrow (1, \text{Entities!})$ Given a batch of head entities and relations \Rightarrow shape (size of batch,| Entities!)

```
class dicee.models.ConvO(args: dict)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training (bool)` – Boolean represents whether this module is in training or evaluation mode.

`name = 'ConvO'`

`conv2d`

`fc_num_input`

`fc1`

`bn_conv2d`

`norm_fc1`

`feature_map_dropout`

`static octonion_normalizer(emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4, emb_rel_e5, emb_rel_e6, emb_rel_e7)`

`residual_convolution(O_1, O_2)`

forward_triples (*x: torch.Tensor*) → torch.Tensor

Parameters

x

forward_k_vs_all (*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **Entities**) Given a batch of head entities and relations => shape (size of batch,|Entities)

```
class dicee.models.AConvO(args: dict)
```

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Octonion Knowledge Graph Embeddings

name = 'AConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

static octonion_normalizer (*emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4, emb_rel_e5, emb_rel_e6, emb_rel_e7*)

residual_convolution (*O_1, O_2*)

forward_triples (*x: torch.Tensor*) → torch.Tensor

Parameters

x

forward_k_vs_all (*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **Entities**) Given a batch of head entities and relations => shape (size of batch,|Entities)

```
class dicee.models.Keci(args)
```

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
```

(continues on next page)

```

super().__init__()
self.conv1 = nn.Conv2d(1, 20, 5)
self.conv2 = nn.Conv2d(20, 20, 5)

def forward(self, x):
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

name = 'Keci'

p

q

r

requires_grad_for_interactions = True

compute_sigma_pp (*hp, rp*)

Compute $\text{sigma_pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k$

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., $e_1 e_1, e_1 e_2, e_1 e_3$,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals: $e_1 e_2, e_1 e_3, e_2 e_3$.

compute_sigma_qq (*hq, rq*)

Compute $\text{sigma_qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k \text{sigma_q}$ captures the interactions between along q bases For instance, let q e_1, e_2, e_3 , we compute interactions between $e_1 e_2, e_1 e_3$, and $e_2 e_3$ This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

for k in range(j + 1, q):

 results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

```
sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))
```

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., e_1e_1 , e_1e_2 , e_1e_3 ,

e_2e_1 , e_2e_2 , e_2e_3 , e_3e_1 , e_3e_2 , e_3e_3

Then select the triangular matrix without diagonals: e_1e_2 , e_1e_3 , e_2e_3 .

```
compute_sigma_pq(*, hp, hq, rp, rq)
```

```
sum_{i=1}^p sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j
```

```
results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):
```

```
    for j in range(q):
```

```
        sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
```

```
    print(sigma_pq.shape)
```

```
apply_coefficients(hp, hq, rp, rq)
```

Multiplying a base vector with its scalar coefficient

```
clifford_multiplication(h0, hp, hq, r0, rp, rq)
```

Compute our CL multiplication

```
h = h_0 + sum_{i=1}^p h_i e_i + sum_{j=p+1}^{p+q} h_j e_j
r = r_0 + sum_{i=1}^p r_i e_i + sum_{j=p+1}^{p+q} r_j e_j
```

$e_i^2 = +1$ for $i \leq p$ $e_j^2 = -1$ for $p < j \leq p+q$ $e_i e_j = -e_j e_i$ for i

e_j

$h r = \text{sigma}_0 + \text{sigma}_p + \text{sigma}_q + \text{sigma}_{\{pp\}} + \text{sigma}_{\{q\}} + \text{sigma}_{\{pq\}}$ where

(1) $\text{sigma}_0 = h_0 r_0 + \sum_{i=1}^p (h_0 r_i - h_i r_0) e_i - \sum_{j=p+1}^{p+q} (h_j r_j) e_j$

(2) $\text{sigma}_p = \sum_{i=1}^p (h_0 r_i + h_i r_0) e_i$

(3) $\text{sigma}_q = \sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j$

(4) $\text{sigma}_{\{pp\}} = \sum_{i=1}^p \sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k$

(5) $\text{sigma}_{\{qq\}} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k$

(6) $\text{sigma}_{\{pq\}} = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$

```
construct_cl_multivector(x: torch.FloatTensor, r: int, p: int, q: int)
```

```
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]
```

Construct a batch of multivectors $\text{Cl}_{\{p,q\}}(\mathbb{R}^d)$

Parameter

x : torch.FloatTensor with (n,d) shape

returns

- **a0** (torch.FloatTensor with (n,r) shape)
- **ap** (torch.FloatTensor with (n,r,p) shape)
- **aq** (torch.FloatTensor with (n,r,q) shape)

```
forward_k_vs_with_explicit(x: torch.Tensor)
```

```
k_vs_all_score(bpe_head_ent_emb, bpe_rel_ent_emb, E)
```

forward_k_vs_all (*x*: torch.Tensor) → torch.FloatTensor

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations \mathbb{R}^d .
- (2) Construct head entity and relation embeddings according to $Cl_{\{p,q\}}(\mathbb{R}^d)$.
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this functons are identical Parameter ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, **IEI**) shape

construct_batch_selected_cl_multivector (*x*: torch.FloatTensor, *r*: int, *p*: int, *q*: int)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of batchs multivectors $Cl_{\{p,q\}}(\mathbb{R}^d)$

Parameter

x: torch.FloatTensor with (n,k, d) shape

returns

- **a0** (torch.FloatTensor with (n,k, m) shape)
- **ap** (torch.FloatTensor with (n,k, m, p) shape)
- **aq** (torch.FloatTensor with (n,k, m, q) shape)

forward_k_vs_sample (*x*: torch.LongTensor, *target_entity_idx*: torch.LongTensor) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,2) shape

target_entity_idx: torch.LongTensor with (n, k) shape k denotes the selected number of examples.

rtype

torch.FloatTensor with (n, k) shape

score (*h*, *r*, *t*)

forward_triples (*x*: torch.Tensor) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,3) shape

rtype

torch.FloatTensor with (n) shape

class dicee.models.**CKeci** (*args*)

Bases: *Keci*

Without learning dimension scaling

name = 'CKeci'

requires_grad_for_interactions = False

```
class dicee.models.DeCaL(args)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

name = 'DeCaL'

entity_embeddings

relation_embeddings

p

q

r

re

forward_triples (*x: torch.Tensor*) \rightarrow torch.FloatTensor

Parameter

x: torch.LongTensor with (n,) shape

rtype

torch.FloatTensor with (n) shape

cl_pqr (*a: torch.tensor*) → torch.tensor

Input: tensor(batch_size, emb_dim) → output: tensor with 1+p+q+r components with size (batch_size, emb_dim/(1+p+q+r)) each.

1) takes a tensor of size (batch_size, emb_dim), split it into 1 + p + q + r components, hence 1+p+q+r must be a divisor of the emb_dim. 2) Return a list of the 1+p+q+r components vectors, each are tensors of size (batch_size, emb_dim/(1+p+q+r))

compute_sigmas_single (*list_h_emb, list_r_emb, list_t_emb*)

here we compute all the sums with no others vectors interaction taken with the scalar product with t, that is,

$$s0 = h_0 r_0 t_0 s1 = \sum_{i=1}^p h_i r_i t_0 s2 = \sum_{j=p+1}^{p+q} h_j r_j t_0 s3 = \sum_{i=1}^q (h_0 r_i t_i + h_i r_0 t_i) s4 = \sum_{i=p+1}^{p+q} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+1}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i)$$

and return:

$$sigma_0 t = \sigma_0 \cdot t_0 = s0 + s1 - s2 s3, s4 \text{ and } s5$$

compute_sigmas_multivect (*list_h_emb, list_r_emb*)

Here we compute and return all the sums with vectors interaction for the same and different bases.

For same bases vectors interaction we have

$$\sigma_p p = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (h_i r_{i'} - h_{i'} r_i) (model \text{ the interactions between } e_i \text{ and } e_{i'} \text{ for } 1 \leq i, i' \leq p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{j'} r_j)$$

For different base vector interactions, we have

$$\sigma_p q = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) (interactions \text{ between } e_i \text{ and } e_j \text{ for } 1 \leq i \leq p \text{ and } p+1 \leq j \leq p+q) \sigma_p r = \sum_{i=1}^p \sum_{j=p+q+1}^{p+q+r} (h_i r_j - h_j r_i)$$

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations
- (2) Construct head entity and relation embeddings according to $Cl_{\{p,q,r\}}(\mathbb{R}^d)$.
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this functions are identical Parameter ——— x: torch.LongTensor with (n,) shape :rtype: torch.FloatTensor with (n, **IE**) shape

apply_coefficients (*h0, hp, hq, hk, r0, rp, rq, rk*)

Multiplying a base vector with its scalar coefficient

construct_cl_multivector (*x: torch.FloatTensor, re: int, p: int, q: int, r: int*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors $Cl_{\{p,q,r\}}(\mathbb{R}^d)$

Parameter

x: torch.FloatTensor with (n,d) shape

returns

- **a0** (torch.FloatTensor)
- **ap** (torch.FloatTensor)
- **aq** (torch.FloatTensor)
- **ar** (torch.FloatTensor)

compute_sigma_pp (hp, rp)

Compute .. math:

$$\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (x_{iy_{i'}} - x_{i'y_i})$$

sigma_{pp} captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

for k in range(i + 1, p):

 results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

compute_sigma_qq (hq, rq)

Compute

$$\sigma_{q,q}^* = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (x_j y_{j'} - x_{j'} y_j) E q.16$$

sigma_{q} captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

for k in range(j + 1, q):

 results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

compute_sigma_rr (hk, rk)

$$\sigma_{r,r}^* = \sum_{k=p+q+1}^{p+q+r-1} \sum_{k'=k+1}^p (x_k y_{k'} - x_{k'} y_k)$$

```
compute_sigma_pq(*, hp, hq, rp, rq)
```

Compute

$$\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$$

```
results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):
```

```
    for j in range(q):
```

```
        sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
```

```
    print(sigma_pq.shape)
```

```
compute_sigma_pr(*, hp, hk, rp, rk)
```

Compute

$$\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$$

```
results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):
```

```
    for j in range(q):
```

```
        sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
```

```
    print(sigma_pq.shape)
```

```
compute_sigma_qr(*, hq, hk, rq, rk)
```

$$\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$$

```
results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):
```

```
    for j in range(q):
```

```
        sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
```

```
    print(sigma_pq.shape)
```

```
class dicee.models.BaseKGE(args: dict)
```

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)
```

(continues on next page)

```
def forward(self, x):
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training (bool)` – Boolean represents whether this module is in training or evaluation mode.

args

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

loss

`selected_optimizer = None`

`normalizer_class = None`

`normalize_head_entity_embeddings`

`normalize_relation_embeddings`

`normalize_tail_entity_embeddings`

`hidden_normalizer`

```

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

    Parameters
         $\mathbf{x} (B \times 2 \times T)$ 

forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors

    Parameters
    -----

init_params_with_sanity_checking ()

forward (x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
        y_idx: torch.LongTensor = None)

    Parameters

        •  $\mathbf{x}$ 

        •  $\mathbf{y\_idx}$ 

        • ordered_bpe_entities

forward_triples (x: torch.LongTensor)  $\rightarrow$  torch.Tensor

    Parameters

         $\mathbf{x}$ 

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

get_sentence_representation (x: torch.LongTensor)

    Parameters

        • ( $\mathbf{b} (x \text{ shape})$ )

        • 3

        •  $\mathbf{t}$ )

```

```

get_bpe_head_and_relation_representation (x: torch.LongTensor)
    → Tuple[torch.FloatTensor, torch.FloatTensor]

```

Parameters

$\mathbf{x} (B \times 2 \times T)$

```

get_embeddings () → Tuple[numpy.ndarray, numpy.ndarray]

```

```

class dicee.models.PykeenKGE (args: dict)

```

Bases: `dicee.models.base_model.BaseKGE`

A class for using knowledge graph embedding models implemented in Pykeen

Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Pykeen_HolE: Pykeen_HolE: Pykeen_TransD: Pykeen_TransE: Pykeen_TransF: Pykeen_TransH: Pykeen_TransR:

model_kwargs

name

model

loss_history = []

args

entity_embeddings = None

relation_embeddings = None

```

forward_k_vs_all (x: torch.LongTensor)

```

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads and relations + apply Dropout & Normalization if given. h, r = self.get_head_relation_representation(x) # (2) Reshape (1). if self.last_dim > 0:

h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim)

(3) Reshape all entities. if self.last_dim > 0:

t = self.entity_embeddings.weight.reshape(self.num_entities, self.embedding_dim, self.last_dim)

else:

t = self.entity_embeddings.weight

(4) Call the score_t from interactions to generate triple scores. return self.interaction.score_t(h=h, r=r, all_entities=t, slice_size=1)

```

forward_triples (x: torch.LongTensor) → torch.FloatTensor

```

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads, relations and tails and apply Dropout & Normalization if given. h, r, t = self.get_triple_representation(x) # (2) Reshape (1). if self.last_dim > 0:

h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim) t = t.reshape(len(x), self.embedding_dim, self.last_dim)

(3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice_size=None, slice_dim=0)

```
abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)
```

```
class dicee.models.BaseKGE (args: dict)
```

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training (bool)` – Boolean represents whether this module is in training or evaluation mode.

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

```

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all(x: torch.LongTensor)

    Parameters
         $\mathbf{x}$  ( $B \times 2 \times T$ )

forward_byte_pair_encoded_triple(x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors

    Parameters
    -----

init_params_with_sanity_checking()

forward(x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
        y_idx: torch.LongTensor = None)

    Parameters
        •  $\mathbf{x}$ 
        •  $\mathbf{y\_idx}$ 
        • ordered_bpe_entities

```

```

forward_triples (x: torch.LongTensor) → torch.Tensor

    Parameters
    x

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

get_sentence_representation (x: torch.LongTensor)

    Parameters
    • (b (x shape)
    • 3
    • t)

get_bpe_head_and_relation_representation (x: torch.LongTensor)
    → Tuple[torch.FloatTensor, torch.FloatTensor]

    Parameters
    x (B × 2 × x T)

get_embeddings () → Tuple[numpy.ndarray, numpy.ndarray]

class dicee.models.FMult (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    name = 'FMult'
    entity_embeddings
    relation_embeddings
    k
    num_sample = 50
    gamma
    roots
    weights
    compute_func (weights: torch.FloatTensor, x) → torch.FloatTensor
    chain_func (weights, x: torch.FloatTensor)
    forward_triples (idx_triple: torch.Tensor) → torch.Tensor

    Parameters
    x

```

```

class dicee.models.GFMult (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    name = 'GFMult'
    entity_embeddings
    relation_embeddings
    k
    num_sample = 250
    roots
    weights
    compute_func (weights: torch.FloatTensor, x) → torch.FloatTensor
    chain_func (weights, x: torch.FloatTensor)
    forward_triples (idx_triple: torch.Tensor) → torch.Tensor

```

Parameters

x

```

class dicee.models.FMult2 (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    name = 'FMult2'
    n_layers = 3
    k
    n = 50
    score_func = 'compositional'
    discrete_points
    entity_embeddings
    relation_embeddings
    build_func (Vec)
    build_chain_funcs (list_Vec)
    compute_func (W, b, x) → torch.FloatTensor
    function (list_W, list_b)
    trapezoid (list_W, list_b)

```

forward_triples (*idx_triple*: *torch.Tensor*) \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: $f(x) = \sum_{k=0}^{d-1} w_k e^{kix}$. and use the three differents scoring function as in the paper to evaluate the score

name = 'LFMult1'

entity_embeddings

relation_embeddings

forward_triples (*idx_triple*)

Parameters

x

tri_score (*h, r, t*)

vtp_score (*h, r, t*)

class *dicee.models.LFMult* (*args*)

Bases: *dicee.models.base_model.BaseKGE*

Embedding with polynomial functions. We represent all entities and relations in the polynomial space as: $f(x) = \sum_{i=0}^{d-1} a_i x^{i\%d}$ 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. $h = \sigma(w_h^T x + b_h)$, $r = \sigma(w_r^T x + b_r)$, $t = \sigma(w_t^T x + b_t)$

linear (*x, w, b*)

scalar_batch_NN (*a, b, c*)
 element wise multiplication between a,b and c: Inputs : a, b, c ==> torch.tensor of size batch_size x m x d
 Output : a tensor of size batch_size x d

tri_score (*coeff_h, coeff_r, coeff_t*)
 this part implement the trilinear scoring techniques:

$$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*b_j*c_k\}\{1+(i+j+k)\%d\}$$

1. generate the range for i,j and k from [0 d-1]
2. perform $\text{dfrac}\{a_i*b_j*c_k\}\{1+(i+j+k)\%d\}$ in parallel for every batch
3. take the sum over each batch

vtp_score (*h, r, t*)
 this part implement the vector triple product scoring techniques:

$$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*c_j*b_k - b_i*c_j*a_k\}\{(1+(i+j)\%d)(1+k)\}$$

1. generate the range for i,j and k from [0 d-1]
2. Compute the first and second terms of the sum
3. Multiply with then denominator and take the sum
4. take the sum over each batch

comp_func (*h, r, t*)
 this part implement the function composition scoring techniques: i.e. score = <hor, t>

polynomial (*coeff, x, degree*)
 This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,...d] and return a vector tensor (coeff[0][0] + coeff[0][1]x +...+ coeff[0][d]x^d,

$$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d)$$

pop (*coeff, x, degree*)
 This function allow us to evaluate the composition of two polynomes without for loops :) it takes a matrix tensor of coefficients (coeff), a matrix tensor of points x and range of integer [0,1,...d]
 and return a tensor (coeff[0][0] + coeff[0][1]x +...+ coeff[0][d]x^d,

$$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d)$$

class dicee.models.DualE (*args*)
 Bases: *dicee.models.base_model.BaseKGE*
 Dual Quaternion Knowledge Graph Embeddings (<https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657>)
name = 'DualE'
entity_embeddings
relation_embeddings
num_ent = None
kvsall_score (*e_1_h, e_2_h, e_3_h, e_4_h, e_5_h, e_6_h, e_7_h, e_8_h, e_1_t, e_2_t, e_3_t, e_4_t, e_5_t, e_6_t, e_7_t, e_8_t, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8*) → torch.tensor
 KvsAll scoring function

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_triples (*idx_triple: torch.tensor*) → torch.tensor

Negative Sampling forward pass:

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_k_vs_all (*x*)

KvsAll forward pass

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

T (*x: torch.tensor*) → torch.tensor

Transpose function

Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)

dicee.query_generator

Classes

QueryGenerator

Module Contents

```
class dicee.query_generator.QueryGenerator (train_path, val_path: str, test_path: str,  
      ent2id: Dict = None, rel2id: Dict = None, seed: int = 1, gen_valid: bool = False,  
      gen_test: bool = True)
```

train_path

val_path

test_path

```

gen_valid = False

gen_test = True

seed = 1

max_ans_num = 1000000.0

mode

ent2id = None

rel2id: Dict = None

ent_in: Dict

ent_out: Dict

query_name_to_struct

list2tuple(list_data)

tuple2list(x: List | Tuple) → List | Tuple
    Convert a nested tuple to a nested list.

set_global_seed(seed: int)
    Set seed

construct_graph(paths: List[str]) → Tuple[Dict, Dict]
    Construct graph from triples Returns dicts with incoming and outgoing edges

fill_query(query_structure: List[str | List], ent_in: Dict, ent_out: Dict, answer: int) → bool
    Private method for fill_query logic.

achieve_answer(query: List[str | List], ent_in: Dict, ent_out: Dict) → set
    Private method for achieve_answer logic. @TODO: Document the code

write_links(ent_out, small_ent_out)

ground_queries(query_structure: List[str | List], ent_in: Dict, ent_out: Dict, small_ent_in: Dict,
               small_ent_out: Dict, gen_num: int, query_name: str)
    Generating queries and achieving answers

unmap(query_type, queries, tp_answers, fp_answers, fn_answers)

unmap_query(query_structure, query, id2ent, id2rel)

generate_queries(query_struct: List, gen_num: int, query_type: str)
    Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting
    queries and answers in return @ TODO: create a class for each single query struct

save_queries(query_type: str, gen_num: int, save_path: str)

abstract load_queries(path)

get_queries(query_type: str, gen_num: int)

static save_queries_and_answers(path: str, data: List[Tuple[str, Tuple[collections.defaultdict]]])
    → None
    Save Queries into Disk

```

static load_queries_and_answers (*path: str*) → List[Tuple[str, Tuple[collections.defaultdict]]]
 Load Queries from Disk to Memory

dicee.read_preprocess_save_load_kg

Submodules

dicee.read_preprocess_save_load_kg.preprocess

Classes

<i>PreprocessKG</i>	Preprocess the data in memory
---------------------	-------------------------------

Module Contents

class dicee.read_preprocess_save_load_kg.preprocess.**PreprocessKG** (*kg*)

Preprocess the data in memory

kg

start () → None

Preprocess train, valid and test datasets stored in knowledge graph instance

Parameter

rtype

None

preprocess_with_byte_pair_encoding ()

preprocess_with_byte_pair_encoding_with_padding () → None

Preprocess with byte pair encoding and add padding

preprocess_with_pandas () → None

Preprocess with pandas: add reciprocal triples, construct vocabulary, and index datasets

preprocess_with_polars () → None

Preprocess with polars: add reciprocal triples and create indexed datasets

sequential_vocabulary_construction () → None

(1) Read input data into memory

(2) Remove triples with a condition

(3) **Serialize vocabularies in a pandas dataframe where**

=> the index is integer and => a single column is string (e.g. URI)

dicee.read_preprocess_save_load_kg.read_from_disk

Classes

<i>ReadFromDisk</i>	Read the data from disk into memory
---------------------	-------------------------------------

Module Contents

class `dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk(kg)`

Read the data from disk into memory

kg

start () → None

Read a knowledge graph from disk into memory

Data will be available at the `train_set`, `test_set`, `valid_set` attributes.

Parameter

None

rtype

None

add_noisy_triples_into_training ()

`dicee.read_preprocess_save_load_kg.save_load_disk`

Classes

LoadSaveToDisk

Module Contents

class `dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk(kg)`

kg

save ()

load ()

dicee.read_preprocess_save_load_kg.util

Functions

<code>polars_dataframe_indexer</code> (\rightarrow polars.DataFrame)	Replaces 'subject', 'relation', and 'object' columns in the input Polars DataFrame with their corresponding index values
<code>pandas_dataframe_indexer</code> (\rightarrow pandas.DataFrame)	Replaces 'subject', 'relation', and 'object' columns in the input Pandas DataFrame with their corresponding index values
<code>apply_reciprocal_or_noise</code> (add_reciprocal, eval_model)	Add reciprocal triples if conditions are met
<code>timeit</code> (func)	
<code>read_with_polars</code> (\rightarrow polars.DataFrame)	Load and Preprocess via Polars
<code>read_with_pandas</code> (data_path[, read_only_few, ...])	Load and Preprocess via Pandas
<code>read_from_disk</code> (\rightarrow Tuple[polars.DataFrame, pandas.DataFrame])	
<code>count_triples</code> (\rightarrow int)	Returns the total number of triples in the triple store.
<code>fetch_worker</code> (endpoint, offsets, chunk_size, ...)	Worker process: fetch assigned chunks and save to disk with per-worker tqdm.
<code>read_from_triple_store_with_polars</code> (endpoint[, ...])	Main function to read all triples in parallel, save as Parquet, and load into Polars dataframe.
<code>read_from_triple_store_with_pandas</code> ([endpoint]	Read triples from triple store into pandas dataframe
<code>get_er_vocab</code> (data[, file_path])	
<code>get_re_vocab</code> (data[, file_path])	
<code>get_ee_vocab</code> (data[, file_path])	
<code>create_constraints</code> (triples[, file_path])	
<code>load_with_pandas</code> (\rightarrow None)	Deserialize data
<code>save_numpy_ndarray</code> (*, data, file_path)	
<code>load_numpy_ndarray</code> (*, file_path)	
<code>save_pickle</code> (*, data[, file_path])	
<code>load_pickle</code> (*[, file_path])	
<code>create_recipriocal_triples</code> (x)	Add inverse triples into dask dataframe
<code>dataset_sanity_checking</code> (\rightarrow None)	

Module Contents

`dicee.read_preprocess_save_load_kg.util.polars_dataframe_indexer` (
 df_polars: *polars.DataFrame*, *idx_entity*: *polars.DataFrame*, *idx_relation*: *polars.DataFrame*)
 \rightarrow *polars.DataFrame*

Replaces 'subject', 'relation', and 'object' columns in the input Polars DataFrame with their corresponding index values from the entity and relation index DataFrames.

This function processes the DataFrame in three main steps: 1. Replace the 'relation' values with the corresponding index from *idx_relation*. 2. Replace the 'subject' values with the corresponding index from *idx_entity*. 3. Replace the 'object' values with the corresponding index from *idx_entity*.

Parameters:

df_polars

[polars.DataFrame] The input Polars DataFrame containing columns: 'subject', 'relation', and 'object'.

idx_entity

[polars.DataFrame] A Polars DataFrame that contains the mapping between entity names and their corresponding indices. Must have columns: 'entity' and 'index'.

idx_relation

[polars.DataFrame] A Polars DataFrame that contains the mapping between relation names and their corresponding indices. Must have columns: 'relation' and 'index'.

Returns:

polars.DataFrame

A DataFrame with the 'subject', 'relation', and 'object' columns replaced by their corresponding indices.

Example Usage:

```
>>> df_polars = pl.DataFrame({
    "subject": ["Alice", "Bob", "Charlie"],
    "relation": ["knows", "works_with", "lives_in"],
    "object": ["Dave", "Eve", "Frank"]
})
>>> idx_entity = pl.DataFrame({
    "entity": ["Alice", "Bob", "Charlie", "Dave", "Eve", "Frank"],
    "index": [0, 1, 2, 3, 4, 5]
})
>>> idx_relation = pl.DataFrame({
    "relation": ["knows", "works_with", "lives_in"],
    "index": [0, 1, 2]
})
>>> polars_dataframe_indexer(df_polars, idx_entity, idx_relation)
```

Steps:

1. Join the input DataFrame *df_polars* on the 'relation' column with *idx_relation* to replace the relations with their indices.
2. Join on 'subject' to replace it with the corresponding entity index using a left join on *idx_entity*.
3. Join on 'object' to replace it with the corresponding entity index using a left join on *idx_entity*.
4. Select only the 'subject', 'relation', and 'object' columns to return the final result.

```
dicee.read_preprocess_save_load_kg.util.pandas_dataframe_indexer(
    df_pandas: pandas.DataFrame, idx_entity: pandas.DataFrame, idx_relation: pandas.DataFrame
) → pandas.DataFrame
```

Replaces 'subject', 'relation', and 'object' columns in the input Pandas DataFrame with their corresponding index values from the entity and relation index DataFrames.

Parameters:

df_pandas

[pd.DataFrame] The input Pandas DataFrame containing columns: 'subject', 'relation', and 'object'.

idx_entity

[pd.DataFrame] A Pandas DataFrame that contains the mapping between entity names and their corresponding indices. Must have columns: 'entity' and 'index'.

idx_relation

[pd.DataFrame] A Pandas DataFrame that contains the mapping between relation names and their corresponding indices. Must have columns: 'relation' and 'index'.

Returns:

pd.DataFrame

A DataFrame with the 'subject', 'relation', and 'object' columns replaced by their corresponding indices.

```
dicee.read_preprocess_save_load_kg.util.apply_reciprocal_or_noise (add_reciprocal: bool,  
    eval_model: str, df: object = None, info: str = None)
```

Add reciprocal triples if conditions are met

```
dicee.read_preprocess_save_load_kg.util.timeit (func)
```

```
dicee.read_preprocess_save_load_kg.util.read_with_polars (data_path,  
    read_only_few: int = None, sample_triples_ratio: float = None, separator: str = None)  
    → polars.DataFrame
```

Load and Preprocess via Polars

```
dicee.read_preprocess_save_load_kg.util.read_with_pandas (data_path,  
    read_only_few: int = None, sample_triples_ratio: float = None, separator: str = None)
```

Load and Preprocess via Pandas

```
dicee.read_preprocess_save_load_kg.util.read_from_disk (data_path: str,  
    read_only_few: int = None, sample_triples_ratio: float = None, backend: str = None,  
    separator: str = None) → Tuple[polars.DataFrame, pandas.DataFrame]
```

```
dicee.read_preprocess_save_load_kg.util.count_triples (endpoint: str) → int
```

Returns the total number of triples in the triple store.

```
dicee.read_preprocess_save_load_kg.util.fetch_worker (endpoint: str, offsets: list[int],  
    chunk_size: int, output_dir: str, worker_id: int)
```

Worker process: fetch assigned chunks and save to disk with per-worker tqdm.

```
dicee.read_preprocess_save_load_kg.util.read_from_triple_store_with_polars (  
    endpoint: str, chunk_size: int = 500000, output_dir: str = 'triples_parquet')
```

Main function to read all triples in parallel, save as Parquet, and load into Polars dataframe.

```
dicee.read_preprocess_save_load_kg.util.read_from_triple_store_with_pandas (  
    endpoint: str = None)
```

Read triples from triple store into pandas dataframe

```
dicee.read_preprocess_save_load_kg.util.get_er_vocab (data, file_path: str = None)
```

```
dicee.read_preprocess_save_load_kg.util.get_re_vocab (data, file_path: str = None)
```

```
dicee.read_preprocess_save_load_kg.util.get_ee_vocab (data, file_path: str = None)
```

```
dicee.read_preprocess_save_load_kg.util.create_constraints (triples, file_path: str = None)
```

(1) Extract domains and ranges of relations

(2) Store a mapping from relations to entities that are outside of the domain and range. Create constrained entities based on the range of relations :param triples: :return: Tuple[dict, dict]

```
dicee.read_preprocess_save_load_kg.util.load_with_pandas (self) → None
```

Deserialize data

```
dicee.read_preprocess_save_load_kg.util.save_numpy_ndarray (*, data: numpy.ndarray,  
file_path: str)
```

```
dicee.read_preprocess_save_load_kg.util.load_numpy_ndarray (*, file_path: str)
```

```
dicee.read_preprocess_save_load_kg.util.save_pickle (*, data: object, file_path=str)
```

```
dicee.read_preprocess_save_load_kg.util.load_pickle (*, file_path=str)
```

```
dicee.read_preprocess_save_load_kg.util.create_recipriocal_triples (x)
```

Add inverse triples into dask dataframe :param x: :return:

```
dicee.read_preprocess_save_load_kg.util.dataset_sanity_checking (  
train_set: numpy.ndarray, num_entities: int, num_relations: int) → None
```

Parameters

- **train_set**
- **num_entities**
- **num_relations**

Returns

Classes

<i>PreprocessKG</i>	Preprocess the data in memory
<i>LoadSaveToDisk</i>	
<i>ReadFromDisk</i>	Read the data from disk into memory

Package Contents

```
class dicee.read_preprocess_save_load_kg.PreprocessKG (kg)
```

Preprocess the data in memory

kg

start () → None

Preprocess train, valid and test datasets stored in knowledge graph instance

Parameter

rtype
None

`preprocess_with_byte_pair_encoding()`

`preprocess_with_byte_pair_encoding_with_padding()` → None

Preprocess with byte pair encoding and add padding

`preprocess_with_pandas()` → None

Preprocess with pandas: add reciprocal triples, construct vocabulary, and index datasets

`preprocess_with_polars()` → None

Preprocess with polars: add reciprocal triples and create indexed datasets

`sequential_vocabulary_construction()` → None

(1) Read input data into memory

(2) Remove triples with a condition

(3) **Serialize vocabularies in a pandas dataframe where**

=> the index is integer and => a single column is string (e.g. URI)

`class dicee.read_preprocess_save_load_kg.LoadSaveToDisk(kg)`

`kg`

`save()`

`load()`

`class dicee.read_preprocess_save_load_kg.ReadFromDisk(kg)`

Read the data from disk into memory

`kg`

`start()` → None

Read a knowledge graph from disk into memory

Data will be available at the `train_set`, `test_set`, `valid_set` attributes.

Parameter

None

rtype

None

`add_noisy_triples_into_training()`

dicee.sanity_checkers

Functions

`is_sparql_endpoint_alive([sparql_endpoint])`

`validate_knowledge_graph(args)`

Validating the source of knowledge graph

`sanity_checking_with_arguments(args)`

`sanity_check_callback_args(args)`

Perform sanity checks on callback-related arguments.

Module Contents

`dicee.sanity_checkers.is_sparql_endpoint_alive` (*sparql_endpoint: str = None*)

`dicee.sanity_checkers.validate_knowledge_graph` (*args*)

Validating the source of knowledge graph

`dicee.sanity_checkers.sanity_checking_with_arguments` (*args*)

`dicee.sanity_checkers.sanity_check_callback_args` (*args*)

Perform sanity checks on callback-related arguments.

`dicee.scripts`

Submodules

`dicee.scripts.index_serve`

```
$ docker pull qdrant/qdrant && docker run -p 6333:6333 -p 6334:6334 -v $(pwd)/qdrant_storage:/qdrant/storage:z  
qdrant/qdrant $ dicee_vector_db -index -serve -path CountryEmbeddings -collection "countries_vdb"
```

Attributes

app

neural_searcher

Classes

NeuralSearcher

StringListRequest

Functions

```
get_default_arguments()
```

```
index(args)
```

```
root()
```

```
search_embeddings(q)
```

```
retrieve_embeddings(q)
```

```
search_embeddings_batch(request)
```

```
serve(args)
```

```
main()
```

Module Contents

```
dicee.scripts.index_serve.get_default_arguments()
```

```
dicee.scripts.index_serve.index(args)
```

```
dicee.scripts.index_serve.app
```

```
dicee.scripts.index_serve.neural_searcher = None
```

```
class dicee.scripts.index_serve.NeuralSearcher(args)
```

```
    collection_name
```

```
    entity_to_idx = None
```

```
    qdrant_client
```

```
    topk = 5
```

```
    retrieve_embedding(entity: str = None, entities: List[str] = None) → List
```

```
    search(entity: str)
```

```
async dicee.scripts.index_serve.root()
```

```
async dicee.scripts.index_serve.search_embeddings(q: str)
```

```
async dicee.scripts.index_serve.retrieve_embeddings(q: str)
```

```
class dicee.scripts.index_serve.StringListRequest
```

```
    Bases: pydantic.BaseModel
```

```
    queries: List[str]
```

```
    reducer: str | None = None
```

async `dicee.scripts.index_serve.search_embeddings_batch` (*request: StringListRequest*)

`dicee.scripts.index_serve.serve` (*args*)

`dicee.scripts.index_serve.main` ()

dicee.scripts.run

Functions

<code>get_default_arguments</code> ([<i>description</i>])	Extends pytorch_lightning Trainer's arguments with ours
<code>main</code> ()	

Module Contents

`dicee.scripts.run.get_default_arguments` (*description=None*)

Extends pytorch_lightning Trainer's arguments with ours

`dicee.scripts.run.main` ()

dicee.static_funcs

Functions

<code>create_recipriocal_triples</code> (<i>x</i>)	Add inverse triples into dask dataframe
<code>get_er_vocab</code> (<i>data</i> [, <i>file_path</i>])	
<code>get_re_vocab</code> (<i>data</i> [, <i>file_path</i>])	
<code>get_ee_vocab</code> (<i>data</i> [, <i>file_path</i>])	
<code>timeit</code> (<i>func</i>)	
<code>save_pickle</code> (*[, <i>data</i> , <i>file_path</i>])	
<code>load_pickle</code> ([<i>file_path</i>])	
<code>load_term_mapping</code> ([<i>file_path</i>])	
<code>select_model</code> (<i>args</i> [, <i>is_continual_training</i> , <i>storage_path</i>])	
<code>load_model</code> (→ <i>Tuple</i> [<i>object</i> , <i>Tuple</i> [<i>dict</i> , <i>dict</i>]])	Load weights and initialize pytorch module from namespace arguments
<code>load_model_ensemble</code> (...)	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
<code>save_numpy_ndarray</code> (*, <i>data</i> , <i>file_path</i>)	
<code>numpy_data_type_changer</code> (→ <i>numpy.ndarray</i>)	Detect most efficient data type for a given triples
<code>save_checkpoint_model</code> (→ <i>None</i>)	Store Pytorch model into disk

continues on next page

Table 2 – continued from previous page

<code>store(→ None)</code>	
<code>add_noisy_triples(→ pandas.DataFrame)</code>	Add randomly constructed triples
<code>read_or_load_kg(args, cls)</code>	
<code>intialize_model(→ Tuple[object, str])</code>	
<code>load_json(→ dict)</code>	
<code>save_embeddings(→ None)</code>	Save it as CSV if memory allows.
<code>random_prediction(pre_trained_kge)</code>	
<code>deploy_triple_prediction(pre_trained_kge, str_subject, ...)</code>	
<code>deploy_tail_entity_prediction(pre_trained_kge, ...)</code>	
<code>deploy_head_entity_prediction(pre_trained_kge, ...)</code>	
<code>deploy_relation_prediction(pre_trained_kge, ...)</code>	
<code>vocab_to_parquet(vocab_to_idx, name, ...)</code>	
<code>create_experiment_folder([folder_name])</code>	
<code>continual_training_setup_executor(→ None)</code>	
<code>exponential_function(→ torch.FloatTensor)</code>	
<code>load_numpy(→ numpy.ndarray)</code>	
<code>evaluate(entity_to_idx, scores, easy_answers, # @TODO: CD: Renamed this function hard_answers)</code>	
<code>download_file(url[, destination_folder])</code>	
<code>download_files_from_url(→ None)</code>	
<code>download_pretrained_model(→ str)</code>	
<code>write_csv_from_model_parallel(path)</code>	Create
<code>from_pretrained_model_write_embeddings_int, None)</code>	

Module Contents

`dicee.static_funcs.create_recipriocal_triples(x)`

Add inverse triples into dask dataframe :param x: :return:

`dicee.static_funcs.get_er_vocab(data, file_path: str = None)`

`dicee.static_funcs.get_re_vocab(data, file_path: str = None)`

`dicee.static_funcs.get_ee_vocab(data, file_path: str = None)`

```

dicee.static_funcs.timeit (func)

dicee.static_funcs.save_pickle (*, data: object = None, file_path=str)

dicee.static_funcs.load_pickle (file_path=str)

dicee.static_funcs.load_term_mapping (file_path=str)

dicee.static_funcs.select_model (args: dict, is_continual_training: bool = None,
                                storage_path: str = None)

dicee.static_funcs.load_model (path_of_experiment_folder: str, model_name='model.pt', verbose=0)
    → Tuple[object, Tuple[dict, dict]]
    Load weights and initialize pytorch module from namespace arguments

dicee.static_funcs.load_model_ensemble (path_of_experiment_folder: str)
    → Tuple[dicee.models.base_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]
    Construct Ensemble Of weights and initialize pytorch module from namespace arguments
    (1) Detect models under given path
    (2) Accumulate parameters of detected models
    (3) Normalize parameters
    (4) Insert (3) into model.

dicee.static_funcs.save_numpy_ndarray (*, data: numpy.ndarray, file_path: str)

dicee.static_funcs.numpy_data_type_changer (train_set: numpy.ndarray, num: int)
    → numpy.ndarray
    Detect most efficient data type for a given triples :param train_set: :param num: :return:

dicee.static_funcs.save_checkpoint_model (model, path: str) → None
    Store Pytorch model into disk

dicee.static_funcs.store (trained_model, model_name: str = 'model', full_storage_path: str = None,
                        save_embeddings_as_csv=False) → None

dicee.static_funcs.add_noisy_triples (train_set: pandas.DataFrame, add_noise_rate: float)
    → pandas.DataFrame
    Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

dicee.static_funcs.read_or_load_kg (args, cls)

dicee.static_funcs.intialize_model (args: dict, verbose=0) → Tuple[object, str]

dicee.static_funcs.load_json (p: str) → dict

dicee.static_funcs.save_embeddings (embeddings: numpy.ndarray, indexes, path: str) → None
    Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:

dicee.static_funcs.random_prediction (pre_trained_kge)

dicee.static_funcs.deploy_triple_prediction (pre_trained_kge, str_subject, str_predicate,
                                           str_object)

dicee.static_funcs.deploy_tail_entity_prediction (pre_trained_kge, str_subject, str_predicate,
                                                top_k)

```

```

dicee.static_funcs.deploy_head_entity_prediction(pre_trained_kge, str_object, str_predicate,
top_k)

dicee.static_funcs.deploy_relation_prediction(pre_trained_kge, str_subject, str_object, top_k)

dicee.static_funcs.vocab_to_parquet(vocab_to_idx, name, path_for_serialization, print_into)

dicee.static_funcs.create_experiment_folder(folder_name='Experiments')

dicee.static_funcs.continual_training_setup_executor(executor) → None

dicee.static_funcs.exponential_function(x: numpy.ndarray, lam: float, ascending_order=True)
→ torch.FloatTensor

dicee.static_funcs.load_numpy(path) → numpy.ndarray

dicee.static_funcs.evaluate(entity_to_idx, scores, easy_answers, hard_answers)
# @TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

dicee.static_funcs.download_file(url, destination_folder='.')

dicee.static_funcs.download_files_from_url(base_url: str, destination_folder='.') → None

```

Parameters

- **base_url** (e.g. [“https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll”](https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll))
- **destination_folder** (e.g. `"KINSHIP-Keci-dim128-epoch256-KvsAll"`)

```

dicee.static_funcs.download_pretrained_model(url: str) → str

```

```

dicee.static_funcs.write_csv_from_model_parallel(path: str)

```

Create

```

dicee.static_funcs.from_pretrained_model_write_embeddings_into_csv(path: str) → None

```

dicee.static_funcs_training

Functions

```

make_iterable_verbose(→ Iterable)

```

```

evaluate_lp([model, triple_idx, num_entities, ...])

```

```

evaluate_bpe_lp(model, triple_idx, ..., info])

```

```

efficient_zero_grad(model)

```

Module Contents

```

dicee.static_funcs_training.make_iterable_verbose(iterable_object, verbose, desc='Default',
position=None, leave=True) → Iterable

```

```
dicee.static_funcs_training.evaluate_lp (model=None, triple_idx=None, num_entities=None,
      er_vocab: Dict[Tuple, List] = None, re_vocab: Dict[Tuple, List] = None, info='Eval Starts',
      batch_size=128, chunk_size=1000)
```

```
dicee.static_funcs_training.evaluate_bpe_lp (model, triple_idx: List[Tuple],
      all_bpe_shaped_entities, er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List],
      info='Eval Starts')
```

```
dicee.static_funcs_training.efficient_zero_grad (model)
```

dicee.static_preprocess_funcs

Attributes

```
enable_log
```

Functions

```
timeit(func)
```

```
preprocesses_input_args(args) Sanity Checking in input arguments
```

```
create_constraints(→ Tuple[dict, dict, dict, dict])
```

```
get_er_vocab(data)
```

```
get_re_vocab(data)
```

```
get_ee_vocab(data)
```

```
mapping_from_first_two_cols_to_third(train_se
```

Module Contents

```
dicee.static_preprocess_funcs.enable_log = False
```

```
dicee.static_preprocess_funcs.timeit (func)
```

```
dicee.static_preprocess_funcs.preprocesses_input_args (args)
```

Sanity Checking in input arguments

```
dicee.static_preprocess_funcs.create_constraints (triples: numpy.ndarray)
      → Tuple[dict, dict, dict, dict]
```

(1) Extract domains and ranges of relations

(2) Store a mapping from relations to entities that are outside of the domain and range. Create constraints entities based on the range of relations :param triples: :return:

```
dicee.static_preprocess_funcs.get_er_vocab (data)
```

```
dicee.static_preprocess_funcs.get_re_vocab (data)
```

```
dicee.static_preprocess_funcs.get_ee_vocab(data)
```

```
dicee.static_preprocess_funcs.mapping_from_first_two_cols_to_third(train_set_idx)
```

dicee.trainer

Submodules

dicee.trainer.dice_trainer

Classes

<i>DICE_Trainer</i>	DICE_Trainer implement
---------------------	------------------------

Functions

<i>load_term_mapping</i> ([file_path])
<i>initialize_trainer</i> (...)
<i>get_callbacks</i> (args)

Module Contents

```
dicee.trainer.dice_trainer.load_term_mapping(file_path=str)
```

```
dicee.trainer.dice_trainer.initialize_trainer(args, callbacks)  
→ dicee.trainer.torch_trainer.TorchTrainer | dicee.trainer.model_parallelism.TensorParallel | dicee.trainer.torch_trainer_ddp
```

```
dicee.trainer.dice_trainer.get_callbacks(args)
```

```
class dicee.trainer.dice_trainer.DICE_Trainer(args, is_continual_training: bool, storage_path,  
evaluator=None)
```

DICE_Trainer implement

- 1- Pytorch Lightning trainer (<https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html>)
- 2- Multi-GPU Trainer(<https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>)
- 3- CPU Trainer

args

is_continual_training:bool

storage_path:str

evaluator:

report:dict

report

args

trainer = None

is_continual_training

storage_path

evaluator = None

form_of_labelling = None

continual_start (*knowledge_graph*)

- (1) Initialize training.
- (2) Load model
- (3) Load trainer (3) Fit model

Parameter

returns

- *model*
- **form_of_labelling** (*str*)

initialize_trainer (*callbacks: List*)

→ *lightning.Trainer* | *dicke.trainer.model_parallelism.TensorParallel* | *dicke.trainer.torch_trainer.TorchTrainer* | *dicke.*

Initialize Trainer from input arguments

initialize_or_load_model ()

init_dataloader (*dataset: torch.utils.data.Dataset*) → *torch.utils.data.DataLoader*

init_dataset () → *torch.utils.data.Dataset*

start (*knowledge_graph: dicke.knowledge_graph.KG* | *numpy.memmap*)

→ *Tuple[dicke.models.base_model.BaseKGE, str]*

Start the training

- (1) Initialize Trainer
- (2) Initialize or load a pretrained KGE model

in DDP setup, we need to load the memory map of already read/index KG.

k_fold_cross_validation (*dataset*) → *Tuple[dicke.models.base_model.BaseKGE, str]*

Perform K-fold Cross-Validation

1. Obtain K train and test splits.
2. **For each split,**
 - 2.1 initialize trainer and model
 - 2.2. Train model with configuration provided in args.
 - 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.
3. Report the mean and average MRR .

Parameters

- **self**
- **dataset**

Returns

model

dicee.trainer.model_parallelism

Classes

<i>TensorParallel</i>	Abstract class for Trainer class for knowledge graph embedding models
-----------------------	---

Functions

<i>extract_input_outputs</i> (z[, device])
<i>find_good_batch_size</i> (train_loader, tp_ensemble_model)
<i>forward_backward_update_loss</i> (→ float)

Module Contents

dicee.trainer.model_parallelism.**extract_input_outputs** (z: list, device=None)

dicee.trainer.model_parallelism.**find_good_batch_size** (train_loader, tp_ensemble_model)

dicee.trainer.model_parallelism.**forward_backward_update_loss** (z: Tuple, ensemble_model)
→ float

class dicee.trainer.model_parallelism.**TensorParallel** (args, callbacks)

Bases: *dicee.abstracts.AbstractTrainer*

Abstract class for Trainer class for knowledge graph embedding models

Parameter

args

[str] ?

callbacks: list

?

fit (*args, **kwargs)

Train model

dicee.trainer.torch_trainer

Classes

<i>TorchTrainer</i>	TorchTrainer for using single GPU or multi CPUs on a single node
---------------------	--

Module Contents

class `dicee.trainer.torch_trainer.TorchTrainer` (*args*, *callbacks*)

Bases: `dicee.abstracts.AbstractTrainer`

TorchTrainer for using single GPU or multi CPUs on a single node

Arguments

`callbacks`: list of Abstract callback instances

`loss_function` = None

`optimizer` = None

`model` = None

`train_dataloaders` = None

`training_step` = None

`process`

`fit` (**args*, *train_dataloaders*, ***kwargs*) → None

Training starts

Arguments

kwargs:Tuple

empty dictionary

Return type

batch loss (float)

forward_backward_update (*x_batch: torch.Tensor*, *y_batch: torch.Tensor*) → torch.Tensor

Compute forward, loss, backward, and parameter update

Arguments

Return type

batch loss (float)

extract_input_outputs_set_device (*batch: list*) → Tuple

Construct inputs and outputs from a batch of inputs with outputs From a batch of inputs and put

Arguments

Return type

(tuple) mini-batch on select device

dicee.trainer.torch_trainer_ddp

Classes

<code>TorchDDPTrainer</code>	A Trainer based on torch.nn.parallel.DistributedDataParallel
<code>NodeTrainer</code>	

Functions

<code>make_iterable_verbose(→ Iterable)</code>
--

Module Contents

`dicee.trainer.torch_trainer_ddp.make_iterable_verbose(iterable_object, verbose, desc='Default', position=None, leave=True) → Iterable`

class `dicee.trainer.torch_trainer_ddp.TorchDDPTrainer(args, callbacks)`

Bases: `dicee.abstracts.AbstractTrainer`

A Trainer based on torch.nn.parallel.DistributedDataParallel

Arguments

entity_idx
mapping.

relation_idx
mapping.

form
?

store
?

label_smoothing_rate
Using hard targets (0,1) drives weights to infinity. An outlier produces enormous gradients.

Return type
torch.utils.data.Dataset

fit (*args, **kwargs)
Train model

class `dicee.trainer.torch_trainer_ddp.NodeTrainer(trainer, model: torch.nn.Module, train_dataset_loader: torch.utils.data.DataLoader, callbacks, num_epochs: int)`

trainer

local_rank

global_rank

```

optimizer

train_dataset_loader

loss_func

callbacks

model

num_epochs

loss_history = []

ctx

scaler

extract_input_outputs (z: list)

train()
    Training loop for DDP

```

Classes

DICE_Trainer

DICE_Trainer implement

Package Contents

```
class dicee.trainer.DICE_Trainer (args, is_continual_training: bool, storage_path, evaluator=None)
```

DICE_Trainer implement

- 1- Pytorch Lightning trainer (<https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html>)
- 2- Multi-GPU Trainer(<https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>)
- 3- CPU Trainer

args

is_continual_training:bool

storage_path:str

evaluator:

report:dict

report

args

trainer = None

is_continual_training

storage_path

evaluator = None

form_of_labelling = None

continual_start (*knowledge_graph*)

- (1) Initialize training.
- (2) Load model
- (3) Load trainer (3) Fit model

Parameter

returns

- *model*
- **form_of_labelling** (*str*)

initialize_trainer (*callbacks: List*)

→ *lightning.Trainer* | *dicke.trainer.model_parallelism.TensorParallel* | *dicke.trainer.torch_trainer.TorchTrainer* | *dicke.t*

Initialize Trainer from input arguments

initialize_or_load_model ()

init_dataloader (*dataset: torch.utils.data.Dataset*) → *torch.utils.data.DataLoader*

init_dataset () → *torch.utils.data.Dataset*

start (*knowledge_graph: dicke.knowledge_graph.KG* | *numpy.memmap*)

→ *Tuple[dicke.models.base_model.BaseKGE, str]*

Start the training

- (1) Initialize Trainer
- (2) Initialize or load a pretrained KGE model

in DDP setup, we need to load the memory map of already read/index KG.

k_fold_cross_validation (*dataset*) → *Tuple[dicke.models.base_model.BaseKGE, str]*

Perform K-fold Cross-Validation

1. Obtain K train and test splits.
2. **For each split,**
 - 2.1 initialize trainer and model
 - 2.2. Train model with configuration provided in args.
 - 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.
3. Report the mean and average MRR .

Parameters

- **self**
- **dataset**

Returns

model

14.2 Attributes

`__version__`

14.3 Classes

<i>Pyke</i>	A Physical Embedding Model for Knowledge Graphs
<i>DistMult</i>	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
<i>CKeci</i>	Without learning dimension scaling
<i>Keci</i>	Base class for all neural network modules.
<i>TransE</i>	Translating Embeddings for Modeling
<i>DeCaL</i>	Base class for all neural network modules.
<i>DualE</i>	Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)
<i>Complex</i>	Base class for all neural network modules.
<i>AConEx</i>	Additive Convolutional Complex Knowledge Graph Embeddings
<i>AConvO</i>	Additive Convolutional Octonion Knowledge Graph Embeddings
<i>AConvQ</i>	Additive Convolutional Quaternion Knowledge Graph Embeddings
<i>ConvQ</i>	Convolutional Quaternion Knowledge Graph Embeddings
<i>ConvO</i>	Base class for all neural network modules.
<i>ConEx</i>	Convolutional Complex Knowledge Graph Embeddings
<i>QMult</i>	Base class for all neural network modules.
<i>OMult</i>	Base class for all neural network modules.
<i>Shallom</i>	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
<i>LFMult</i>	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
<i>CoKE</i>	Base class for all neural network modules.
<i>PykeenKGE</i>	A class for using knowledge graph embedding models implemented in Pykeen
<i>Byte</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>EnsembleKGE</i>	
<i>DICE_Trainer</i>	DICE_Trainer implement
<i>KGE</i>	Knowledge Graph Embedding Class for interactive usage of pre-trained models
<i>BPE_NegativeSamplingDataset</i>	An abstract class representing a Dataset.
<i>MultiLabelDataset</i>	An abstract class representing a Dataset.
<i>MultiClassClassificationDataset</i>	Dataset for the 1vsALL training strategy
<i>OnevsAllDataset</i>	Dataset for the 1vsALL training strategy
<i>KvsAll</i>	Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.
<i>AllvsAll</i>	Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.
<i>OnevsSample</i>	A custom PyTorch Dataset class for knowledge graph embeddings, which includes
<i>KvsSampleDataset</i>	KvsSample a Dataset:
<i>NegSampleDataset</i>	An abstract class representing a Dataset.
<i>TriplePredictionDataset</i>	Triple Dataset

continues on next page

Table 3 – continued from previous page

<i>CVDataModule</i>	Create a Dataset for cross validation
<i>LiteralDataset</i>	Dataset for loading and processing literal data for training Literal Embedding model.
<i>QueryGenerator</i>	

14.4 Functions

<i>create_recipriocal_triples</i> (x)	Add inverse triples into dask dataframe
<i>get_er_vocab</i> (data[, file_path])	
<i>get_re_vocab</i> (data[, file_path])	
<i>get_ee_vocab</i> (data[, file_path])	
<i>timeit</i> (func)	
<i>save_pickle</i> (*[, data, file_path])	
<i>load_pickle</i> ([file_path])	
<i>load_term_mapping</i> ([file_path])	
<i>select_model</i> (args[, is_continual_training, storage_path])	
<i>load_model</i> (→ Tuple[object, Tuple[dict, dict]])	Load weights and initialize pytorch module from namespace arguments
<i>load_model_ensemble</i> (...)	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
<i>save_numpy_ndarray</i> (*[, data, file_path])	
<i>numpy_data_type_changer</i> (→ numpy.ndarray)	Detect most efficient data type for a given triples
<i>save_checkpoint_model</i> (→ None)	Store Pytorch model into disk
<i>store</i> (→ None)	
<i>add_noisy_triples</i> (→ pandas.DataFrame)	Add randomly constructed triples
<i>read_or_load_kg</i> (args, cls)	
<i>intialize_model</i> (→ Tuple[object, str])	
<i>load_json</i> (→ dict)	
<i>save_embeddings</i> (→ None)	Save it as CSV if memory allows.
<i>random_prediction</i> (pre_trained_kge)	
<i>deploy_triple_prediction</i> (pre_trained_kge, str_subject, ...)	
<i>deploy_tail_entity_prediction</i> (pre_trained_kge, ...)	

continues on next page

Table 4 – continued from previous page

<code>deploy_head_entity_prediction(pre_trained_kge,</code>	
<code>...)</code>	
<code>deploy_relation_prediction(pre_trained_kge,</code>	
<code>...)</code>	
<code>vocab_to_parquet(vocab_to_idx, name, ...)</code>	
<code>create_experiment_folder([folder_name])</code>	
<code>continual_training_setup_executor(→ None)</code>	
<code>exponential_function(→ torch.FloatTensor)</code>	
<code>load_numpy(→ numpy.ndarray)</code>	
<code>evaluate(entity_to_idx, scores, easy_answers, # @TODO: CD: Renamed this function</code>	
<code>hard_answers)</code>	
<code>download_file(url[, destination_folder])</code>	
<code>download_files_from_url(→ None)</code>	
<code>download_pretrained_model(→ str)</code>	
<code>write_csv_from_model_parallel(path)</code>	Create
<code>from_pretrained_model_write_embeddings_into</code>	
<code>None)</code>	
<code>mapping_from_first_two_cols_to_third(train_se</code>	
<code>timeit(func)</code>	
<code>load_term_mapping([file_path])</code>	
<code>reload_dataset(path, form_of_labelling, ...)</code>	Reload the files from disk to construct the Pytorch dataset
<code>construct_dataset(→ torch.utils.data.Dataset)</code>	

14.5 Package Contents

class `dicee.Pyke` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

A Physical Embedding Model for Knowledge Graphs

name = 'Pyke'

dist_func

margin = 1.0

forward_triples (*x: torch.LongTensor*)

Parameters

x

```
class dicee.DistMult (args)
    Bases: dicee.models.base_model.BaseKGE

    Embedding Entities and Relations for Learning and Inference in Knowledge Bases https://arxiv.org/abs/1412.6575

    name = 'DistMult'

    k_vs_all_score (emb_h: torch.FloatTensor, emb_r: torch.FloatTensor, emb_E: torch.FloatTensor)

        Parameters
            • emb_h
            • emb_r
            • emb_E

    forward_k_vs_all (x: torch.LongTensor)

    forward_k_vs_sample (x: torch.LongTensor, target_entity_idx: torch.LongTensor)

    score (h, r, t)
```

```
class dicee.CKeci (args)
    Bases: Keci

    Without learning dimension scaling

    name = 'CKeci'

    requires_grad_for_interactions = False
```

```
class dicee.Keci (args)
    Bases: dicee.models.base_model.BaseKGE
```

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training (bool)` – Boolean represents whether this module is in training or evaluation mode.

`name = 'Keci'`

`p`

`q`

`r`

`requires_grad_for_interactions = True`

`compute_sigma_pp (hp, rp)`

Compute $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_{ir_k} - h_{kr_i}) e_i e_k$

σ_{pp} captures the interactions between along p bases For instance, let $p = e_1, e_2, e_3$, we compute interactions between $e_1 e_2, e_1 e_3$, and $e_2 e_3$ This can be implemented with a nested two for loops

`results = []` for `i` in `range(p - 1)`:

for `k` in `range(i + 1, p)`:

`results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])`

`sigma_pp = torch.stack(results, dim=2)` assert `sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))`

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., $e_1 e_1, e_1 e_2, e_1 e_3$,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals: $e_1 e_2, e_1 e_3, e_2 e_3$.

`compute_sigma_qq (hq, rq)`

Compute $\sigma_{qq} = \sum_{j=1}^{q-1} \sum_{k=j+1}^q (h_{jr_k} - h_{kr_j}) e_j e_k$ σ_{qq} captures the interactions between along q bases For instance, let $q = e_1, e_2, e_3$, we compute interactions between $e_1 e_2, e_1 e_3$, and $e_2 e_3$ This can be implemented with a nested two for loops

`results = []` for `j` in `range(q - 1)`:

for `k` in `range(j + 1, q)`:

`results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])`

`sigma_qq = torch.stack(results, dim=2)` assert `sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))`

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., $e_1 e_1, e_1 e_2, e_1 e_3$,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals: $e_1 e_2, e_1 e_3, e_2 e_3$.

`compute_sigma_pq (*, hp, hq, rp, rq)`

$\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_{ir_j} - h_{jr_i}) e_i e_j$

`results = []` `sigma_pq = torch.zeros(b, r, p, q)` for `i` in `range(p)`:

```

    for j in range(q):
        sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
    print(sigma_pq.shape)
apply_coefficients (hp, hq, rp, rq)
    Multiplying a base vector with its scalar coefficient
clifford_multiplication (h0, hp, hq, r0, rp, rq)
    Compute our CL multiplication
    
$$h = h_0 + \sum_{i=1}^p h_i e_i + \sum_{j=p+1}^{p+q} h_j e_j$$


$$r = r_0 + \sum_{i=1}^p r_i e_i + \sum_{j=p+1}^{p+q} r_j e_j$$


$$e_i^2 = +1 \text{ for } i \leq p, e_j^2 = -1 \text{ for } p < j \leq p+q, e_i e_j = -e_j e_i \text{ for } i \neq j$$


$$h r = \sigma_0 + \sigma_p + \sigma_q + \sigma_{pp} + \sigma_{qq} + \sigma_{pq}$$

    where
    (1)  $\sigma_0 = h_0 r_0 + \sum_{i=1}^p (h_i r_i) e_i - \sum_{j=p+1}^{p+q} (h_j r_j) e_j$ 
    (2)  $\sigma_p = \sum_{i=1}^p (h_i r_0 + h_0 r_i) e_i$ 
    (3)  $\sigma_q = \sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j$ 
    (4)  $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k$ 
    (5)  $\sigma_{qq} = \sum_{j=1}^{q-1} \sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k$ 
    (6)  $\sigma_{pq} = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$ 
construct_cl_multivector (x: torch.FloatTensor, r: int, p: int, q: int)
    → tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]
    Construct a batch of multivectors  $Cl_{\{p,q\}}(\mathbb{R}^d)$ 

```

Parameter

x: torch.FloatTensor with (n,d) shape

returns

- **a0** (torch.FloatTensor with (n,r) shape)
- **ap** (torch.FloatTensor with (n,r,p) shape)
- **aq** (torch.FloatTensor with (n,r,q) shape)

```

forward_k_vs_with_explicit (x: torch.Tensor)
k_vs_all_score (bpe_head_ent_emb, bpe_rel_ent_emb, E)
forward_k_vs_all (x: torch.Tensor) → torch.FloatTensor
    Kvsall training

```

- (1) Retrieve real-valued embedding vectors for heads and relations \mathbb{R}^d .
- (2) Construct head entity and relation embeddings according to $Cl_{\{p,q\}}(\mathbb{R}^d)$.
- (3) Perform CL multiplication
- (4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this functions are identical **Parameter** ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, **IEI**) shape

construct_batch_selected_cl_multivector (*x: torch.FloatTensor, r: int, p: int, q: int*)
 → tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of batchs multivectors $CL_{\{p,q\}}(\mathbb{R}^d)$

Parameter

x: torch.FloatTensor with (n,k, d) shape

returns

- **a0** (*torch.FloatTensor with (n,k, m) shape*)
- **ap** (*torch.FloatTensor with (n,k, m, p) shape*)
- **aq** (*torch.FloatTensor with (n,k, m, q) shape*)

forward_k_vs_sample (*x: torch.LongTensor, target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,2) shape

target_entity_idx: torch.LongTensor with (n, k) shape k denotes the selected number of examples.

rtype

torch.FloatTensor with (n, k) shape

score (*h, r, t*)

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,3) shape

rtype

torch.FloatTensor with (n) shape

class `dicee.TransE` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Translating Embeddings for Modeling Multi-relational Data <https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf>

name = 'TransE'

margin = 4

score (*head_ent_emb, rel_ent_emb, tail_ent_emb*)

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

class `dicee.DeCaL` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

name = 'DeCaL'

entity_embeddings

relation_embeddings

p

q

r

re

forward_triples (*x: torch.Tensor*) \rightarrow torch.FloatTensor

Parameter

x: torch.LongTensor with (n,) shape

rtype

torch.FloatTensor with (n) shape

cl_pqr (*a: torch.tensor*) \rightarrow torch.tensor

Input: tensor(batch_size, emb_dim) \rightarrow output: tensor with 1+p+q+r components with size (batch_size, emb_dim/(1+p+q+r)) each.

1) takes a tensor of size (batch_size, emb_dim), split it into 1 + p + q + r components, hence 1+p+q+r must be a divisor of the emb_dim. 2) Return a list of the 1+p+q+r components vectors, each are tensors of size (batch_size, emb_dim/(1+p+q+r))

compute_sigmas_single (*list_h_emb, list_r_emb, list_t_emb*)

here we compute all the sums with no others vectors interaction taken with the scalar product with t, that is,

$$s0 = h_0 r_0 t_0 s1 = \sum_{i=1}^p h_i r_i t_0 s2 = \sum_{j=p+1}^{p+q} h_j r_j t_0 s3 = \sum_{i=1}^q (h_0 r_i t_i + h_i r_0 t_i) s4 = \sum_{i=p+1}^{p+q} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+1}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i)$$

and return:

$$sigma_0 t = \sigma_0 \cdot t_0 = s0 + s1 - s2 s3, s4 \text{ and } s5$$

compute_sigmas_multivect (*list_h_emb, list_r_emb*)

Here we compute and return all the sums with vectors interaction for the same and different bases.

For same bases vectors interaction we have

$$\sigma_p p = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (h_i r_{i'} - h_{i'} r_i) (model \text{ the interactions between } e_i \text{ and } e_{i'} \text{ for } 1 \leq i, i' \leq p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{j'} r_j)$$

For different base vector interactions, we have

$$\sigma_p q = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) (interactions \text{ between } e_i \text{ and } e_j \text{ for } 1 \leq i \leq p \text{ and } p+1 \leq j \leq p+q) \sigma_p r = \sum_{i=1}^p \sum_{r=p+q+1}^{p+q+r} (h_i r_r - h_r r_i)$$

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations
- (2) Construct head entity and relation embeddings according to $Cl_{\{p,q,r\}}(\mathbb{R}^d)$.
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this functions are identical Parameter ——— x: torch.LongTensor with (n,) shape :rtype: torch.FloatTensor with (n, **IE**) shape

apply_coefficients (*h0, hp, hq, hk, r0, rp, rq, rk*)

Multiplying a base vector with its scalar coefficient

construct_cl_multivector (*x: torch.FloatTensor, re: int, p: int, q: int, r: int*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors $Cl_{\{p,q,r\}}(\mathbb{R}^d)$

Parameter

x: torch.FloatTensor with (n,d) shape

returns

- **a0** (*torch.FloatTensor*)
- **ap** (*torch.FloatTensor*)
- **aq** (*torch.FloatTensor*)
- **ar** (*torch.FloatTensor*)

compute_sigma_pp (*hp, rp*)

Compute .. math:

$$\sigma_{p,p} = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (x_i y_{i'} - x_{i'} y_i)$$

σ_{pp} captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

for k in range(i + 1, p):

 results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

compute_sigma_qq (*hq, rq*)

Compute

$$\sigma_{q,q}^* = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (x_j y_{j'} - x_{j'} y_j) Eq.16$$

σ_{qq} captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

for k in range(j + 1, q):

 results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., e1e1, e1e2, e1e3,

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

Then select the triangular matrix without diagonals: e1e2, e1e3, e2e3.

compute_sigma_rr (*hk, rk*)

$$\sigma_{r,r}^* = \sum_{k=p+q+1}^{p+q+r-1} \sum_{k'=k+1}^p (x_k y_{k'} - x_{k'} y_k)$$

compute_sigma_pq (*, *hp, hq, rp, rq*)

Compute

$$\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$$

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):

 sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

```

print(sigma_pq.shape)
compute_sigma_pr (*, hp, hk, rp, rk)
    Compute

```

$$\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$$

```

results = []
sigma_pq = torch.zeros(b, r, p, q)
for i in range(p):
    for j in range(q):
        sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
print(sigma_pq.shape)
compute_sigma_qr (*, hq, hk, rq, rk)

```

$$\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$$

```

results = []
sigma_pq = torch.zeros(b, r, p, q)
for i in range(p):
    for j in range(q):
        sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
print(sigma_pq.shape)

class dicee.DualE (args)
    Bases: dicee.models.base_model.BaseKGE

    Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)

    name = 'DualE'

    entity_embeddings

    relation_embeddings

    num_ent = None

    kvsall_score (e_1_h, e_2_h, e_3_h, e_4_h, e_5_h, e_6_h, e_7_h, e_8_h, e_1_t, e_2_t, e_3_t, e_4_t,
        e_5_t, e_6_t, e_7_t, e_8_t, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8) → torch.tensor
        KvsAll scoring function

```

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_triples (idx_triple: torch.tensor) → torch.tensor

Negative Sampling forward pass:

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_k_vs_all(x)

KvsAll forward pass

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

T(x: torch.tensor) → torch.tensor

Transpose function

Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)

class dicee.**Complex**(args)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

name = 'Complex'

static score (*head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor, tail_ent_emb: torch.FloatTensor*)

static k_vs_all_score (*emb_h: torch.FloatTensor, emb_r: torch.FloatTensor, emb_E: torch.FloatTensor*)

Parameters

- **emb_h**
- **emb_r**
- **emb_E**

forward_k_vs_all (*x: torch.LongTensor*) → torch.FloatTensor

forward_k_vs_sample (*x: torch.LongTensor, target_entity_idx: torch.LongTensor*)

class `dicee.AConEx` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Additive Convolutional Complex Knowledge Graph Embeddings

name = 'AConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

residual_convolution (*C_1: Tuple[torch.Tensor, torch.Tensor], C_2: Tuple[torch.Tensor, torch.Tensor]*) → torch.FloatTensor

Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Parameters

x

forward_k_vs_sample (*x: torch.Tensor, target_entity_idx: torch.Tensor*)

class `dicee.AConvO` (*args: dict*)

Bases: `dicee.models.base_model.BaseKGE`

Additive Convolutional Octonion Knowledge Graph Embeddings

```

name = 'AConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

static octonion_normalizer(emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4,
                           emb_rel_e5, emb_rel_e6, emb_rel_e7)

residual_convolution(O_1, O_2)

forward_triples(x: torch.Tensor) → torch.Tensor

    Parameters
    x

forward_k_vs_all(x: torch.Tensor)
    Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] =>
    [0.0,0.1,...,0.8], shape=> (1, |Entities|) Given a batch of head entities and relations => shape (size of batch,|
    Entities|)

class dicee.AConvQ(args)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional Quaternion Knowledge Graph Embeddings
    name = 'AConvQ'
    entity_embeddings
    relation_embeddings
    conv2d
    fc_num_input
    fc1
    bn_conv1
    bn_conv2
    feature_map_dropout
    residual_convolution(Q_1, Q_2)
    forward_triples(indexed_triple: torch.Tensor) → torch.Tensor

    Parameters
    x

```

forward_k_vs_all (*x*: *torch.Tensor*)

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **|Entities|**) Given a batch of head entities and relations => shape (size of batch, |Entities|)

class *dicee.ConvQ* (*args*)

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings

name = 'ConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution (*Q_1*, *Q_2*)

forward_triples (*indexed_triple*: *torch.Tensor*) → *torch.Tensor*

Parameters

x

forward_k_vs_all (*x*: *torch.Tensor*)

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **|Entities|**) Given a batch of head entities and relations => shape (size of batch, |Entities|)

class *dicee.ConvO* (*args*: *dict*)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)
```

(continues on next page)

```
def forward(self, x):
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

name = '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.ConEx (*args*)

Bases: *dicee.models.base_model.BaseKGE*

Convolutional ComplEx Knowledge Graph Embeddings

name = 'ConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

residual_convolution (*C_1*: *Tuple[torch.Tensor, torch.Tensor]*,
 C_2: *Tuple[torch.Tensor, torch.Tensor]*) → *torch.FloatTensor*

Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

forward_k_vs_all (*x*: *torch.Tensor*) → *torch.FloatTensor*

forward_triples (*x*: *torch.Tensor*) → *torch.FloatTensor*

Parameters

x

forward_k_vs_sample (*x*: *torch.Tensor*, *target_entity_idx*: *torch.Tensor*)

class *dicee.QMult* (*args*)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

`name = 'QMult'`

`explicit = True`

`quaternion_multiplication_followed_by_inner_product (h, r, t)`

Parameters

- **h** – shape: (*batch_dims, dim) The head representations.
- **r** – shape: (*batch_dims, dim) The head representations.
- **t** – shape: (*batch_dims, dim) The tail representations.

Returns

Triple scores.

`static quaternion_normalizer (x: torch.FloatTensor) → torch.FloatTensor`

Normalize the length of relation vectors, if the forward constraint has not been applied yet.

Absolute value of a quaternion

$$|a + bi + cj + dk| = \sqrt{a^2 + b^2 + c^2 + d^2}$$

L2 norm of quaternion vector:

$$\|x\|^2 = \sum_{i=1}^d |x_i|^2 = \sum_{i=1}^d (x_i.re^2 + x_i.im_1^2 + x_i.im_2^2 + x_i.im_3^2)$$

Parameters

x – The vector.

Returns

The normalized vector.

`score (head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor,
tail_ent_emb: torch.FloatTensor)`

`k_vs_all_score (bpe_head_ent_emb, bpe_rel_ent_emb, E)`

Parameters

- **bpe_head_ent_emb**
- **bpe_rel_ent_emb**
- **E**

`forward_k_vs_all (x)`

Parameters

x

`forward_k_vs_sample (x, target_entity_idx)`

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples,i.e., [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **lEntitiesl**) Given a batch of head entities and relations => shape (size of batch,l Entitiesl)

```
class dicee.OMult (args)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training (bool)` – Boolean represents whether this module is in training or evaluation mode.

```
name = 'OMult'
```

```
static octonion_normalizer(emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4,
                           emb_rel_e5, emb_rel_e6, emb_rel_e7)
```

```
score(head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor,
      tail_ent_emb: torch.FloatTensor)
```

```
k_vs_all_score(bpe_head_ent_emb, bpe_rel_ent_emb, E)
```

```
forward_k_vs_all(x)
```

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples,i.e., `[score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8]`, shape=> (1, **Entities**) Given a batch of head entities and relations => shape (size of batch,| Entities|)

```
class dicee.Shallom (args)
```

Bases: `dicee.models.base_model.BaseKGE`

A shallow neural model for relation prediction (<https://arxiv.org/abs/2101.09090>)

```
name = 'Shallom'
```

shallow

get_embeddings () → Tuple[numpy.ndarray, None]

forward_k_vs_all (x) → torch.FloatTensor

forward_triples (x) → torch.FloatTensor

Parameters

x

Returns

class dicee.LFMult (args)

Bases: *dicee.models.base_model.BaseKGE*

Embedding with polynomial functions. We represent all entities and relations in the polynomial space as: $f(x) = \sum_{i=0}^{d-1} a_i x^i$ and use the three different scoring function as in the paper to evaluate the score. We also consider combining with Neural Networks.

name = 'LFMult'

entity_embeddings

relation_embeddings

degree

m

x_values

forward_triples (idx_triple)

Parameters

x

construct_multi_coeff (x)

poly_NN (x, coefh, coefr, coeft)

Constructing a 2 layers NN to represent the embeddings. $h = \sigma(w_h^T x + b_h)$, $r = \sigma(w_r^T x + b_r)$, $t = \sigma(w_t^T x + b_t)$

linear (x, w, b)

scalar_batch_NN (a, b, c)

element wise multiplication between a,b and c: Inputs : a, b, c ==> torch.tensor of size batch_size x m x d
Output : a tensor of size batch_size x d

tri_score (coeff_h, coeff_r, coeff_t)

this part implement the trilinear scoring techniques:

$$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*b_j*c_k\}\{1+(i+j+k)\%d\}$$

1. generate the range for i,j and k from [0 d-1]
2. perform $\text{dfrac}\{a_i*b_j*c_k\}\{1+(i+j+k)\%d\}$ in parallel for every batch
3. take the sum over each batch

vtp_score (*h, r, t*)

this part implement the vector triple product scoring techniques:

$$\text{score}(h,r,t) = \int_{\{0\}^{\{1\}}} h(x)r(x)t(x) \, dx = \sum_{\{i,j,k = 0\}^{d-1}} \text{dfrac}\{a_i*c_j*b_k - b_i*c_j*a_k\}\{(1+(i+j)\%d)(1+k)\}$$

1. generate the range for i,j and k from [0 d-1]
2. Compute the first and second terms of the sum
3. Multiply with then denominator and take the sum
4. take the sum over each batch

comp_func (*h, r, t*)

this part implement the function composition scoring techniques: i.e. score = <hor, t>

polynomial (*coeff, x, degree*)

This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,...d] and return a vector tensor (coeff[0][0] + coeff[0][1]x +...+ coeff[0][d]x^d,

$$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d)$$

pop (*coeff, x, degree*)

This function allow us to evaluate the composition of two polynomes without for loops :) it takes a matrix tensor of coefficients (coeff), a matrix tensor of points x and range of integer [0,1,...d]

$$\text{and return a tensor } (\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d,$$

$$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d)$$

class `dicee.CoKE` (*args, config: CoKEConfig = CoKEConfig()*)

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training (bool)` – Boolean represents whether this module is in training or evaluation mode.

`name = 'CoKE'`

`config`

`pos_emb`

`mask_emb`

`blocks`

`ln_f`

`coke_dropout`

`forward_k_vs_all (x: torch.Tensor)`

`score (emb_h, emb_r, emb_t)`

`forward_k_vs_sample (x: torch.LongTensor, target_entity_idx: torch.LongTensor)`

`class dicee.PykeenKGE (args: dict)`

Bases: `dicee.models.base_model.BaseKGE`

A class for using knowledge graph embedding models implemented in Pykeen

Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Pykeen_HolE: Pykeen_HolE: Pykeen_TransD: Pykeen_TransE: Pykeen_TransF: Pykeen_TransH: Pykeen_TransR:

`model_kwargs`

`name`

`model`

`loss_history = []`

`args`

`entity_embeddings = None`

`relation_embeddings = None`

`forward_k_vs_all (x: torch.LongTensor)`

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads and relations + apply Dropout & Normalization if given. h, r = self.get_head_relation_representation(x) # (2) Reshape (1). if self.last_dim > 0:

h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim)

```

# (3) Reshape all entities. if self.last_dim > 0:
    t = self.entity_embeddings.weight.reshape(self.num_entities, self.embedding_dim, self.last_dim)

else:
    t = self.entity_embeddings.weight

# (4) Call the score_t from interactions to generate triple scores. return self.interaction.score_t(h=h, r=r,
all_entities=t, slice_size=1)

forward_triples (x: torch.LongTensor) → torch.FloatTensor
# => Explicit version by this we can apply bn and dropout

# (1) Retrieve embeddings of heads, relations and tails and apply Dropout & Normalization if given. h, r, t =
self.get_triple_representation(x) # (2) Reshape (1). if self.last_dim > 0:
    h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim,
    self.last_dim) t = t.reshape(len(x), self.embedding_dim, self.last_dim)

# (3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice_size=None, slice_dim=0)

abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)

```

```
class dicee.ByteE (*args, **kwargs)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

```

name = 'BytE'

config

temperature = 0.5

topk = 2

transformer

lm_head

loss_function(yhat_batch, y_batch)

```

Parameters

- **yhat_batch**
- **y_batch**

forward (*x: torch.LongTensor*)

Parameters

x (*B by T tensor*)

generate (*idx, max_new_tokens, temperature=1.0, top_k=None*)

Take a conditioning sequence of indices *idx* (LongTensor of shape (b,t)) and complete the sequence *max_new_tokens* times, feeding the predictions back into the model each time. Most likely you'll want to make sure to be in `model.eval()` mode of operation for this.

training_step (*batch, batch_idx=None*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** – The output of your data iterable, normally a `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple dataloaders used)

Returns

- **Tensor** - The loss tensor
- **dict** - A dictionary which can include any keys, but must include the key `'loss'` in the case of automatic optimization.
- **None** - In automatic optimization, this will skip to the next batch (but is not supported for multi-GPU, TPU, or DeepSpeed). For manual optimization, this has no special meaning, as returning the loss is not required.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```

def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss

```

To use multiple optimizers, you can switch to ‘manual optimization’ and control their stepping:

```
def __init__(self):
    super().__init__()
    self.automatic_optimization = False

# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

    # do training_step with encoder
    ...
    opt1.step()
    # do training_step with decoder
    ...
    opt2.step()
```

Note

When `accumulate_grad_batches > 1`, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

class `dicee.BaseKGE` (*args: dict*)

Bases: `BaseKGE Lightning`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the

child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

```

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

    Parameters
         $\mathbf{x}$  ( $B \times 2 \times T$ )

forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors

    Parameters
        -----

init_params_with_sanity_checking()

forward (x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
        y_idx: torch.LongTensor = None)

    Parameters
        •  $\mathbf{x}$ 
        •  $\mathbf{y\_idx}$ 
        • ordered_bpe_entities

forward_triples (x: torch.LongTensor) → torch.Tensor

    Parameters
         $\mathbf{x}$ 

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

get_sentence_representation (x: torch.LongTensor)

    Parameters
        • ( $\mathbf{b}$  ( $x$  shape)
        • 3
        •  $\mathbf{t}$ )

get_bpe_head_and_relation_representation (x: torch.LongTensor)
    → Tuple[torch.FloatTensor, torch.FloatTensor]

    Parameters
         $\mathbf{x}$  ( $B \times 2 \times T$ )

get_embeddings () → Tuple[numpy.ndarray, numpy.ndarray]

class dicee.EnsembleKGE (models: list = None, seed_model=None, pretrained_models: List = None)

    name

```

```

train_mode = True

args

named_children()

property example_input_array

parameters()

modules()

__iter__()

__len__()

eval()

to(device)

state_dict()
    Return the state dict of the ensemble.

load_state_dict(state_dict, strict=True)
    Load the state dict into the ensemble.

mem_of_model()

__call__(x_batch)

step()

get_embeddings()

__str__()

dicee.create_recipriocal_triples(x)
    Add inverse triples into dask dataframe :param x: :return:

dicee.get_er_vocab(data, file_path: str = None)

dicee.get_re_vocab(data, file_path: str = None)

dicee.get_ee_vocab(data, file_path: str = None)

dicee.timeit(func)

dicee.save_pickle(*, data: object = None, file_path=str)

dicee.load_pickle(file_path=str)

dicee.load_term_mapping(file_path=str)

dicee.select_model(args: dict, is_continual_training: bool = None, storage_path: str = None)

dicee.load_model(path_of_experiment_folder: str, model_name='model.pt', verbose=0)
    → Tuple[object, Tuple[dict, dict]]
    Load weights and initialize pytorch module from namespace arguments

```

```

dicee.load_model_ensemble(path_of_experiment_folder: str)
    → Tuple[dicee.models.base_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]
    Construct Ensemble Of weights and initialize pytorch module from namespace arguments
    (1) Detect models under given path
    (2) Accumulate parameters of detected models
    (3) Normalize parameters
    (4) Insert (3) into model.

dicee.save_numpy_ndarray(*, data: numpy.ndarray, file_path: str)

dicee.numpy_data_type_changer(train_set: numpy.ndarray, num: int) → numpy.ndarray
    Detect most efficient data type for a given triples :param train_set: :param num: :return:

dicee.save_checkpoint_model(model, path: str) → None
    Store Pytorch model into disk

dicee.store(trained_model, model_name: str = 'model', full_storage_path: str = None,
    save_embeddings_as_csv=False) → None

dicee.add_noisy_triples(train_set: pandas.DataFrame, add_noise_rate: float) → pandas.DataFrame
    Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

dicee.read_or_load_kg(args, cls)

dicee.intialize_model(args: dict, verbose=0) → Tuple[object, str]

dicee.load_json(p: str) → dict

dicee.save_embeddings(embeddings: numpy.ndarray, indexes, path: str) → None
    Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:

dicee.random_prediction(pre_trained_kge)

dicee.deploy_triple_prediction(pre_trained_kge, str_subject, str_predicate, str_object)

dicee.deploy_tail_entity_prediction(pre_trained_kge, str_subject, str_predicate, top_k)

dicee.deploy_head_entity_prediction(pre_trained_kge, str_object, str_predicate, top_k)

dicee.deploy_relation_prediction(pre_trained_kge, str_subject, str_object, top_k)

dicee.vocab_to_parquet(vocab_to_idx, name, path_for_serialization, print_into)

dicee.create_experiment_folder(folder_name='Experiments')

dicee.continual_training_setup_executor(executor) → None

dicee.exponential_function(x: numpy.ndarray, lam: float, ascending_order=True) → torch.FloatTensor

dicee.load_numpy(path) → numpy.ndarray

dicee.evaluate(entity_to_idx, scores, easy_answers, hard_answers)
    # @TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

dicee.download_file(url, destination_folder='')

```

`dicee.download_files_from_url (base_url: str, destination_folder='.') → None`

Parameters

- **base_url** (e.g. [“https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll”](https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll))
- **destination_folder** (e.g. `"KINSHIP-Keci-dim128-epoch256-KvsAll"`)

`dicee.download_pretrained_model (url: str) → str`

`dicee.write_csv_from_model_parallel (path: str)`

Create

`dicee.from_pretrained_model_write_embeddings_into_csv (path: str) → None`

class `dicee.DICE_Trainer (args, is_continual_training: bool, storage_path, evaluator=None)`

DICE_Trainer implement

- 1- Pytorch Lightning trainer (<https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html>)
- 2- Multi-GPU Trainer(<https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>)
- 3- CPU Trainer

args

is_continual_training:bool

storage_path:str

evaluator:

report:dict

report

args

trainer = None

is_continual_training

storage_path

evaluator = None

form_of_labelling = None

continual_start (*knowledge_graph*)

(1) Initialize training.

(2) Load model

(3) Load trainer (3) Fit model

Parameter

returns

- *model*
- **form_of_labelling** (*str*)

initialize_trainer (*callbacks: List*)
→ *lightning.Trainer* | *dicee.trainer.model_parallelism.TensorParallel* | *dicee.trainer.torch_trainer.TorchTrainer* | *dicee.*

Initialize Trainer from input arguments

initialize_or_load_model ()

init_dataloader (*dataset: torch.utils.data.Dataset*) → *torch.utils.data.DataLoader*

init_dataset () → *torch.utils.data.Dataset*

start (*knowledge_graph: dicee.knowledge_graph.KG* | *numpy.memmap*)
→ *Tuple[dicee.models.base_model.BaseKGE, str]*

Start the training

(1) Initialize Trainer

(2) Initialize or load a pretrained KGE model

in DDP setup, we need to load the memory map of already read/index KG.

k_fold_cross_validation (*dataset*) → *Tuple[dicee.models.base_model.BaseKGE, str]*

Perform K-fold Cross-Validation

1. Obtain K train and test splits.

2. **For each split,**

2.1 initialize trainer and model 2.2. Train model with configuration provided in args. 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.

3. Report the mean and average MRR .

Parameters

- **self**
- **dataset**

Returns

model

class *dicee.KGE* (*path=None, url=None, construct_ensemble=False, model_name=None*)

Bases: *dicee.abstracts.BaseInteractiveKGE*, *dicee.abstracts.InteractiveQueryDecomposition*, *dicee.abstracts.BaseInteractiveTrainKGE*

Knowledge Graph Embedding Class for interactive usage of pre-trained models

__str__ ()

to (*device: str*) → None

get_transductive_entity_embeddings (*indices: torch.LongTensor* | *List[str]*, *as_pytorch=False*,
as_numpy=False, as_list=True) → *torch.FloatTensor* | *numpy.ndarray* | *List[float]*

create_vector_database (*collection_name: str, distance: str, location: str = 'localhost',*
port: int = 6333)

generate (*h="", r=""*)

eval_lp_performance (*dataset=List[Tuple[str, str, str]]*, *filtered=True*)

predict_missing_head_entity (*relation: List[str] | str, tail_entity: List[str] | str, within=None, batch_size=2, topk=1, return_indices=False*) → Tuple

Given a relation and a tail entity, return top k ranked head entity.

$\text{argmax}_{\{e \in E\}} f(e, r, t)$, where $r \in R, t \in E$.

Parameter

relation: Union[List[str], str]

String representation of selected relations.

tail_entity: Union[List[str], str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_relations (*head_entity: List[str] | str, tail_entity: List[str] | str, within=None, batch_size=2, topk=1, return_indices=False*) → Tuple

Given a head entity and a tail entity, return top k ranked relations.

$\text{argmax}_{\{r \in R\}} f(h, r, t)$, where $h, t \in E$.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_tail_entity (*head_entity: List[str] | str, relation: List[str] | str, within: List[str] = None, batch_size=2, topk=1, return_indices=False*) → torch.FloatTensor

Given a head entity and a relation, return top k ranked entities

$\text{argmax}_{\{e \in E\}} f(h, r, e)$, where $h \in E$ and $r \in R$.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

scores

predict (*, *h*: List[str] | str = None, *r*: List[str] | str = None, *t*: List[str] | str = None, *within*=None, *logits*=True) → torch.FloatTensor

Parameters

- **logits**
- **h**
- **r**
- **t**
- **within**

predict_topk (*, *h*: str | List[str] = None, *r*: str | List[str] = None, *t*: str | List[str] = None, *topk*: int = 10, *within*: List[str] = None, *batch_size*: int = 1024)

Predict missing item in a given triple.

Returns

- If you query a single (h, r, ?) or (?, r, t) or (h, ?, t), returns List[(item, score)]
- If you query a batch of B, returns List of B such lists.

triple_score (*h*: List[str] | str = None, *r*: List[str] | str = None, *t*: List[str] | str = None, *logits*=False) → torch.FloatTensor

Predict triple score

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

Returns: Tuple

pytorch tensor of triple score

return_multi_hop_query_results (*aggregated_query_for_all_entities*, *k*: int, *only_scores*)

single_hop_query_answering (*query*: tuple, *only_scores*: bool = True, *k*: int = None)

answer_multi_hop_query (*query_type*: str = None, *query*: Tuple[str | Tuple[str, str], Ellipsis] = None, *queries*: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, *tnorm*: str = 'prod', *neg_norm*: str = 'standard', *lambda_*: float = 0.0, *k*: int = 10, *only_scores*=False) → List[Tuple[str, torch.Tensor]]

@TODO: Refactoring is needed # @TODO: Score computation for each query type should be done in a static function

Find an answer set for EPFO queries including negation and disjunction

Parameter

query_type: str The type of the query, e.g., “2p”.

query: Union[str, Tuple[str, Tuple[str, str]]] The query itself, either a string or a nested tuple.

queries: List of Tuple[Union[str, Tuple[str, str]], ...]

tnorm: str The t-norm operator.

neg_norm: str The negation norm.

lambda_: float lambda parameter for sugeno and yager negation norms

k: int The top-k substitutions for intermediate variables.

returns

- *List[Tuple[str, torch.Tensor]]*
- *Entities and corresponding scores sorted in the descening order of scores*

find_missing_triples (confidence: float, entities: List[str] = None, relations: List[str] = None, topk: int = 10, at_most: int = sys.maxsize) → Set

Find missing triples

Iterative over a set of entities E and a set of relation R :

forall e in E and forall r in R f(e,r,x)

Return (e,r,x)

return G and f(e,r,x) > confidence

confidence: float

A threshold for an output of a sigmoid function given a triple.

topk: int

Highest ranked k item to select triples with f(e,r,x) > confidence .

at_most: int

Stop after finding at_most missing triples

{(e,r,x) | f(e,r,x) > confidence and (e,r,x)}

return G

deploy (share: bool = False, top_k: int = 10)

predict_literals (entity: List[str] | str = None, attribute: List[str] | str = None, denormalize_preds: bool = True) → numpy.ndarray

Predicts literal values for given entities and attributes.

Parameters

- **entity** (Union[List[str], str]) – Entity or list of entities to predict literals for.

- **attribute** (*Union[List[str], str]*) – Attribute or list of attributes to predict literals for.
- **denormalize_preds** (*bool*) – If True, denormalizes the predictions.

Returns

Predictions for the given entities and attributes.

Return type

numpy ndarray

`dicee.mapping_from_first_two_cols_to_third(train_set_idx)`

`dicee.timeit(func)`

`dicee.load_term_mapping(file_path=str)`

`dicee.reload_dataset(path: str, form_of_labelling, scoring_technique, neg_ratio, label_smoothing_rate)`

Reload the files from disk to construct the Pytorch dataset

`dicee.construct_dataset(*, train_set: numpy.ndarray | list, valid_set=None, test_set=None, ordered_bpe_entities=None, train_target_indices=None, target_dim: int = None, entity_to_idx: dict, relation_to_idx: dict, form_of_labelling: str, scoring_technique: str, neg_ratio: int, label_smoothing_rate: float, byte_pair_encoding=None, block_size: int = None)`
→ torch.utils.data.Dataset

class `dicee.BPE_NegativeSamplingDataset(train_set: torch.LongTensor, ordered_shaped_bpe_entities: torch.LongTensor, neg_ratio: int)`

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

`DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

train_set

ordered_bpe_entities

num_bpe_entities

neg_ratio

num_datapoints

__len__()

__getitem__ (*idx*)

collate_fn (*batch_shaped_bpe_triples: List[Tuple[torch.Tensor, torch.Tensor]]*)

```
class dicee.MultiLabelDataset (train_set: torch.LongTensor, train_indices_target: torch.LongTensor,  
                                target_dim: int, torch_ordered_shaped_bpe_entities: torch.LongTensor)
```

Bases: `torch.utils.data.Dataset`

An abstract class representing a Dataset.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

`DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

`train_set`

`train_indices_target`

`target_dim`

`num_datapoints`

`torch_ordered_shaped_bpe_entities`

`collate_fn = None`

`__len__()`

`__getitem__(idx)`

```
class dicee.MultiClassClassificationDataset (subword_units: numpy.ndarray, block_size: int = 8)
```

Bases: `torch.utils.data.Dataset`

Dataset for the 1vsALL training strategy

Parameters

- `train_set_idx` – Indexed triples for the training.
- `entity_idx` – mapping.
- `relation_idx` – mapping.
- `form` – ?
- `num_workers` – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

Return type

`torch.utils.data.Dataset`

`train_data`

`block_size = 8`

`num_of_data_points`

```

collate_fn = None

__len__()

__getitem__(idx)

class dicee.OnevsAllDataset (train_set_idx: numpy.ndarray, entity_idxes)
    Bases: torch.utils.data.Dataset
    Dataset for the 1vsALL training strategy

    Parameters
        • train_set_idx – Indexed triples for the training.
        • entity_idxes – mapping.
        • relation_idxes – mapping.
        • form – ?
        • num_workers – int for https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

    Return type
        torch.utils.data.Dataset

    train_data

    target_dim

    collate_fn = None

    __len__()

    __getitem__(idx)

```

```

class dicee.KvsAll (train_set_idx: numpy.ndarray, entity_idxes, relation_idxes, form, store=None,
                    label_smoothing_rate: float = 0.0)
    Bases: torch.utils.data.Dataset

```

Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.

Let D denote a dataset for KvsAll training and be defined as $D := \{(x, y)_i\}_i^N$, where $x: (h, r)$ is a unique tuple of an entity h in E and a relation r in R that has been seed in the input graph. y : denotes a multi-label vector in $[0, 1]^{|E|}$ is a binary label.

orall $y_i = 1$ s.t. $(h \ r \ E_i)$ in KG

Note

TODO

train_set_idx
[numpy.ndarray] n by 3 array representing n triples

entity_idxes
[dictionary] string representation of an entity to its integer id

relation_idxes
[dictionary] string representation of a relation to its integer id

```
self : torch.utils.data.Dataset
```

```
>>> a = 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_idxes, relation_idxes, label_smoothing_rate=0.0)
```

```
Bases: torch.utils.data.Dataset
```

Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.

Let D denote a dataset for AllvsAll training and be defined as $D := \{(x, y)_i\}_i^N$, where $x: (h, r)$ is a possible unique tuple of an entity h in E and a relation r in R . Hence $N = |E| \times |R|$ y : denotes a multi-label vector in $[0, 1]^{|E|}$ is a binary label.

orall $y_i = 1$ s.t. $(h, r \in E_i)$ in KG

Note

AllvsAll extends KvsAll via none existing (h,r). Hence, it adds data points that are labelled without 1s, only with 0s.

train_set_idx

[numpy.ndarray] n by 3 array representing n triples

entity_idxes

[dictionary] string representation of an entity to its integer id

relation_idxes

[dictionary] string representation of a relation to its integer id

```
self : torch.utils.data.Dataset
```

```
>>> a = AllvsAll()
>>> a
? array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
train_data = None
```

```
train_target = None
```

```
label_smoothing_rate
```

```
collate_fn = None
```

target_dim

__len__()

__getitem__(*idx*)

```
class dicee.OnevsSample(train_set: numpy.ndarray, num_entities, num_relations,  
                        neg_sample_ratio: int = None, label_smoothing_rate: float = 0.0)
```

Bases: torch.utils.data.Dataset

A custom PyTorch Dataset class for knowledge graph embeddings, which includes both positive and negative sampling for a given dataset for multi-class classification problem..

Parameters

- **train_set** (*np.ndarray*) – A numpy array containing triples of knowledge graph data. Each triple consists of (head_entity, relation, tail_entity).
- **num_entities** (*int*) – The number of unique entities in the knowledge graph.
- **num_relations** (*int*) – The number of unique relations in the knowledge graph.
- **neg_sample_ratio** (*int, optional*) – The number of negative samples to be generated per positive sample. Must be a positive integer and less than num_entities.
- **label_smoothing_rate** (*float, optional*) – A label smoothing rate to apply to the positive and negative labels. Defaults to 0.0.

train_data

The input data converted into a PyTorch tensor.

Type

torch.Tensor

num_entities

Number of entities in the dataset.

Type

int

num_relations

Number of relations in the dataset.

Type

int

neg_sample_ratio

Ratio of negative samples to be drawn for each positive sample.

Type

int

label_smoothing_rate

The smoothing factor applied to the labels.

Type

torch.Tensor

collate_fn

A function that can be used to collate data samples into batches (set to None by default).

Type

function, optional

`train_data`

`num_entities`

`num_relations`

`neg_sample_ratio = None`

`label_smoothing_rate`

`collate_fn = None`

`__len__()`

Returns the number of samples in the dataset.

`__getitem__(idx)`

Retrieves a single data sample from the dataset at the given index.

Parameters

`idx` (*int*) – The index of the sample to retrieve.

Returns

A tuple consisting of:

- `x` (`torch.Tensor`): The head and relation part of the triple.
- `y_idx` (`torch.Tensor`): The concatenated indices of the true object (tail entity) and the indices of the negative samples.
- `y_vec` (`torch.Tensor`): A vector containing the labels for the positive and negative samples, with label smoothing applied.

Return type

tuple

```
class dicee.KvsSampleDataset (train_set_idx: numpy.ndarray, entity_idxes, relation_idxes, form,
                              store=None, neg_ratio=None, label_smoothing_rate: float = 0.0)
```

Bases: `torch.utils.data.Dataset`

KvsSample a Dataset:

$D := \{(x, y)_i\}_i^N$, where

. $x: (h, r)$ is a unique h in E and a relation r in R and . y in $[0, 1]^{|E|}$ is a binary label.

forall $y_i = 1$ s.t. (h, r, E_i) in KG

At each mini-batch construction, we subsample(y), hence n

$|new_y| \ll |E|$ new_y contains all 1's if $\sum(y) < neg_sample\ ratio$ new_y contains

`train_set_idx`

Indexed triples for the training.

`entity_idxes`

mapping.

`relation_idxes`

mapping.

`form`

?

```
store
?
label_smoothing_rate
?
```

```
torch.utils.data.Dataset
```

```
train_data = None
train_target = None
neg_ratio = None
num_entities
label_smoothing_rate
collate_fn = None
max_num_of_classes
__len__()
__getitem__(idx)
```

```
class dicee.NegSampleDataset (train_set: numpy.ndarray, num_entities: int, num_relations: int,
                               neg_sample_ratio: int = 1)
```

```
Bases: torch.utils.data.Dataset
```

An abstract class representing a Dataset.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

`DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

```
neg_sample_ratio
train_triples
length
num_entities
num_relations
labels
train_set = []
```

```

__len__()

__getitem__(idx)

class dicee.TriplePredictionDataset (train_set: numpy.ndarray, num_entities: int, num_relations: int,
    neg_sample_ratio: int = 1, label_smoothing_rate: float = 0.0)
    Bases: torch.utils.data.Dataset
        Triple Dataset
            D:= {(x_i)_i ^N, where
                . x:(h,r, t) in KG is a unique h in E and a relation r in R and . collect_fn => Generates
                negative triples
            collect_fn:
orall (h,r,t) in G obtain, create negative triples {(h,r,x),(r,t),(h,m,t)}
        y:labels are represented in torch.float16

    train_set_idx
        Indexed triples for the training.

    entity_idxxs
        mapping.

    relation_idxxs
        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.CVDDataModule (train_set_idx: numpy.ndarray, num_entities, num_relations, neg_sample_ratio,
    batch_size, num_workers)
    Bases: pytorch_lightning.LightningDataModule
        Create a Dataset for cross validation

```

Parameters

- **train_set_idx** – Indexed triples for the training.
- **num_entities** – entity to index mapping.
- **num_relations** – relation to index mapping.
- **batch_size** – int
- **form** – ?
- **num_workers** – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

Return type

?

train_set_idx

num_entities

num_relations

neg_sample_ratio

batch_size

num_workers

train_dataloader() → torch.utils.data.DataLoader

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:param-ref:~pytorch_lightning.trainer.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

- `fit()`
- `prepare_data()`
- `setup()`

Note

Lightning tries to add the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

setup(*args, **kwargs)

Called at the beginning of fit (train + validate), validate, test, or predict. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

Parameters

stage – either 'fit', 'validate', 'test', or 'predict'

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

        # don't do this
        self.something = else

    def setup(self, stage):
        data = load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

transfer_batch_to_device(*args, **kwargs)

Override this hook if your `DataLoader` returns tensors wrapped in a custom data structure.

The data types listed below (and any arbitrary nesting of them) are supported out of the box:

- `torch.Tensor` or anything that implements `.to(...)`
- `list`
- `dict`
- `tuple`

For anything else, you need to define how the data is moved to the target device (CPU, GPU, TPU, ...).

Note

This hook should only transfer the data and not modify it, nor should it move the data to any other device than the one passed in as argument (unless you know what you are doing). To check the current state of execution of this hook you can use `self.trainer.training/testing/validating/predicting` so that you can add different logic as per your requirement.

Parameters

- **batch** – A batch of data that needs to be transferred to a new device.

- **device** – The target device as defined in PyTorch.
- **dataloader_idx** – The index of the dataloader to which the batch belongs.

Returns

A reference to the data on the new device.

Example:

```
def transfer_batch_to_device(self, batch, device, dataloader_idx):
    if isinstance(batch, CustomBatch):
        # move all tensors in your custom data structure to the device
        batch.samples = batch.samples.to(device)
        batch.targets = batch.targets.to(device)
    elif dataloader_idx == 0:
        # skip device transfer for the first dataloader or anything you wish
        pass
    else:
        batch = super().transfer_batch_to_device(batch, device, dataloader_idx)
    return batch
```

See also

- `move_data_to_device()`
- `apply_to_collection()`

`prepare_data(*args, **kwargs)`

Use this to download and prepare data. Downloading and saving data with multiple processes (distributed settings) will result in corrupted data. Lightning ensures this method is called only within a single process, so you can safely add your downloading logic within.

Warning

DO NOT set state to the model (use `setup` instead) since this is NOT called on every device

Example:

```
def prepare_data(self):
    # good
    download_data()
    tokenize()
    etc()

    # bad
    self.split = data_split
    self.some_state = some_other_state()
```

In a distributed environment, `prepare_data` can be called in two ways (using `prepare_data_per_node`)

1. Once per node. This is the default and is only called on `LOCAL_RANK=0`.
2. Once in total. Only called on `GLOBAL_RANK=0`.

Example:

```
# DEFAULT
# called once per node on LOCAL_RANK=0 of that node
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = True

# call on GLOBAL_RANK=0 (great for shared file systems)
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = False
```

This is called before requesting the dataloaders:

```
model.prepare_data()
initialize_distributed()
model.setup(stage)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
model.predict_dataloader()
```

```
class dicee.LiteralDataset (file_path: str, ent_idx: dict = None, normalization_type: str = 'z-norm',
                           sampling_ratio: float = None, loader_backend: str = 'pandas')
```

Bases: torch.utils.data.Dataset

Dataset for loading and processing literal data for training Literal Embedding model. This dataset handles the loading, normalization, and preparation of triples for training a literal embedding model.

Extends torch.utils.data.Dataset for supporting PyTorch dataloaders.

train_file_path

Path to the training data file.

Type

str

normalization

Type of normalization to apply ('z-norm', 'min-max', or None).

Type

str

normalization_params

Parameters used for normalization.

Type

dict

sampling_ratio

Fraction of the training set to use for ablations.

Type

float

entity_to_idx

Mapping of entities to their indices.

Type

dict

num_entities

Total number of entities.

Type

int

data_property_to_idx

Mapping of data properties to their indices.

Type

dict

num_data_properties

Total number of data properties.

Type

int

loader_backend

Backend to use for loading data ('pandas' or 'rdflib').

Type

str

train_file_path

loader_backend = 'pandas'

normalization_type = 'z-norm'

normalization_params

sampling_ratio = None

entity_to_idx = None

num_entities

__getitem__ (*index*)

__len__ ()

static load_and_validate_literal_data (*file_path: str = None, loader_backend: str = 'pandas'*)
→ pandas.DataFrame

Loads and validates the literal data file. :param file_path: Path to the literal data file. :type file_path: str

Returns

DataFrame containing the loaded and validated data.

Return type

pd.DataFrame

static denormalize (*preds_norm, attributes, normalization_params*) → numpy.ndarray

Denormalizes the predictions based on the normalization type.

Args: *preds_norm* (np.ndarray): Normalized predictions to be denormalized. *attributes* (list): List of attributes corresponding to the predictions. *normalization_params* (dict): Dictionary containing normalization parameters for each attribute.

Returns

Denormalized predictions.

Return type

np.ndarray

class dicee.**QueryGenerator** (*train_path, val_path: str, test_path: str, ent2id: Dict = None, rel2id: Dict = None, seed: int = 1, gen_valid: bool = False, gen_test: bool = True*)

train_path

val_path

test_path

gen_valid = False

gen_test = True

seed = 1

max_ans_num = 1000000.0

mode

ent2id = None

rel2id: Dict = None

ent_in: Dict

ent_out: Dict

query_name_to_struct

list2tuple (*list_data*)

tuple2list (*x: List | Tuple*) → List | Tuple

Convert a nested tuple to a nested list.

set_global_seed (*seed: int*)

Set seed

construct_graph (*paths: List[str]*) → Tuple[Dict, Dict]

Construct graph from triples Returns dicts with incoming and outgoing edges

fill_query (*query_structure: List[str | List], ent_in: Dict, ent_out: Dict, answer: int*) → bool

Private method for fill_query logic.

achieve_answer (*query: List[str | List], ent_in: Dict, ent_out: Dict*) → set

Private method for achieve_answer logic. @TODO: Document the code

write_links (*ent_out, small_ent_out*)

```

ground_queries (query_structure: List[str | List], ent_in: Dict, ent_out: Dict, small_ent_in: Dict,
                 small_ent_out: Dict, gen_num: int, query_name: str)
    Generating queries and achieving answers

unmap (query_type, queries, tp_answers, fp_answers, fn_answers)

unmap_query (query_structure, query, id2ent, id2rel)

generate_queries (query_struct: List, gen_num: int, query_type: str)
    Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting
    queries and answers in return @ TODO: create a class for each single query struct

save_queries (query_type: str, gen_num: int, save_path: str)

abstract load_queries (path)

get_queries (query_type: str, gen_num: int)

static save_queries_and_answers (path: str, data: List[Tuple[str, Tuple[collections.defaultdict]]])
    → None
    Save Queries into Disk

static load_queries_and_answers (path: str) → List[Tuple[str, Tuple[collections.defaultdict]]]
    Load Queries from Disk to Memory

dicee.__version__ = '0.1.5'

```

Python Module Index

d

- [dicee](#), 12
- [dicee.__main__](#), 12
- [dicee.abstracts](#), 12
- [dicee.analyse_experiments](#), 19
- [dicee.callbacks](#), 20
- [dicee.config](#), 29
- [dicee.dataset_classes](#), 32
- [dicee.eval_static_funcs](#), 46
- [dicee.evaluator](#), 48
- [dicee.executer](#), 49
- [dicee.knowledge_graph](#), 51
- [dicee.knowledge_graph_embeddings](#), 53
- [dicee.models](#), 57
 - [dicee.models.adopt](#), 57
 - [dicee.models.base_model](#), 66
 - [dicee.models.clifford](#), 75
 - [dicee.models.complex](#), 82
 - [dicee.models.dualE](#), 84
 - [dicee.models.ensemble](#), 85
 - [dicee.models.function_space](#), 86
 - [dicee.models.literal](#), 90
 - [dicee.models.octonion](#), 91
 - [dicee.models.pykeen_models](#), 94
 - [dicee.models.quaternion](#), 96
 - [dicee.models.real](#), 99
 - [dicee.models.static_funcs](#), 102
 - [dicee.models.transformers](#), 102
- [dicee.query_generator](#), 161
- [dicee.read_preprocess_save_load_kg](#), 163
- [dicee.read_preprocess_save_load_kg.preprocess](#), 163
- [dicee.read_preprocess_save_load_kg.read_from_disk](#), 163
- [dicee.read_preprocess_save_load_kg.save_load_disk](#), 164
- [dicee.read_preprocess_save_load_kg.util](#), 165
- [dicee.sanity_checkers](#), 169
- [dicee.scripts](#), 170
 - [dicee.scripts.index_serve](#), 170
 - [dicee.scripts.run](#), 172
- [dicee.static_funcs](#), 172
- [dicee.static_funcs_training](#), 175
- [dicee.static_preprocess_funcs](#), 176
- [dicee.trainer](#), 177
 - [dicee.trainer.dice_trainer](#), 177
 - [dicee.trainer.model_parallelism](#), 179
 - [dicee.trainer.torch_trainer](#), 179
 - [dicee.trainer.torch_trainer_ddp](#), 181

Index

Non-alphabetical

`__call__` () (*dicee.EnsembleKGE method*), 211
`__call__` () (*dicee.models.base_model.IdentityClass method*), 75
`__call__` () (*dicee.models.ensemble.EnsembleKGE method*), 86
`__call__` () (*dicee.models.IdentityClass method*), 122, 135, 141
`__getitem__` () (*dicee.AllvsAll method*), 222
`__getitem__` () (*dicee.BPE_NegativeSamplingDataset method*), 218
`__getitem__` () (*dicee.dataset_classes.AllvsAll method*), 37
`__getitem__` () (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 34
`__getitem__` () (*dicee.dataset_classes.KvsAll method*), 36
`__getitem__` () (*dicee.dataset_classes.KvsSampleDataset method*), 39
`__getitem__` () (*dicee.dataset_classes.LiteralDataset method*), 46
`__getitem__` () (*dicee.dataset_classes.MultiClassClassificationDataset method*), 35
`__getitem__` () (*dicee.dataset_classes.MultiLabelDataset method*), 34
`__getitem__` () (*dicee.dataset_classes.NegSampleDataset method*), 40
`__getitem__` () (*dicee.dataset_classes.OnevsAllDataset method*), 35
`__getitem__` () (*dicee.dataset_classes.OnevsSample method*), 38
`__getitem__` () (*dicee.dataset_classes.TriplePredictionDataset method*), 41
`__getitem__` () (*dicee.KvsAll method*), 221
`__getitem__` () (*dicee.KvsSampleDataset method*), 224
`__getitem__` () (*dicee.LiteralDataset method*), 230
`__getitem__` () (*dicee.MultiClassClassificationDataset method*), 220
`__getitem__` () (*dicee.MultiLabelDataset method*), 219
`__getitem__` () (*dicee.NegSampleDataset method*), 225
`__getitem__` () (*dicee.OnevsAllDataset method*), 220
`__getitem__` () (*dicee.OnevsSample method*), 223
`__getitem__` () (*dicee.TriplePredictionDataset method*), 225
`__iter__` () (*dicee.config.Namespace method*), 32
`__iter__` () (*dicee.EnsembleKGE method*), 211
`__iter__` () (*dicee.knowledge_graph.KG method*), 53
`__iter__` () (*dicee.models.ensemble.EnsembleKGE method*), 86
`__len__` () (*dicee.AllvsAll method*), 222
`__len__` () (*dicee.BPE_NegativeSamplingDataset method*), 218
`__len__` () (*dicee.dataset_classes.AllvsAll method*), 37
`__len__` () (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 34
`__len__` () (*dicee.dataset_classes.KvsAll method*), 36
`__len__` () (*dicee.dataset_classes.KvsSampleDataset method*), 39
`__len__` () (*dicee.dataset_classes.LiteralDataset method*), 46
`__len__` () (*dicee.dataset_classes.MultiClassClassificationDataset method*), 35
`__len__` () (*dicee.dataset_classes.MultiLabelDataset method*), 34
`__len__` () (*dicee.dataset_classes.NegSampleDataset method*), 40
`__len__` () (*dicee.dataset_classes.OnevsAllDataset method*), 35
`__len__` () (*dicee.dataset_classes.OnevsSample method*), 38
`__len__` () (*dicee.dataset_classes.TriplePredictionDataset method*), 41
`__len__` () (*dicee.EnsembleKGE method*), 211
`__len__` () (*dicee.knowledge_graph.KG method*), 53
`__len__` () (*dicee.KvsAll method*), 221
`__len__` () (*dicee.KvsSampleDataset method*), 224
`__len__` () (*dicee.LiteralDataset method*), 230
`__len__` () (*dicee.models.ensemble.EnsembleKGE method*), 86
`__len__` () (*dicee.MultiClassClassificationDataset method*), 220
`__len__` () (*dicee.MultiLabelDataset method*), 219
`__len__` () (*dicee.NegSampleDataset method*), 224
`__len__` () (*dicee.OnevsAllDataset method*), 220
`__len__` () (*dicee.OnevsSample method*), 223
`__len__` () (*dicee.TriplePredictionDataset method*), 225
`__setstate__` () (*dicee.models.ADOPT method*), 112
`__setstate__` () (*dicee.models.adapt.ADOPT method*), 60
`__str__` () (*dicee.EnsembleKGE method*), 211
`__str__` () (*dicee.KGE method*), 214
`__str__` () (*dicee.knowledge_graph_embeddings.KGE method*), 53
`__str__` () (*dicee.models.ensemble.EnsembleKGE method*), 86
`__version__` (in module *dicee*), 232

A

AbstractCallback (class in *dicee.abstracts*), 16
 AbstractPPECallback (class in *dicee.abstracts*), 17
 AbstractTrainer (class in *dicee.abstracts*), 12
 AccumulateEpochLossCallback (class in *dicee.callbacks*), 21
 achieve_answer() (*dicee.query_generator.QueryGenerator* method), 162
 achieve_answer() (*dicee.QueryGenerator* method), 231
 AConEx (class in *dicee*), 196
 AConEx (class in *dicee.models*), 131
 AConEx (class in *dicee.models.complex*), 83
 AConvO (class in *dicee*), 196
 AConvO (class in *dicee.models*), 144
 AConvO (class in *dicee.models.octonion*), 94
 AConvQ (class in *dicee*), 197
 AConvQ (class in *dicee.models*), 138
 AConvQ (class in *dicee.models.quaternion*), 98
 adaptive_lr (*dicee.config.Namespace* attribute), 32
 adaptive_swa (*dicee.config.Namespace* attribute), 32
 add_new_entity_embeddings() (*dicee.abstracts.BaseInteractiveKGE* method), 15
 add_noise_rate (*dicee.config.Namespace* attribute), 30
 add_noise_rate (*dicee.knowledge_graph.KG* attribute), 52
 add_noisy_triples() (in module *dicee*), 212
 add_noisy_triples() (in module *dicee.static_funcs*), 174
 add_noisy_triples_into_training() (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk* method), 164
 add_noisy_triples_into_training() (*dicee.read_preprocess_save_load_kg.ReadFromDisk* method), 169
 add_reciprocal (*dicee.knowledge_graph.KG* attribute), 52
 ADOPT (class in *dicee.models*), 110
 ADOPT (class in *dicee.models.adopt*), 58
 adopt() (in module *dicee.models.adopt*), 62
 AllvsAll (class in *dicee*), 221
 AllvsAll (class in *dicee.dataset_classes*), 36
 alphas (*dicee.abstracts.AbstractPPECallback* attribute), 18
 alphas (*dicee.callbacks.ASWA* attribute), 24
 analyse() (in module *dicee.analyse_experiments*), 20
 answer_multi_hop_query() (*dicee.KGE* method), 216
 answer_multi_hop_query() (*dicee.knowledge_graph_embeddings.KGE* method), 55
 app (in module *dicee.scripts.index_serve*), 171
 apply_coefficients() (*dicee.DeCaL* method), 192
 apply_coefficients() (*dicee.Keci* method), 189
 apply_coefficients() (*dicee.models.clifford.DeCaL* method), 80
 apply_coefficients() (*dicee.models.clifford.Keci* method), 76
 apply_coefficients() (*dicee.models.DeCaL* method), 149
 apply_coefficients() (*dicee.models.Keci* method), 146
 apply_reciprocal_or_noise() (in module *dicee.read_preprocess_save_load_kg.util*), 167
 apply_semantic_constraint (*dicee.abstracts.BaseInteractiveKGE* attribute), 14
 apply_unit_norm (*dicee.BaseKGE* attribute), 209
 apply_unit_norm (*dicee.models.base_model.BaseKGE* attribute), 72
 apply_unit_norm (*dicee.models.BaseKGE* attribute), 120, 123, 129, 133, 139, 152, 155
 args (*dicee.BaseKGE* attribute), 209
 args (*dicee.DICE_Trainer* attribute), 213
 args (*dicee.EnsembleKGE* attribute), 211
 args (*dicee.evaluator.Evaluator* attribute), 48
 args (*dicee.executer.Execute* attribute), 50
 args (*dicee.models.base_model.BaseKGE* attribute), 72
 args (*dicee.models.base_model.IdentityClass* attribute), 74
 args (*dicee.models.BaseKGE* attribute), 119, 123, 128, 133, 139, 152, 155
 args (*dicee.models.ensemble.EnsembleKGE* attribute), 86
 args (*dicee.models.IdentityClass* attribute), 122, 135, 141
 args (*dicee.models.pykeen_models.PykeenKGE* attribute), 95
 args (*dicee.models.PykeenKGE* attribute), 154
 args (*dicee.PykeenKGE* attribute), 205
 args (*dicee.trainer.DICE_Trainer* attribute), 182
 args (*dicee.trainer.dice_trainer.DICE_Trainer* attribute), 177
 ASWA (class in *dicee.callbacks*), 24
 aswa (*dicee.analyse_experiments.Experiment* attribute), 19
 attn (*dicee.models.Block* attribute), 125
 attn (*dicee.models.transformers.Block* attribute), 107
 attn_dropout (*dicee.models.transformers.SelfAttention* attribute), 105

attributes (*dicee.abstracts.AbstractTrainer* attribute), 13
auto_batch_finding (*dicee.config.Namespace* attribute), 32

B

backend (*dicee.config.Namespace* attribute), 30
backend (*dicee.knowledge_graph.KG* attribute), 52
BaseInteractiveKGE (*class in dicee.abstracts*), 14
BaseInteractiveTrainKGE (*class in dicee.abstracts*), 18
BaseKGE (*class in dicee*), 208
BaseKGE (*class in dicee.models*), 119, 122, 128, 132, 138, 151, 155
BaseKGE (*class in dicee.models.base_model*), 71
BaseKGELightning (*class in dicee.models*), 113
BaseKGELightning (*class in dicee.models.base_model*), 66
batch_kronecker_product () (*dicee.callbacks.KronE* static method), 26
batch_size (*dicee.analyse_experiments.Experiment* attribute), 19
batch_size (*dicee.callbacks.PseudoLabellingCallback* attribute), 24
batch_size (*dicee.config.Namespace* attribute), 30
batch_size (*dicee.CVDataModule* attribute), 226
batch_size (*dicee.dataset_classes.CVDataModule* attribute), 41
batches_per_epoch (*dicee.callbacks.LRScheduler* attribute), 28
bias (*dicee.models.CoKEConfig* attribute), 127
bias (*dicee.models.real.CoKEConfig* attribute), 100
bias (*dicee.models.transformers.GPTConfig* attribute), 107
bias (*dicee.models.transformers.LayerNorm* attribute), 104
Block (*class in dicee.models*), 124
Block (*class in dicee.models.transformers*), 106
block_size (*dicee.BaseKGE* attribute), 210
block_size (*dicee.config.Namespace* attribute), 32
block_size (*dicee.dataset_classes.MultiClassClassificationDataset* attribute), 35
block_size (*dicee.models.base_model.BaseKGE* attribute), 73
block_size (*dicee.models.BaseKGE* attribute), 120, 124, 129, 134, 140, 153, 156
block_size (*dicee.models.CoKEConfig* attribute), 126
block_size (*dicee.models.real.CoKEConfig* attribute), 100
block_size (*dicee.models.transformers.GPTConfig* attribute), 107
block_size (*dicee.MultiClassClassificationDataset* attribute), 219
blocks (*dicee.CoKE* attribute), 205
blocks (*dicee.models.CoKE* attribute), 128
blocks (*dicee.models.real.CoKE* attribute), 101
bn_conv1 (*dicee.AConvQ* attribute), 197
bn_conv1 (*dicee.ConvQ* attribute), 198
bn_conv1 (*dicee.models.AConvQ* attribute), 138
bn_conv1 (*dicee.models.ConvQ* attribute), 137
bn_conv1 (*dicee.models.quaternion.AConvQ* attribute), 98
bn_conv1 (*dicee.models.quaternion.ConvQ* attribute), 98
bn_conv2 (*dicee.AConvQ* attribute), 197
bn_conv2 (*dicee.ConvQ* attribute), 198
bn_conv2 (*dicee.models.AConvQ* attribute), 138
bn_conv2 (*dicee.models.ConvQ* attribute), 137
bn_conv2 (*dicee.models.quaternion.AConvQ* attribute), 98
bn_conv2 (*dicee.models.quaternion.ConvQ* attribute), 98
bn_conv2d (*dicee.AConEx* attribute), 196
bn_conv2d (*dicee.AConvO* attribute), 197
bn_conv2d (*dicee.ConEx* attribute), 200
bn_conv2d (*dicee.ConvO* attribute), 199
bn_conv2d (*dicee.models.AConEx* attribute), 131
bn_conv2d (*dicee.models.AConvO* attribute), 144
bn_conv2d (*dicee.models.complex.AConEx* attribute), 83
bn_conv2d (*dicee.models.complex.ConEx* attribute), 82
bn_conv2d (*dicee.models.ConEx* attribute), 130
bn_conv2d (*dicee.models.ConvO* attribute), 143
bn_conv2d (*dicee.models.octonion.AConvO* attribute), 94
bn_conv2d (*dicee.models.octonion.ConvO* attribute), 93
BPE_NegativeSamplingDataset (*class in dicee*), 218
BPE_NegativeSamplingDataset (*class in dicee.dataset_classes*), 33
build_chain_funcs () (*dicee.models.FMult2* method), 158
build_chain_funcs () (*dicee.models.function_space.FMult2* method), 88
build_func () (*dicee.models.FMult2* method), 158

build_func() (*dicee.models.function_space.FMult2 method*), 88
 Byte (class in *dicee*), 206
 Byte (class in *dicee.models.transformers*), 102
 byte_pair_encoding (*dicee.analyse_experiments.Experiment attribute*), 19
 byte_pair_encoding (*dicee.BaseKGE attribute*), 209
 byte_pair_encoding (*dicee.config.Namespace attribute*), 31
 byte_pair_encoding (*dicee.knowledge_graph.KG attribute*), 52
 byte_pair_encoding (*dicee.models.base_model.BaseKGE attribute*), 73
 byte_pair_encoding (*dicee.models.BaseKGE attribute*), 120, 123, 129, 134, 140, 153, 156

C

c_attn (*dicee.models.transformers.SelfAttention attribute*), 105
 c_fc (*dicee.models.transformers.MLP attribute*), 106
 c_proj (*dicee.models.transformers.MLP attribute*), 106
 c_proj (*dicee.models.transformers.SelfAttention attribute*), 105
 callbacks (*dicee.abstracts.AbstractTrainer attribute*), 13
 callbacks (*dicee.analyse_experiments.Experiment attribute*), 19
 callbacks (*dicee.config.Namespace attribute*), 30
 callbacks (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 182
 causal (*dicee.models.CoKEConfig attribute*), 127
 causal (*dicee.models.real.CoKEConfig attribute*), 100
 causal (*dicee.models.transformers.GPTConfig attribute*), 107
 causal (*dicee.models.transformers.SelfAttention attribute*), 105
 chain_func() (*dicee.models.FMult method*), 157
 chain_func() (*dicee.models.function_space.FMult method*), 87
 chain_func() (*dicee.models.function_space.GFMult method*), 87
 chain_func() (*dicee.models.GFMult method*), 158
 CKeci (class in *dicee*), 187
 CKeci (class in *dicee.models*), 147
 CKeci (class in *dicee.models.clifford*), 78
 cl_pqr() (*dicee.DeCaL method*), 191
 cl_pqr() (*dicee.models.clifford.DeCaL method*), 79
 cl_pqr() (*dicee.models.DeCaL method*), 149
 cleanup() (*dicee.executer.Execute method*), 50
 clifford_multiplication() (*dicee.Keci method*), 189
 clifford_multiplication() (*dicee.models.clifford.Keci method*), 76
 clifford_multiplication() (*dicee.models.Keci method*), 146
 clip_lambda (*dicee.models.ADOPT attribute*), 112
 clip_lambda (*dicee.models.adopt.ADOPT attribute*), 60
 CoKE (class in *dicee*), 204
 CoKE (class in *dicee.models*), 127
 CoKE (class in *dicee.models.real*), 100
 coke_dropout (*dicee.CoKE attribute*), 205
 coke_dropout (*dicee.models.CoKE attribute*), 128
 coke_dropout (*dicee.models.real.CoKE attribute*), 101
 CoKEConfig (class in *dicee.models*), 126
 CoKEConfig (class in *dicee.models.real*), 100
 collate_fn (*dicee.AllvsAll attribute*), 221
 collate_fn (*dicee.dataset_classes.AllvsAll attribute*), 37
 collate_fn (*dicee.dataset_classes.KvsAll attribute*), 36
 collate_fn (*dicee.dataset_classes.KvsSampleDataset attribute*), 39
 collate_fn (*dicee.dataset_classes.MultiClassClassificationDataset attribute*), 35
 collate_fn (*dicee.dataset_classes.MultiLabelDataset attribute*), 34
 collate_fn (*dicee.dataset_classes.OnevsAllDataset attribute*), 35
 collate_fn (*dicee.dataset_classes.OnevsSample attribute*), 38
 collate_fn (*dicee.KvsAll attribute*), 221
 collate_fn (*dicee.KvsSampleDataset attribute*), 224
 collate_fn (*dicee.MultiClassClassificationDataset attribute*), 219
 collate_fn (*dicee.MultiLabelDataset attribute*), 219
 collate_fn (*dicee.OnevsAllDataset attribute*), 220
 collate_fn (*dicee.OnevsSample attribute*), 222, 223
 collate_fn() (*dicee.BPE_NegativeSamplingDataset method*), 218
 collate_fn() (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 34
 collate_fn() (*dicee.dataset_classes.TriplePredictionDataset method*), 41
 collate_fn() (*dicee.TriplePredictionDataset method*), 225
 collection_name (*dicee.scripts.index_serve.NeuralSearcher attribute*), 171
 comp_func() (*dicee.LFMult method*), 204

`comp_func()` (*dicee.models.function_space.LFMult method*), 89
`comp_func()` (*dicee.models.LFMult method*), 160
`Complex` (*class in dicee*), 195
`Complex` (*class in dicee.models*), 131
`Complex` (*class in dicee.models.complex*), 83
`compute_convergence()` (*in module dicee.callbacks*), 24
`compute_func()` (*dicee.models.FMult method*), 157
`compute_func()` (*dicee.models.FMult2 method*), 158
`compute_func()` (*dicee.models.function_space.FMult method*), 87
`compute_func()` (*dicee.models.function_space.FMult2 method*), 88
`compute_func()` (*dicee.models.function_space.GFMult method*), 87
`compute_func()` (*dicee.models.GFMult method*), 158
`compute_mrr()` (*dicee.callbacks.ASWA static method*), 25
`compute_sigma_pp()` (*dicee.DeCaL method*), 192
`compute_sigma_pp()` (*dicee.Keci method*), 188
`compute_sigma_pp()` (*dicee.models.clifford.DeCaL method*), 80
`compute_sigma_pp()` (*dicee.models.clifford.Keci method*), 76
`compute_sigma_pp()` (*dicee.models.DeCaL method*), 150
`compute_sigma_pp()` (*dicee.models.Keci method*), 145
`compute_sigma_pq()` (*dicee.DeCaL method*), 193
`compute_sigma_pq()` (*dicee.Keci method*), 188
`compute_sigma_pq()` (*dicee.models.clifford.DeCaL method*), 81
`compute_sigma_pq()` (*dicee.models.clifford.Keci method*), 76
`compute_sigma_pq()` (*dicee.models.DeCaL method*), 151
`compute_sigma_pq()` (*dicee.models.Keci method*), 146
`compute_sigma_pr()` (*dicee.DeCaL method*), 194
`compute_sigma_pr()` (*dicee.models.clifford.DeCaL method*), 81
`compute_sigma_pr()` (*dicee.models.DeCaL method*), 151
`compute_sigma_qq()` (*dicee.DeCaL method*), 193
`compute_sigma_qq()` (*dicee.Keci method*), 188
`compute_sigma_qq()` (*dicee.models.clifford.DeCaL method*), 81
`compute_sigma_qq()` (*dicee.models.clifford.Keci method*), 76
`compute_sigma_qq()` (*dicee.models.DeCaL method*), 150
`compute_sigma_qq()` (*dicee.models.Keci method*), 145
`compute_sigma_qr()` (*dicee.DeCaL method*), 194
`compute_sigma_qr()` (*dicee.models.clifford.DeCaL method*), 82
`compute_sigma_qr()` (*dicee.models.DeCaL method*), 151
`compute_sigma_rr()` (*dicee.DeCaL method*), 193
`compute_sigma_rr()` (*dicee.models.clifford.DeCaL method*), 81
`compute_sigma_rr()` (*dicee.models.DeCaL method*), 150
`compute_sigmas_multivect()` (*dicee.DeCaL method*), 192
`compute_sigmas_multivect()` (*dicee.models.clifford.DeCaL method*), 79
`compute_sigmas_multivect()` (*dicee.models.DeCaL method*), 149
`compute_sigmas_single()` (*dicee.DeCaL method*), 191
`compute_sigmas_single()` (*dicee.models.clifford.DeCaL method*), 79
`compute_sigmas_single()` (*dicee.models.DeCaL method*), 149
`ConEx` (*class in dicee*), 199
`ConEx` (*class in dicee.models*), 130
`ConEx` (*class in dicee.models.complex*), 82
`config` (*dicee.BytE attribute*), 207
`config` (*dicee.CoKE attribute*), 205
`config` (*dicee.models.CoKE attribute*), 127
`config` (*dicee.models.real.CoKE attribute*), 101
`config` (*dicee.models.transformers.BytE attribute*), 103
`config` (*dicee.models.transformers.GPT attribute*), 108
`configs` (*dicee.abstracts.BaseInteractiveKGE attribute*), 14
`configure_optimizers()` (*dicee.models.base_model.BaseKGELightning method*), 70
`configure_optimizers()` (*dicee.models.BaseKGELightning method*), 117
`configure_optimizers()` (*dicee.models.transformers.GPT method*), 108
`construct_batch_selected_cl_multivector()` (*dicee.Keci method*), 189
`construct_batch_selected_cl_multivector()` (*dicee.models.clifford.Keci method*), 77
`construct_batch_selected_cl_multivector()` (*dicee.models.Keci method*), 147
`construct_cl_multivector()` (*dicee.DeCaL method*), 192
`construct_cl_multivector()` (*dicee.Keci method*), 189
`construct_cl_multivector()` (*dicee.models.clifford.DeCaL method*), 80
`construct_cl_multivector()` (*dicee.models.clifford.Keci method*), 77
`construct_cl_multivector()` (*dicee.models.DeCaL method*), 149
`construct_cl_multivector()` (*dicee.models.Keci method*), 146

`construct_dataset()` (in module *dicee*), 218
`construct_dataset()` (in module *dicee.dataset_classes*), 33
`construct_ensemble` (*dicee.abstracts.BaseInteractiveKGE* attribute), 14
`construct_graph()` (*dicee.query_generator.QueryGenerator* method), 162
`construct_graph()` (*dicee.QueryGenerator* method), 231
`construct_input_and_output()` (*dicee.abstracts.BaseInteractiveKGE* method), 16
`construct_multi_coeff()` (*dicee.LFMulti* method), 203
`construct_multi_coeff()` (*dicee.models.function_space.LFMulti* method), 89
`construct_multi_coeff()` (*dicee.models.LFMulti* method), 159
`continual_learning` (*dicee.config.Namespace* attribute), 32
`continual_start()` (*dicee.DICE_Trainer* method), 213
`continual_start()` (*dicee.executer.ContinuousExecute* method), 51
`continual_start()` (*dicee.trainer.DICE_Trainer* method), 182
`continual_start()` (*dicee.trainer.dice_trainer.DICE_Trainer* method), 178
`continual_training_setup_executor()` (in module *dicee*), 212
`continual_training_setup_executor()` (in module *dicee.static_funcs*), 175
`ContinuousExecute` (class in *dicee.executer*), 51
`conv2d` (*dicee.AConEx* attribute), 196
`conv2d` (*dicee.AConvO* attribute), 197
`conv2d` (*dicee.AConvQ* attribute), 197
`conv2d` (*dicee.ConEx* attribute), 199
`conv2d` (*dicee.ConvO* attribute), 199
`conv2d` (*dicee.ConvQ* attribute), 198
`conv2d` (*dicee.models.AConEx* attribute), 131
`conv2d` (*dicee.models.AConvO* attribute), 144
`conv2d` (*dicee.models.AConvQ* attribute), 138
`conv2d` (*dicee.models.complex.AConEx* attribute), 83
`conv2d` (*dicee.models.complex.ConEx* attribute), 82
`conv2d` (*dicee.models.ConEx* attribute), 130
`conv2d` (*dicee.models.ConvO* attribute), 143
`conv2d` (*dicee.models.ConvQ* attribute), 137
`conv2d` (*dicee.models.octonion.AConvO* attribute), 94
`conv2d` (*dicee.models.octonion.ConvO* attribute), 93
`conv2d` (*dicee.models.quaternion.AConvQ* attribute), 98
`conv2d` (*dicee.models.quaternion.ConvQ* attribute), 98
`ConvO` (class in *dicee*), 198
`ConvO` (class in *dicee.models*), 143
`ConvO` (class in *dicee.models.octonion*), 93
`ConvQ` (class in *dicee*), 198
`ConvQ` (class in *dicee.models*), 137
`ConvQ` (class in *dicee.models.quaternion*), 97
`count_triples()` (in module *dicee.read_preprocess_save_load_kg.util*), 167
`create_and_store_kg()` (*dicee.executer.Execute* method), 50
`create_constraints()` (in module *dicee.read_preprocess_save_load_kg.util*), 167
`create_constraints()` (in module *dicee.static_preprocess_funcs*), 176
`create_experiment_folder()` (in module *dicee*), 212
`create_experiment_folder()` (in module *dicee.static_funcs*), 175
`create_random_data()` (*dicee.callbacks.PseudoLabellingCallback* method), 24
`create_recipriocal_triples()` (in module *dicee*), 211
`create_recipriocal_triples()` (in module *dicee.read_preprocess_save_load_kg.util*), 168
`create_recipriocal_triples()` (in module *dicee.static_funcs*), 173
`create_vector_database()` (*dicee.KGE* method), 214
`create_vector_database()` (*dicee.knowledge_graph_embeddings.KGE* method), 53
`crop_block_size()` (*dicee.models.transformers.GPT* method), 108
`ctx` (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 182
`current_epoch` (*dicee.callbacks.SWA* attribute), 29
`CVDDataModule` (class in *dicee*), 225
`CVDDataModule` (class in *dicee.dataset_classes*), 41
`cycle_length` (*dicee.callbacks.LRScheduler* attribute), 28

D

`data_module` (*dicee.callbacks.PseudoLabellingCallback* attribute), 24
`data_property_embeddings` (*dicee.models.literal.LiteralEmbeddings* attribute), 91
`data_property_to_idx` (*dicee.dataset_classes.LiteralDataset* attribute), 45
`data_property_to_idx` (*dicee.LiteralDataset* attribute), 230
`dataset_dir` (*dicee.config.Namespace* attribute), 29
`dataset_dir` (*dicee.knowledge_graph.KG* attribute), 52

- `dataset_sanity_checking()` (in module *dicee.read_preprocess_save_load_kg.util*), 168
- `DeCaL` (class in *dicee*), 190
- `DeCaL` (class in *dicee.models*), 147
- `DeCaL` (class in *dicee.models.clifford*), 78
- `decide()` (*dicee.callbacks.ASWA* method), 25
- `default_eval_model` (*dicee.callbacks.PeriodicEvalCallback* attribute), 28
- `degree` (*dicee.LFMult* attribute), 203
- `degree` (*dicee.models.function_space.LFMult* attribute), 89
- `degree` (*dicee.models.LFMult* attribute), 159
- `denormalize()` (*dicee.dataset_classes.LiteralDataset* static method), 46
- `denormalize()` (*dicee.LiteralDataset* static method), 230
- `deploy()` (*dicee.KGE* method), 217
- `deploy()` (*dicee.knowledge_graph_embeddings.KGE* method), 56
- `deploy_head_entity_prediction()` (in module *dicee*), 212
- `deploy_head_entity_prediction()` (in module *dicee.static_funcs*), 174
- `deploy_relation_prediction()` (in module *dicee*), 212
- `deploy_relation_prediction()` (in module *dicee.static_funcs*), 175
- `deploy_tail_entity_prediction()` (in module *dicee*), 212
- `deploy_tail_entity_prediction()` (in module *dicee.static_funcs*), 174
- `deploy_triple_prediction()` (in module *dicee*), 212
- `deploy_triple_prediction()` (in module *dicee.static_funcs*), 174
- `describe()` (*dicee.knowledge_graph.KG* method), 53
- `description_of_input` (*dicee.knowledge_graph.KG* attribute), 53
- `device` (*dicee.models.literal.LiteralEmbeddings* property), 91
- `DICE_Trainer` (class in *dicee*), 213
- `DICE_Trainer` (class in *dicee.trainer*), 182
- `DICE_Trainer` (class in *dicee.trainer.dice_trainer*), 177
- dicee*
 - module, 12
- dicee.__main__*
 - module, 12
- dicee.abstracts*
 - module, 12
- dicee.analyse_experiments*
 - module, 19
- dicee.callbacks*
 - module, 20
- dicee.config*
 - module, 29
- dicee.dataset_classes*
 - module, 32
- dicee.eval_static_funcs*
 - module, 46
- dicee.evaluator*
 - module, 48
- dicee.executer*
 - module, 49
- dicee.knowledge_graph*
 - module, 51
- dicee.knowledge_graph_embeddings*
 - module, 53
- dicee.models*
 - module, 57
- dicee.models.adopt*
 - module, 57
- dicee.models.base_model*
 - module, 66
- dicee.models.clifford*
 - module, 75
- dicee.models.complex*
 - module, 82
- dicee.models.dualE*
 - module, 84
- dicee.models.ensemble*
 - module, 85
- dicee.models.function_space*
 - module, 86
- dicee.models.literal*

- module, 90
- `dicee.models.octonion`
 - module, 91
- `dicee.models.pykeen_models`
 - module, 94
- `dicee.models.quaternion`
 - module, 96
- `dicee.models.real`
 - module, 99
- `dicee.models.static_funcs`
 - module, 102
- `dicee.models.transformers`
 - module, 102
- `dicee.query_generator`
 - module, 161
- `dicee.read_preprocess_save_load_kg`
 - module, 163
- `dicee.read_preprocess_save_load_kg.preprocess`
 - module, 163
- `dicee.read_preprocess_save_load_kg.read_from_disk`
 - module, 163
- `dicee.read_preprocess_save_load_kg.save_load_disk`
 - module, 164
- `dicee.read_preprocess_save_load_kg.util`
 - module, 165
- `dicee.sanity_checkers`
 - module, 169
- `dicee.scripts`
 - module, 170
- `dicee.scripts.index_serve`
 - module, 170
- `dicee.scripts.run`
 - module, 172
- `dicee.static_funcs`
 - module, 172
- `dicee.static_funcs_training`
 - module, 175
- `dicee.static_preprocess_funcs`
 - module, 176
- `dicee.trainer`
 - module, 177
- `dicee.trainer.dice_trainer`
 - module, 177
- `dicee.trainer.model_parallelism`
 - module, 179
- `dicee.trainer.torch_trainer`
 - module, 179
- `dicee.trainer.torch_trainer_ddp`
 - module, 181
- `discrete_points` (*dicee.models.FMult2 attribute*), 158
- `discrete_points` (*dicee.models.function_space.FMult2 attribute*), 88
- `dist_func` (*dicee.models.Pyke attribute*), 126
- `dist_func` (*dicee.models.real.Pyke attribute*), 100
- `dist_func` (*dicee.Pyke attribute*), 186
- `DistMult` (*class in dicee*), 186
- `DistMult` (*class in dicee.models*), 125
- `DistMult` (*class in dicee.models.real*), 99
- `distributed` (*dicee.executer.Execute attribute*), 50
- `download_file()` (*in module dicee*), 212
- `download_file()` (*in module dicee.static_funcs*), 175
- `download_files_from_url()` (*in module dicee*), 212
- `download_files_from_url()` (*in module dicee.static_funcs*), 175
- `download_pretrained_model()` (*in module dicee*), 213
- `download_pretrained_model()` (*in module dicee.static_funcs*), 175
- `dropout` (*dicee.models.CoKEConfig attribute*), 127
- `dropout` (*dicee.models.literal.LiteralEmbeddings attribute*), 90, 91
- `dropout` (*dicee.models.real.CoKEConfig attribute*), 100
- `dropout` (*dicee.models.transformers.GPTConfig attribute*), 107

dropout (*dicee.models.transformers.MLP attribute*), 106
 dropout (*dicee.models.transformers.SelfAttention attribute*), 105
 DualE (*class in dicee*), 194
 DualE (*class in dicee.models*), 160
 DualE (*class in dicee.models.dualE*), 84
 dummy_eval() (*dicee.evaluator.Evaluator method*), 49
 dummy_id (*dicee.knowledge_graph.KG attribute*), 52
 during_training (*dicee.evaluator.Evaluator attribute*), 48

E

ee_vocab (*dicee.evaluator.Evaluator attribute*), 48
 efficient_zero_grad() (*in module dicee.static_funcs_training*), 176
 embedding_dim (*dicee.analyse_experiments.Experiment attribute*), 19
 embedding_dim (*dicee.BaseKGE attribute*), 209
 embedding_dim (*dicee.config.Namespace attribute*), 30
 embedding_dim (*dicee.models.base_model.BaseKGE attribute*), 72
 embedding_dim (*dicee.models.BaseKGE attribute*), 119, 123, 128, 133, 139, 152, 155
 embedding_dim (*dicee.models.literal.LiteralEmbeddings attribute*), 91
 embedding_dims (*dicee.models.literal.LiteralEmbeddings attribute*), 90
 enable_log (*in module dicee.static_preprocess_funcs*), 176
 enc (*dicee.knowledge_graph.KG attribute*), 52
 end() (*dicee.executor.Execute method*), 50
 EnsembleKGE (*class in dicee*), 210
 EnsembleKGE (*class in dicee.models.ensemble*), 86
 ent2id (*dicee.query_generator.QueryGenerator attribute*), 162
 ent2id (*dicee.QueryGenerator attribute*), 231
 ent_in (*dicee.query_generator.QueryGenerator attribute*), 162
 ent_in (*dicee.QueryGenerator attribute*), 231
 ent_out (*dicee.query_generator.QueryGenerator attribute*), 162
 ent_out (*dicee.QueryGenerator attribute*), 231
 entities_str (*dicee.knowledge_graph.KG property*), 53
 entity_embeddings (*dicee.AConvQ attribute*), 197
 entity_embeddings (*dicee.ConvQ attribute*), 198
 entity_embeddings (*dicee.DeCaL attribute*), 191
 entity_embeddings (*dicee.DualE attribute*), 194
 entity_embeddings (*dicee.LFMult attribute*), 203
 entity_embeddings (*dicee.models.AConvQ attribute*), 138
 entity_embeddings (*dicee.models.clifford.DeCaL attribute*), 79
 entity_embeddings (*dicee.models.ConvQ attribute*), 137
 entity_embeddings (*dicee.models.DeCaL attribute*), 148
 entity_embeddings (*dicee.models.DualE attribute*), 160
 entity_embeddings (*dicee.models.dualE.DualE attribute*), 84
 entity_embeddings (*dicee.models.FMult attribute*), 157
 entity_embeddings (*dicee.models.FMult2 attribute*), 158
 entity_embeddings (*dicee.models.function_space.FMult attribute*), 87
 entity_embeddings (*dicee.models.function_space.FMult2 attribute*), 88
 entity_embeddings (*dicee.models.function_space.GFMult attribute*), 87
 entity_embeddings (*dicee.models.function_space.LFMult attribute*), 89
 entity_embeddings (*dicee.models.function_space.LFMult1 attribute*), 88
 entity_embeddings (*dicee.models.GFMult attribute*), 158
 entity_embeddings (*dicee.models.LFMult attribute*), 159
 entity_embeddings (*dicee.models.LFMult1 attribute*), 159
 entity_embeddings (*dicee.models.literal.LiteralEmbeddings attribute*), 90, 91
 entity_embeddings (*dicee.models.pykeen_models.PykeenKGE attribute*), 95
 entity_embeddings (*dicee.models.PykeenKGE attribute*), 154
 entity_embeddings (*dicee.models.quaternion.AConvQ attribute*), 98
 entity_embeddings (*dicee.models.quaternion.ConvQ attribute*), 98
 entity_embeddings (*dicee.PykeenKGE attribute*), 205
 entity_to_idx (*dicee.dataset_classes.LiteralDataset attribute*), 45, 46
 entity_to_idx (*dicee.knowledge_graph.KG attribute*), 52
 entity_to_idx (*dicee.LiteralDataset attribute*), 229, 230
 entity_to_idx (*dicee.scripts.index_serve.NeuralSearcher attribute*), 171
 epoch_count (*dicee.abstracts.AbstractPPECallback attribute*), 17
 epoch_count (*dicee.callbacks.ASWA attribute*), 24
 epoch_counter (*dicee.callbacks.Eval attribute*), 25
 epoch_counter (*dicee.callbacks.KGESaveCallback attribute*), 23
 epoch_counter (*dicee.callbacks.PeriodicEvalCallback attribute*), 27

epoch_ratio (*dicee.callbacks.Eval* attribute), 25
 er_vocab (*dicee.evaluator.Evaluator* attribute), 48
 estimate_mfu () (*dicee.models.transformers.GPT* method), 108
 estimate_q () (*in module dicee.callbacks*), 24
 Eval (*class in dicee.callbacks*), 25
 eval () (*dicee.EnsembleKGE* method), 211
 eval () (*dicee.evaluator.Evaluator* method), 49
 eval () (*dicee.models.ensemble.EnsembleKGE* method), 86
 eval_at_epochs (*dicee.config.Namespace* attribute), 32
 eval_epochs (*dicee.callbacks.PeriodicEvalCallback* attribute), 28
 eval_every_n_epochs (*dicee.config.Namespace* attribute), 32
 eval_lp_performance () (*dicee.KGE* method), 214
 eval_lp_performance () (*dicee.knowledge_graph_embeddings.KGE* method), 53
 eval_model (*dicee.config.Namespace* attribute), 31
 eval_model (*dicee.knowledge_graph.KG* attribute), 52
 eval_rank_of_head_and_tail_byte_pair_encoded_entity () (*dicee.evaluator.Evaluator* method), 49
 eval_rank_of_head_and_tail_entity () (*dicee.evaluator.Evaluator* method), 49
 eval_with_bpe_vs_all () (*dicee.evaluator.Evaluator* method), 49
 eval_with_byte () (*dicee.evaluator.Evaluator* method), 49
 eval_with_data () (*dicee.evaluator.Evaluator* method), 49
 eval_with_vs_all () (*dicee.evaluator.Evaluator* method), 49
 evaluate () (*in module dicee*), 212
 evaluate () (*in module dicee.static_funcs*), 175
 evaluate_bpe_lp () (*in module dicee.static_funcs_training*), 176
 evaluate_ensemble_link_prediction_performance () (*in module dicee.eval_static_funcs*), 48
 evaluate_link_prediction_performance () (*in module dicee.eval_static_funcs*), 47
 evaluate_link_prediction_performance_with_bpe () (*in module dicee.eval_static_funcs*), 47
 evaluate_link_prediction_performance_with_bpe_reciprocals () (*in module dicee.eval_static_funcs*), 47
 evaluate_link_prediction_performance_with_reciprocals () (*in module dicee.eval_static_funcs*), 47
 evaluate_literal_prediction () (*in module dicee.eval_static_funcs*), 47
 evaluate_lp () (*dicee.evaluator.Evaluator* method), 49
 evaluate_lp () (*in module dicee.static_funcs_training*), 175
 evaluate_lp_bpe_k_vs_all () (*dicee.evaluator.Evaluator* method), 49
 evaluate_lp_bpe_k_vs_all () (*in module dicee.eval_static_funcs*), 47
 evaluate_lp_k_vs_all () (*dicee.evaluator.Evaluator* method), 49
 evaluate_lp_with_byte () (*dicee.evaluator.Evaluator* method), 49
 Evaluator (*class in dicee.evaluator*), 48
 evaluator (*dicee.DICE_Trainer* attribute), 213
 evaluator (*dicee.executer.Execute* attribute), 50
 evaluator (*dicee.trainer.DICE_Trainer* attribute), 182
 evaluator (*dicee.trainer.dice_trainer.DICE_Trainer* attribute), 178
 every_x_epoch (*dicee.callbacks.KGESaveCallback* attribute), 23
 example_input_array (*dicee.EnsembleKGE* property), 211
 example_input_array (*dicee.models.ensemble.EnsembleKGE* property), 86
 Execute (*class in dicee.executer*), 50
 exists () (*dicee.knowledge_graph.KG* method), 53
 Experiment (*class in dicee.analyse_experiments*), 19
 experiment_dir (*dicee.callbacks.LRScheduler* attribute), 28
 experiment_dir (*dicee.callbacks.PeriodicEvalCallback* attribute), 27
 explicit (*dicee.models.QMult* attribute), 136
 explicit (*dicee.models.quaternion.QMult* attribute), 97
 explicit (*dicee.QMult* attribute), 201
 exponential_function () (*in module dicee*), 212
 exponential_function () (*in module dicee.static_funcs*), 175
 extract_input_outputs () (*dicee.trainer.torch_trainer_ddp.NodeTrainer* method), 182
 extract_input_outputs () (*in module dicee.trainer.model_parallelism*), 179
 extract_input_outputs_set_device () (*dicee.trainer.torch_trainer.TorchTrainer* method), 180

F

f (*dicee.callbacks.KronE* attribute), 26
 fc (*dicee.models.literal.LiteralEmbeddings* attribute), 91
 fc1 (*dicee.AConEx* attribute), 196
 fc1 (*dicee.AConvO* attribute), 197
 fc1 (*dicee.AConvQ* attribute), 197
 fc1 (*dicee.ConEx* attribute), 199
 fc1 (*dicee.ConvO* attribute), 199
 fc1 (*dicee.ConvQ* attribute), 198

`fc1` (*dicee.models.AConEx* attribute), 131
`fc1` (*dicee.models.AConvO* attribute), 144
`fc1` (*dicee.models.AConvQ* attribute), 138
`fc1` (*dicee.models.complex.AConEx* attribute), 83
`fc1` (*dicee.models.complex.ConEx* attribute), 82
`fc1` (*dicee.models.ConEx* attribute), 130
`fc1` (*dicee.models.ConvO* attribute), 143
`fc1` (*dicee.models.ConvQ* attribute), 137
`fc1` (*dicee.models.octonion.AConvO* attribute), 94
`fc1` (*dicee.models.octonion.ConvO* attribute), 93
`fc1` (*dicee.models.quaternion.AConvQ* attribute), 98
`fc1` (*dicee.models.quaternion.ConvQ* attribute), 98
`fc_num_input` (*dicee.AConEx* attribute), 196
`fc_num_input` (*dicee.AConvO* attribute), 197
`fc_num_input` (*dicee.AConvQ* attribute), 197
`fc_num_input` (*dicee.ConEx* attribute), 199
`fc_num_input` (*dicee.ConvO* attribute), 199
`fc_num_input` (*dicee.ConvQ* attribute), 198
`fc_num_input` (*dicee.models.AConEx* attribute), 131
`fc_num_input` (*dicee.models.AConvO* attribute), 144
`fc_num_input` (*dicee.models.AConvQ* attribute), 138
`fc_num_input` (*dicee.models.complex.AConEx* attribute), 83
`fc_num_input` (*dicee.models.complex.ConEx* attribute), 82
`fc_num_input` (*dicee.models.ConEx* attribute), 130
`fc_num_input` (*dicee.models.ConvO* attribute), 143
`fc_num_input` (*dicee.models.ConvQ* attribute), 137
`fc_num_input` (*dicee.models.octonion.AConvO* attribute), 94
`fc_num_input` (*dicee.models.octonion.ConvO* attribute), 93
`fc_num_input` (*dicee.models.quaternion.AConvQ* attribute), 98
`fc_num_input` (*dicee.models.quaternion.ConvQ* attribute), 98
`fc_out` (*dicee.models.literal.LiteralEmbeddings* attribute), 91
`feature_map_dropout` (*dicee.AConEx* attribute), 196
`feature_map_dropout` (*dicee.AConvO* attribute), 197
`feature_map_dropout` (*dicee.AConvQ* attribute), 197
`feature_map_dropout` (*dicee.ConEx* attribute), 200
`feature_map_dropout` (*dicee.ConvO* attribute), 199
`feature_map_dropout` (*dicee.ConvQ* attribute), 198
`feature_map_dropout` (*dicee.models.AConEx* attribute), 131
`feature_map_dropout` (*dicee.models.AConvO* attribute), 144
`feature_map_dropout` (*dicee.models.AConvQ* attribute), 138
`feature_map_dropout` (*dicee.models.complex.AConEx* attribute), 83
`feature_map_dropout` (*dicee.models.complex.ConEx* attribute), 82
`feature_map_dropout` (*dicee.models.ConEx* attribute), 130
`feature_map_dropout` (*dicee.models.ConvO* attribute), 143
`feature_map_dropout` (*dicee.models.ConvQ* attribute), 137
`feature_map_dropout` (*dicee.models.octonion.AConvO* attribute), 94
`feature_map_dropout` (*dicee.models.octonion.ConvO* attribute), 93
`feature_map_dropout` (*dicee.models.quaternion.AConvQ* attribute), 98
`feature_map_dropout` (*dicee.models.quaternion.ConvQ* attribute), 98
`feature_map_dropout_rate` (*dicee.BaseKGE* attribute), 209
`feature_map_dropout_rate` (*dicee.config.Namespace* attribute), 31
`feature_map_dropout_rate` (*dicee.models.base_model.BaseKGE* attribute), 72
`feature_map_dropout_rate` (*dicee.models.BaseKGE* attribute), 120, 123, 129, 133, 139, 152, 155
`fetch_worker()` (in module *dicee.read_preprocess_save_load_kg.util*), 167
`fill_query()` (*dicee.query_generator.QueryGenerator* method), 162
`fill_query()` (*dicee.QueryGenerator* method), 231
`find_good_batch_size()` (in module *dicee.trainer.model_parallelism*), 179
`find_missing_triples()` (*dicee.KGE* method), 217
`find_missing_triples()` (*dicee.knowledge_graph_embeddings.KGE* method), 56
`fit()` (*dicee.trainer.model_parallelism.TensorParallel* method), 179
`fit()` (*dicee.trainer.torch_trainer_ddp.TorchDDPTrainer* method), 181
`fit()` (*dicee.trainer.torch_trainer.TorchTrainer* method), 180
`flash` (*dicee.models.transformers.SelfAttention* attribute), 105
`FMult` (class in *dicee.models*), 157
`FMult` (class in *dicee.models.function_space*), 87
`FMult2` (class in *dicee.models*), 158
`FMult2` (class in *dicee.models.function_space*), 87
`form_of_labelling` (*dicee.DICE_Trainer* attribute), 213

form_of_labelling (*dicdee.trainer.DICE_Trainer* attribute), 182
 form_of_labelling (*dicdee.trainer.dice_trainer.DICE_Trainer* attribute), 178
 forward() (*dicdee.BaseKGE* method), 210
 forward() (*dicdee.BytE* method), 207
 forward() (*dicdee.models.base_model.BaseKGE* method), 73
 forward() (*dicdee.models.base_model.IdentityClass* static method), 75
 forward() (*dicdee.models.BaseKGE* method), 121, 124, 130, 134, 140, 153, 156
 forward() (*dicdee.models.Block* method), 125
 forward() (*dicdee.models.IdentityClass* static method), 122, 136, 142
 forward() (*dicdee.models.literal.LiteralEmbeddings* method), 91
 forward() (*dicdee.models.transformers.Block* method), 107
 forward() (*dicdee.models.transformers.BytE* method), 103
 forward() (*dicdee.models.transformers.GPT* method), 108
 forward() (*dicdee.models.transformers.LayerNorm* method), 104
 forward() (*dicdee.models.transformers.MLP* method), 106
 forward() (*dicdee.models.transformers.SelfAttention* method), 105
 forward_backward_update() (*dicdee.trainer.torch_trainer.TorchTrainer* method), 180
 forward_backward_update_loss() (in module *dicdee.trainer.model_parallelism*), 179
 forward_byte_pair_encoded_k_vs_all() (*dicdee.BaseKGE* method), 210
 forward_byte_pair_encoded_k_vs_all() (*dicdee.models.base_model.BaseKGE* method), 73
 forward_byte_pair_encoded_k_vs_all() (*dicdee.models.BaseKGE* method), 120, 124, 129, 134, 140, 153, 156
 forward_byte_pair_encoded_triple() (*dicdee.BaseKGE* method), 210
 forward_byte_pair_encoded_triple() (*dicdee.models.base_model.BaseKGE* method), 73
 forward_byte_pair_encoded_triple() (*dicdee.models.BaseKGE* method), 120, 124, 129, 134, 140, 153, 156
 forward_k_vs_all() (*dicdee.AConEx* method), 196
 forward_k_vs_all() (*dicdee.AConvO* method), 197
 forward_k_vs_all() (*dicdee.AConvQ* method), 197
 forward_k_vs_all() (*dicdee.BaseKGE* method), 210
 forward_k_vs_all() (*dicdee.CoKE* method), 205
 forward_k_vs_all() (*dicdee.ComplEx* method), 196
 forward_k_vs_all() (*dicdee.ConEx* method), 200
 forward_k_vs_all() (*dicdee.ConvO* method), 199
 forward_k_vs_all() (*dicdee.ConvQ* method), 198
 forward_k_vs_all() (*dicdee.DeCaL* method), 192
 forward_k_vs_all() (*dicdee.DistMult* method), 187
 forward_k_vs_all() (*dicdee.DualE* method), 195
 forward_k_vs_all() (*dicdee.Keci* method), 189
 forward_k_vs_all() (*dicdee.models.AConEx* method), 131
 forward_k_vs_all() (*dicdee.models.AConvO* method), 144
 forward_k_vs_all() (*dicdee.models.AConvQ* method), 138
 forward_k_vs_all() (*dicdee.models.base_model.BaseKGE* method), 73
 forward_k_vs_all() (*dicdee.models.BaseKGE* method), 121, 124, 130, 134, 140, 153, 157
 forward_k_vs_all() (*dicdee.models.clifford.DeCaL* method), 80
 forward_k_vs_all() (*dicdee.models.clifford.Keci* method), 77
 forward_k_vs_all() (*dicdee.models.CoKE* method), 128
 forward_k_vs_all() (*dicdee.models.ComplEx* method), 132
 forward_k_vs_all() (*dicdee.models.complex.AConEx* method), 83
 forward_k_vs_all() (*dicdee.models.complex.ComplEx* method), 84
 forward_k_vs_all() (*dicdee.models.complex.ConEx* method), 82
 forward_k_vs_all() (*dicdee.models.ConEx* method), 131
 forward_k_vs_all() (*dicdee.models.ConvO* method), 144
 forward_k_vs_all() (*dicdee.models.ConvQ* method), 138
 forward_k_vs_all() (*dicdee.models.DeCaL* method), 149
 forward_k_vs_all() (*dicdee.models.DistMult* method), 126
 forward_k_vs_all() (*dicdee.models.DualE* method), 161
 forward_k_vs_all() (*dicdee.models.dualE.DualE* method), 85
 forward_k_vs_all() (*dicdee.models.Keci* method), 146
 forward_k_vs_all() (*dicdee.models.octonion.AConvO* method), 94
 forward_k_vs_all() (*dicdee.models.octonion.ConvO* method), 94
 forward_k_vs_all() (*dicdee.models.octonion.OMult* method), 93
 forward_k_vs_all() (*dicdee.models.OMult* method), 142
 forward_k_vs_all() (*dicdee.models.pykeen_models.PykeenKGE* method), 95
 forward_k_vs_all() (*dicdee.models.PykeenKGE* method), 154
 forward_k_vs_all() (*dicdee.models.QMult* method), 137
 forward_k_vs_all() (*dicdee.models.quaternion.AConvQ* method), 99
 forward_k_vs_all() (*dicdee.models.quaternion.ConvQ* method), 98
 forward_k_vs_all() (*dicdee.models.quaternion.QMult* method), 97
 forward_k_vs_all() (*dicdee.models.real.CoKE* method), 101

`forward_k_vs_all()` (*dicee.models.real.DistMult method*), 99
`forward_k_vs_all()` (*dicee.models.real.Shallom method*), 100
`forward_k_vs_all()` (*dicee.models.real.TransE method*), 100
`forward_k_vs_all()` (*dicee.models.Shallom method*), 126
`forward_k_vs_all()` (*dicee.models.TransE method*), 126
`forward_k_vs_all()` (*dicee.OMult method*), 202
`forward_k_vs_all()` (*dicee.PykeenKGE method*), 205
`forward_k_vs_all()` (*dicee.QMult method*), 201
`forward_k_vs_all()` (*dicee.Shallom method*), 203
`forward_k_vs_all()` (*dicee.TransE method*), 190
`forward_k_vs_sample()` (*dicee.AConEx method*), 196
`forward_k_vs_sample()` (*dicee.BaseKGE method*), 210
`forward_k_vs_sample()` (*dicee.CoKE method*), 205
`forward_k_vs_sample()` (*dicee.ComplEx method*), 196
`forward_k_vs_sample()` (*dicee.ConEx method*), 200
`forward_k_vs_sample()` (*dicee.DistMult method*), 187
`forward_k_vs_sample()` (*dicee.Keci method*), 190
`forward_k_vs_sample()` (*dicee.models.AConEx method*), 131
`forward_k_vs_sample()` (*dicee.models.base_model.BaseKGE method*), 73
`forward_k_vs_sample()` (*dicee.models.BaseKGE method*), 121, 124, 130, 134, 140, 153, 157
`forward_k_vs_sample()` (*dicee.models.clifford.Keci method*), 78
`forward_k_vs_sample()` (*dicee.models.CoKE method*), 128
`forward_k_vs_sample()` (*dicee.models.ComplEx method*), 132
`forward_k_vs_sample()` (*dicee.models.complex.AConEx method*), 83
`forward_k_vs_sample()` (*dicee.models.complex.ComplEx method*), 84
`forward_k_vs_sample()` (*dicee.models.complex.ConEx method*), 83
`forward_k_vs_sample()` (*dicee.models.ConEx method*), 131
`forward_k_vs_sample()` (*dicee.models.DistMult method*), 126
`forward_k_vs_sample()` (*dicee.models.Keci method*), 147
`forward_k_vs_sample()` (*dicee.models.pykeen_models.PykeenKGE method*), 95
`forward_k_vs_sample()` (*dicee.models.PykeenKGE method*), 154
`forward_k_vs_sample()` (*dicee.models.QMult method*), 137
`forward_k_vs_sample()` (*dicee.models.quaternion.QMult method*), 97
`forward_k_vs_sample()` (*dicee.models.real.CoKE method*), 101
`forward_k_vs_sample()` (*dicee.models.real.DistMult method*), 99
`forward_k_vs_sample()` (*dicee.PykeenKGE method*), 206
`forward_k_vs_sample()` (*dicee.QMult method*), 201
`forward_k_vs_with_explicit()` (*dicee.Keci method*), 189
`forward_k_vs_with_explicit()` (*dicee.models.clifford.Keci method*), 77
`forward_k_vs_with_explicit()` (*dicee.models.Keci method*), 146
`forward_triples()` (*dicee.AConEx method*), 196
`forward_triples()` (*dicee.AConvO method*), 197
`forward_triples()` (*dicee.AConvQ method*), 197
`forward_triples()` (*dicee.BaseKGE method*), 210
`forward_triples()` (*dicee.ConEx method*), 200
`forward_triples()` (*dicee.ConvO method*), 199
`forward_triples()` (*dicee.ConvQ method*), 198
`forward_triples()` (*dicee.DeCaL method*), 191
`forward_triples()` (*dicee.DualE method*), 194
`forward_triples()` (*dicee.Keci method*), 190
`forward_triples()` (*dicee.LFMult method*), 203
`forward_triples()` (*dicee.models.AConEx method*), 131
`forward_triples()` (*dicee.models.AConvO method*), 144
`forward_triples()` (*dicee.models.AConvQ method*), 138
`forward_triples()` (*dicee.models.base_model.BaseKGE method*), 73
`forward_triples()` (*dicee.models.BaseKGE method*), 121, 124, 130, 134, 140, 153, 156
`forward_triples()` (*dicee.models.clifford.DeCaL method*), 79
`forward_triples()` (*dicee.models.clifford.Keci method*), 78
`forward_triples()` (*dicee.models.complex.AConEx method*), 83
`forward_triples()` (*dicee.models.complex.ConEx method*), 82
`forward_triples()` (*dicee.models.ConEx method*), 131
`forward_triples()` (*dicee.models.ConvO method*), 143
`forward_triples()` (*dicee.models.ConvQ method*), 138
`forward_triples()` (*dicee.models.DeCaL method*), 148
`forward_triples()` (*dicee.models.DualE method*), 161
`forward_triples()` (*dicee.models.dualE.DualE method*), 85
`forward_triples()` (*dicee.models.FMult method*), 157
`forward_triples()` (*dicee.models.FMult2 method*), 158

`forward_triples()` (*dicee.models.function_space.FMult method*), 87
`forward_triples()` (*dicee.models.function_space.FMult2 method*), 88
`forward_triples()` (*dicee.models.function_space.GFMult method*), 87
`forward_triples()` (*dicee.models.function_space.LFMult method*), 89
`forward_triples()` (*dicee.models.function_space.LFMult1 method*), 88
`forward_triples()` (*dicee.models.GFMult method*), 158
`forward_triples()` (*dicee.models.Keci method*), 147
`forward_triples()` (*dicee.models.LFMult method*), 159
`forward_triples()` (*dicee.models.LFMult1 method*), 159
`forward_triples()` (*dicee.models.octonion.AConvO method*), 94
`forward_triples()` (*dicee.models.octonion.ConvO method*), 94
`forward_triples()` (*dicee.models.Pyke method*), 126
`forward_triples()` (*dicee.models.pykeen_models.PykeenKGE method*), 95
`forward_triples()` (*dicee.models.PykeenKGE method*), 154
`forward_triples()` (*dicee.models.quaternion.AConvQ method*), 98
`forward_triples()` (*dicee.models.quaternion.ConvQ method*), 98
`forward_triples()` (*dicee.models.real.Pyke method*), 100
`forward_triples()` (*dicee.models.real.Shallom method*), 100
`forward_triples()` (*dicee.models.Shallom method*), 126
`forward_triples()` (*dicee.Pyke method*), 186
`forward_triples()` (*dicee.PykeenKGE method*), 206
`forward_triples()` (*dicee.Shallom method*), 203
`freeze_entity_embeddings` (*dicee.models.literal.LiteralEmbeddings attribute*), 91
`frequency` (*dicee.callbacks.Perturb attribute*), 27
`from_pretrained()` (*dicee.models.transformers.GPT class method*), 108
`from_pretrained_model_write_embeddings_into_csv()` (*in module dicee*), 213
`from_pretrained_model_write_embeddings_into_csv()` (*in module dicee.static_funcs*), 175
`full_storage_path` (*dicee.analyse_experiments.Experiment attribute*), 19
`func_triple_to_bpe_representation` (*dicee.evaluator.Evaluator attribute*), 48
`func_triple_to_bpe_representation()` (*dicee.knowledge_graph.KG method*), 53
`function()` (*dicee.models.FMult2 method*), 158
`function()` (*dicee.models.function_space.FMult2 method*), 88

G

`gamma` (*dicee.models.FMult attribute*), 157
`gamma` (*dicee.models.function_space.FMult attribute*), 87
`gate_residual` (*dicee.models.literal.LiteralEmbeddings attribute*), 90, 91
`gated_residual_proj` (*dicee.models.literal.LiteralEmbeddings attribute*), 91
`gelu` (*dicee.models.transformers.MLP attribute*), 106
`gen_test` (*dicee.query_generator.QueryGenerator attribute*), 162
`gen_test` (*dicee.QueryGenerator attribute*), 231
`gen_valid` (*dicee.query_generator.QueryGenerator attribute*), 161
`gen_valid` (*dicee.QueryGenerator attribute*), 231
`generate()` (*dicee.BytE method*), 207
`generate()` (*dicee.KGE method*), 214
`generate()` (*dicee.knowledge_graph_embeddings.KGE method*), 53
`generate()` (*dicee.models.transformers.BytE method*), 103
`generate_queries()` (*dicee.query_generator.QueryGenerator method*), 162
`generate_queries()` (*dicee.QueryGenerator method*), 232
`get_aswa_state_dict()` (*dicee.callbacks.ASWA method*), 25
`get_bpe_head_and_relation_representation()` (*dicee.BaseKGE method*), 210
`get_bpe_head_and_relation_representation()` (*dicee.models.base_model.BaseKGE method*), 74
`get_bpe_head_and_relation_representation()` (*dicee.models.BaseKGE method*), 121, 124, 130, 135, 141, 153, 157
`get_bpe_token_representation()` (*dicee.abstracts.BaseInteractiveKGE method*), 14
`get_callbacks()` (*in module dicee.trainer.dice_trainer*), 177
`get_default_arguments()` (*in module dicee.analyse_experiments*), 19
`get_default_arguments()` (*in module dicee.scripts.index_serve*), 171
`get_default_arguments()` (*in module dicee.scripts.run*), 172
`get_ee_vocab()` (*in module dicee*), 211
`get_ee_vocab()` (*in module dicee.read_preprocess_save_load_kg.util*), 167
`get_ee_vocab()` (*in module dicee.static_funcs*), 173
`get_ee_vocab()` (*in module dicee.static_preprocess_funcs*), 176
`get_embeddings()` (*dicee.BaseKGE method*), 210
`get_embeddings()` (*dicee.EnsembleKGE method*), 211
`get_embeddings()` (*dicee.models.base_model.BaseKGE method*), 74
`get_embeddings()` (*dicee.models.BaseKGE method*), 121, 124, 130, 135, 141, 154, 157
`get_embeddings()` (*dicee.models.ensemble.EnsembleKGE method*), 86

`get_embeddings()` (*dicee.models.real.Shallom method*), 100
`get_embeddings()` (*dicee.models.Shallom method*), 126
`get_embeddings()` (*dicee.Shallom method*), 203
`get_entity_embeddings()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`get_entity_index()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`get_er_vocab()` (*in module dicee*), 211
`get_er_vocab()` (*in module dicee.read_preprocess_save_load_kg.util*), 167
`get_er_vocab()` (*in module dicee.static_funcs*), 173
`get_er_vocab()` (*in module dicee.static_preprocess_funcs*), 176
`get_eval_report()` (*dicee.abstracts.BaseInteractiveKGE method*), 14
`get_head_relation_representation()` (*dicee.BaseKGE method*), 210
`get_head_relation_representation()` (*dicee.models.base_model.BaseKGE method*), 74
`get_head_relation_representation()` (*dicee.models.BaseKGE method*), 121, 124, 130, 134, 140, 153, 157
`get_kronecker_triple_representation()` (*dicee.callbacks.KronE method*), 26
`get_num_params()` (*dicee.models.transformers.GPT method*), 108
`get_padded_bpe_triple_representation()` (*dicee.abstracts.BaseInteractiveKGE method*), 14
`get_queries()` (*dicee.query_generator.QueryGenerator method*), 162
`get_queries()` (*dicee.QueryGenerator method*), 232
`get_re_vocab()` (*in module dicee*), 211
`get_re_vocab()` (*in module dicee.read_preprocess_save_load_kg.util*), 167
`get_re_vocab()` (*in module dicee.static_funcs*), 173
`get_re_vocab()` (*in module dicee.static_preprocess_funcs*), 176
`get_relation_embeddings()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`get_relation_index()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`get_sentence_representation()` (*dicee.BaseKGE method*), 210
`get_sentence_representation()` (*dicee.models.base_model.BaseKGE method*), 74
`get_sentence_representation()` (*dicee.models.BaseKGE method*), 121, 124, 130, 135, 141, 153, 157
`get_transductive_entity_embeddings()` (*dicee.KGE method*), 214
`get_transductive_entity_embeddings()` (*dicee.knowledge_graph_embeddings.KGE method*), 53
`get_triple_representation()` (*dicee.BaseKGE method*), 210
`get_triple_representation()` (*dicee.models.base_model.BaseKGE method*), 73
`get_triple_representation()` (*dicee.models.BaseKGE method*), 121, 124, 130, 134, 140, 153, 157
`GFMult` (*class in dicee.models*), 157
`GFMult` (*class in dicee.models.function_space*), 87
`global_rank` (*dicee.abstracts.AbstractTrainer attribute*), 13
`global_rank` (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 181
`GPT` (*class in dicee.models.transformers*), 107
`GPTConfig` (*class in dicee.models.transformers*), 107
`gpus` (*dicee.config.Namespace attribute*), 30
`gradient_accumulation_steps` (*dicee.config.Namespace attribute*), 31
`ground_queries()` (*dicee.query_generator.QueryGenerator method*), 162
`ground_queries()` (*dicee.QueryGenerator method*), 231

H

`hidden_dim` (*dicee.models.literal.LiteralEmbeddings attribute*), 91
`hidden_dropout` (*dicee.BaseKGE attribute*), 209
`hidden_dropout` (*dicee.models.base_model.BaseKGE attribute*), 73
`hidden_dropout` (*dicee.models.BaseKGE attribute*), 120, 123, 129, 134, 140, 153, 156
`hidden_dropout_rate` (*dicee.BaseKGE attribute*), 209
`hidden_dropout_rate` (*dicee.config.Namespace attribute*), 31
`hidden_dropout_rate` (*dicee.models.base_model.BaseKGE attribute*), 72
`hidden_dropout_rate` (*dicee.models.BaseKGE attribute*), 120, 123, 129, 133, 139, 152, 155
`hidden_normalizer` (*dicee.BaseKGE attribute*), 209
`hidden_normalizer` (*dicee.models.base_model.BaseKGE attribute*), 73
`hidden_normalizer` (*dicee.models.BaseKGE attribute*), 120, 123, 129, 134, 140, 152, 156

I

`IdentityClass` (*class in dicee.models*), 121, 135, 141
`IdentityClass` (*class in dicee.models.base_model*), 74
`idx_entity_to_bpe_shaped` (*dicee.knowledge_graph.KG attribute*), 52
`index()` (*in module dicee.scripts.index_serve*), 171
`index_triple()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`init_dataloader()` (*dicee.DICE_Trainer method*), 214
`init_dataloader()` (*dicee.trainer.DICE_Trainer method*), 183
`init_dataloader()` (*dicee.trainer.dice_trainer.DICE_Trainer method*), 178
`init_dataset()` (*dicee.DICE_Trainer method*), 214
`init_dataset()` (*dicee.trainer.DICE_Trainer method*), 183

`init_dataset()` (*dicee.trainer.dice_trainer.DICE_Trainer* method), 178
`init_param` (*dicee.config.Namespace* attribute), 31
`init_params_with_sanity_checking()` (*dicee.BaseKGE* method), 210
`init_params_with_sanity_checking()` (*dicee.models.base_model.BaseKGE* method), 73
`init_params_with_sanity_checking()` (*dicee.models.BaseKGE* method), 120, 124, 129, 134, 140, 153, 156
`initial_eval_setting` (*dicee.callbacks.ASWA* attribute), 24
`initialize_or_load_model()` (*dicee.DICE_Trainer* method), 214
`initialize_or_load_model()` (*dicee.trainer.DICE_Trainer* method), 183
`initialize_or_load_model()` (*dicee.trainer.dice_trainer.DICE_Trainer* method), 178
`initialize_trainer()` (*dicee.DICE_Trainer* method), 213
`initialize_trainer()` (*dicee.trainer.DICE_Trainer* method), 183
`initialize_trainer()` (*dicee.trainer.dice_trainer.DICE_Trainer* method), 178
`initialize_trainer()` (in module *dicee.trainer.dice_trainer*), 177
`input_dp_ent_real` (*dicee.BaseKGE* attribute), 209
`input_dp_ent_real` (*dicee.models.base_model.BaseKGE* attribute), 73
`input_dp_ent_real` (*dicee.models.BaseKGE* attribute), 120, 123, 129, 134, 140, 153, 156
`input_dp_rel_real` (*dicee.BaseKGE* attribute), 209
`input_dp_rel_real` (*dicee.models.base_model.BaseKGE* attribute), 73
`input_dp_rel_real` (*dicee.models.BaseKGE* attribute), 120, 123, 129, 134, 140, 153, 156
`input_dropout_rate` (*dicee.BaseKGE* attribute), 209
`input_dropout_rate` (*dicee.config.Namespace* attribute), 31
`input_dropout_rate` (*dicee.models.base_model.BaseKGE* attribute), 72
`input_dropout_rate` (*dicee.models.BaseKGE* attribute), 120, 123, 129, 133, 139, 152, 155
`InteractiveQueryDecomposition` (class in *dicee.abstracts*), 16
`intialize_model()` (in module *dicee*), 212
`intialize_model()` (in module *dicee.static_funcs*), 174
`is_continual_training` (*dicee.DICE_Trainer* attribute), 213
`is_continual_training` (*dicee.evaluator.Evaluator* attribute), 48
`is_continual_training` (*dicee.executer.Execute* attribute), 50
`is_continual_training` (*dicee.trainer.DICE_Trainer* attribute), 182
`is_continual_training` (*dicee.trainer.dice_trainer.DICE_Trainer* attribute), 177
`is_global_zero` (*dicee.abstracts.AbstractTrainer* attribute), 13
`is_rank_zero()` (*dicee.executer.Execute* method), 50
`is_seen()` (*dicee.abstracts.BaseInteractiveKGE* method), 15
`is_sparql_endpoint_alive()` (in module *dicee.sanity_checkers*), 170

K

`k` (*dicee.models.FMult* attribute), 157
`k` (*dicee.models.FMult2* attribute), 158
`k` (*dicee.models.function_space.FMult* attribute), 87
`k` (*dicee.models.function_space.FMult2* attribute), 88
`k` (*dicee.models.function_space.GFMult* attribute), 87
`k` (*dicee.models.GFMult* attribute), 158
`k_fold_cross_validation()` (*dicee.DICE_Trainer* method), 214
`k_fold_cross_validation()` (*dicee.trainer.DICE_Trainer* method), 183
`k_fold_cross_validation()` (*dicee.trainer.dice_trainer.DICE_Trainer* method), 178
`k_vs_all_score()` (*dicee.ComplEx* static method), 196
`k_vs_all_score()` (*dicee.DistMult* method), 187
`k_vs_all_score()` (*dicee.Keci* method), 189
`k_vs_all_score()` (*dicee.models.clifford.Keci* method), 77
`k_vs_all_score()` (*dicee.models.ComplEx* static method), 132
`k_vs_all_score()` (*dicee.models.complex.ComplEx* static method), 84
`k_vs_all_score()` (*dicee.models.DistMult* method), 125
`k_vs_all_score()` (*dicee.models.Keci* method), 146
`k_vs_all_score()` (*dicee.models.octonion.OMult* method), 93
`k_vs_all_score()` (*dicee.models.OMult* method), 142
`k_vs_all_score()` (*dicee.models.QMult* method), 137
`k_vs_all_score()` (*dicee.models.quaternion.QMult* method), 97
`k_vs_all_score()` (*dicee.models.real.DistMult* method), 99
`k_vs_all_score()` (*dicee.OMult* method), 202
`k_vs_all_score()` (*dicee.QMult* method), 201
`Keci` (class in *dicee*), 187
`Keci` (class in *dicee.models*), 144
`Keci` (class in *dicee.models.clifford*), 75
`kernel_size` (*dicee.BaseKGE* attribute), 209
`kernel_size` (*dicee.config.Namespace* attribute), 31
`kernel_size` (*dicee.models.base_model.BaseKGE* attribute), 72

kernel_size (*dicee.models.BaseKGE attribute*), 120, 123, 129, 133, 139, 152, 156
 KG (*class in dicee.knowledge_graph*), 51
 kg (*dicee.callbacks.PseudoLabellingCallback attribute*), 24
 kg (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk attribute*), 169
 kg (*dicee.read_preprocess_save_load_kg.PreprocessKG attribute*), 168
 kg (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG attribute*), 163
 kg (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk attribute*), 164
 kg (*dicee.read_preprocess_save_load_kg.ReadFromDisk attribute*), 169
 kg (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk attribute*), 164
 KGE (*class in dicee*), 214
 KGE (*class in dicee.knowledge_graph_embeddings*), 53
 KGESaveCallback (*class in dicee.callbacks*), 22
 knowledge_graph (*dicee.executer.Execute attribute*), 50
 KronE (*class in dicee.callbacks*), 26
 KvsAll (*class in dicee*), 220
 KvsAll (*class in dicee.dataset_classes*), 35
 kvsall_score () (*dicee.DualE method*), 194
 kvsall_score () (*dicee.models.DualE method*), 160
 kvsall_score () (*dicee.models.dualE.DualE method*), 85
 KvsSampleDataset (*class in dicee*), 223
 KvsSampleDataset (*class in dicee.dataset_classes*), 39

L

label_smoothing_rate (*dicee.AllvsAll attribute*), 221
 label_smoothing_rate (*dicee.config.Namespace attribute*), 31
 label_smoothing_rate (*dicee.dataset_classes.AllvsAll attribute*), 37
 label_smoothing_rate (*dicee.dataset_classes.KvsAll attribute*), 36
 label_smoothing_rate (*dicee.dataset_classes.KvsSampleDataset attribute*), 39
 label_smoothing_rate (*dicee.dataset_classes.OnevsSample attribute*), 38
 label_smoothing_rate (*dicee.dataset_classes.TriplePredictionDataset attribute*), 41
 label_smoothing_rate (*dicee.KvsAll attribute*), 221
 label_smoothing_rate (*dicee.KvsSampleDataset attribute*), 224
 label_smoothing_rate (*dicee.OnevsSample attribute*), 222, 223
 label_smoothing_rate (*dicee.TriplePredictionDataset attribute*), 225
 labels (*dicee.dataset_classes.NegSampleDataset attribute*), 40
 labels (*dicee.NegSampleDataset attribute*), 224
 layer_norm (*dicee.models.literal.LiteralEmbeddings attribute*), 91
 LayerNorm (*class in dicee.models.transformers*), 104
 learning_rate (*dicee.BaseKGE attribute*), 209
 learning_rate (*dicee.models.base_model.BaseKGE attribute*), 72
 learning_rate (*dicee.models.BaseKGE attribute*), 120, 123, 129, 133, 139, 152, 155
 length (*dicee.dataset_classes.NegSampleDataset attribute*), 40
 length (*dicee.dataset_classes.TriplePredictionDataset attribute*), 41
 length (*dicee.NegSampleDataset attribute*), 224
 length (*dicee.TriplePredictionDataset attribute*), 225
 level (*dicee.callbacks.Perturb attribute*), 27
 LFMult (*class in dicee*), 203
 LFMult (*class in dicee.models*), 159
 LFMult (*class in dicee.models.function_space*), 88
 LFMult1 (*class in dicee.models*), 159
 LFMult1 (*class in dicee.models.function_space*), 88
 linear () (*dicee.LFMult method*), 203
 linear () (*dicee.models.function_space.LFMult method*), 89
 linear () (*dicee.models.LFMult method*), 159
 list2tuple () (*dicee.query_generator.QueryGenerator method*), 162
 list2tuple () (*dicee.QueryGenerator method*), 231
 LiteralDataset (*class in dicee*), 229
 LiteralDataset (*class in dicee.dataset_classes*), 44
 LiteralEmbeddings (*class in dicee.models.literal*), 90
 lm_head (*dicee.BytE attribute*), 207
 lm_head (*dicee.models.transformers.BytE attribute*), 103
 lm_head (*dicee.models.transformers.GPT attribute*), 108
 ln_1 (*dicee.models.Block attribute*), 125
 ln_1 (*dicee.models.transformers.Block attribute*), 107
 ln_2 (*dicee.models.Block attribute*), 125
 ln_2 (*dicee.models.transformers.Block attribute*), 107
 ln_f (*dicee.CoKE attribute*), 205

`ln_f` (*dicee.models.CoKE* attribute), 128
`ln_f` (*dicee.models.real.CoKE* attribute), 101
`load` () (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk* method), 169
`load` () (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk* method), 164
`load_and_validate_literal_data` () (*dicee.dataset_classes.LiteralDataset* static method), 46
`load_and_validate_literal_data` () (*dicee.LiteralDataset* static method), 230
`load_from_memmap` () (*dicee.executer.Execute* method), 50
`load_json` () (in module *dicee*), 212
`load_json` () (in module *dicee.static_funcs*), 174
`load_model` () (in module *dicee*), 211
`load_model` () (in module *dicee.static_funcs*), 174
`load_model_ensemble` () (in module *dicee*), 211
`load_model_ensemble` () (in module *dicee.static_funcs*), 174
`load_numpy` () (in module *dicee*), 212
`load_numpy` () (in module *dicee.static_funcs*), 175
`load_numpy_ndarray` () (in module *dicee.read_preprocess_save_load_kg.util*), 168
`load_pickle` () (in module *dicee*), 211
`load_pickle` () (in module *dicee.read_preprocess_save_load_kg.util*), 168
`load_pickle` () (in module *dicee.static_funcs*), 174
`load_queries` () (*dicee.query_generator.QueryGenerator* method), 162
`load_queries` () (*dicee.QueryGenerator* method), 232
`load_queries_and_answers` () (*dicee.query_generator.QueryGenerator* static method), 162
`load_queries_and_answers` () (*dicee.QueryGenerator* static method), 232
`load_state_dict` () (*dicee.EnsembleKGE* method), 211
`load_state_dict` () (*dicee.models.ensemble.EnsembleKGE* method), 86
`load_term_mapping` () (in module *dicee*), 211, 218
`load_term_mapping` () (in module *dicee.static_funcs*), 174
`load_term_mapping` () (in module *dicee.trainer.dice_trainer*), 177
`load_with_pandas` () (in module *dicee.read_preprocess_save_load_kg.util*), 168
`loader_backend` (*dicee.dataset_classes.LiteralDataset* attribute), 45, 46
`loader_backend` (*dicee.LiteralDataset* attribute), 230
`LoadSaveToDisk` (class in *dicee.read_preprocess_save_load_kg*), 169
`LoadSaveToDisk` (class in *dicee.read_preprocess_save_load_kg.save_load_disk*), 164
`local_rank` (*dicee.abstracts.AbstractTrainer* attribute), 13
`local_rank` (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 181
`loss` (*dicee.BaseKGE* attribute), 209
`loss` (*dicee.models.base_model.BaseKGE* attribute), 72
`loss` (*dicee.models.BaseKGE* attribute), 120, 123, 129, 133, 139, 152, 156
`loss_func` (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 182
`loss_function` (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 180
`loss_function` () (*dicee.BytE* method), 207
`loss_function` () (*dicee.models.base_model.BaseKGELightning* method), 67
`loss_function` () (*dicee.models.BaseKGELightning* method), 115
`loss_function` () (*dicee.models.transformers.BytE* method), 103
`loss_history` (*dicee.BaseKGE* attribute), 209
`loss_history` (*dicee.models.base_model.BaseKGE* attribute), 73
`loss_history` (*dicee.models.BaseKGE* attribute), 120, 123, 129, 134, 140, 153, 156
`loss_history` (*dicee.models.pykeen_models.PykeenKGE* attribute), 95
`loss_history` (*dicee.models.PykeenKGE* attribute), 154
`loss_history` (*dicee.PykeenKGE* attribute), 205
`loss_history` (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 182
`lr` (*dicee.analyse_experiments.Experiment* attribute), 19
`lr` (*dicee.config.Namespace* attribute), 30
`lr_init` (*dicee.callbacks.SWA* attribute), 29
`lr_lambda` (*dicee.callbacks.LRScheduler* attribute), 28
`LRScheduler` (class in *dicee.callbacks*), 28

M

`m` (*dicee.LFMult* attribute), 203
`m` (*dicee.models.function_space.LFMult* attribute), 89
`m` (*dicee.models.LFMult* attribute), 159
`main` () (in module *dicee.scripts.index_serve*), 172
`main` () (in module *dicee.scripts.run*), 172
`make_iterable_verbose` () (in module *dicee.static_funcs_training*), 175
`make_iterable_verbose` () (in module *dicee.trainer.torch_trainer_ddp*), 181
`mapping_from_first_two_cols_to_third` () (in module *dicee*), 218
`mapping_from_first_two_cols_to_third` () (in module *dicee.static_preprocess_funcs*), 177

- margin (*dicee.models.Pyke* attribute), 126
- margin (*dicee.models.real.Pyke* attribute), 100
- margin (*dicee.models.real.TransE* attribute), 99
- margin (*dicee.models.TransE* attribute), 126
- margin (*dicee.Pyke* attribute), 186
- margin (*dicee.TransE* attribute), 190
- mask_emb (*dicee.CoKE* attribute), 205
- mask_emb (*dicee.models.CoKE* attribute), 127
- mask_emb (*dicee.models.real.CoKE* attribute), 101
- max_ans_num (*dicee.query_generator.QueryGenerator* attribute), 162
- max_ans_num (*dicee.QueryGenerator* attribute), 231
- max_epochs (*dicee.callbacks.KGESaveCallback* attribute), 23
- max_epochs (*dicee.callbacks.PeriodicEvalCallback* attribute), 27
- max_epochs (*dicee.callbacks.SWA* attribute), 29
- max_length_subword_tokens (*dicee.BaseKGE* attribute), 209
- max_length_subword_tokens (*dicee.knowledge_graph.KG* attribute), 52
- max_length_subword_tokens (*dicee.models.base_model.BaseKGE* attribute), 73
- max_length_subword_tokens (*dicee.models.BaseKGE* attribute), 120, 123, 129, 134, 140, 153, 156
- max_num_of_classes (*dicee.dataset_classes.KvsSampleDataset* attribute), 39
- max_num_of_classes (*dicee.KvsSampleDataset* attribute), 224
- mem_of_model () (*dicee.EnsembleKGE* method), 211
- mem_of_model () (*dicee.models.base_model.BaseKGE Lightning* method), 66
- mem_of_model () (*dicee.models.BaseKGE Lightning* method), 114
- mem_of_model () (*dicee.models.ensemble.EnsembleKGE* method), 86
- method (*dicee.callbacks.Perturb* attribute), 27
- MLP (*class in dicee.models.transformers*), 105
- mlp (*dicee.models.Block* attribute), 125
- mlp (*dicee.models.transformers.Block* attribute), 107
- mode (*dicee.query_generator.QueryGenerator* attribute), 162
- mode (*dicee.QueryGenerator* attribute), 231
- model (*dicee.config.Namespace* attribute), 30
- model (*dicee.models.pykeen_models.PykeenKGE* attribute), 95
- model (*dicee.models.PykeenKGE* attribute), 154
- model (*dicee.PykeenKGE* attribute), 205
- model (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 182
- model (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 180
- model_kwargs (*dicee.models.pykeen_models.PykeenKGE* attribute), 95
- model_kwargs (*dicee.models.PykeenKGE* attribute), 154
- model_kwargs (*dicee.PykeenKGE* attribute), 205
- model_name (*dicee.analyse_experiments.Experiment* attribute), 19
- module
 - dicee, 12
 - dicee.__main__, 12
 - dicee.abstracts, 12
 - dicee.analyse_experiments, 19
 - dicee.callbacks, 20
 - dicee.config, 29
 - dicee.dataset_classes, 32
 - dicee.eval_static_funcs, 46
 - dicee.evaluator, 48
 - dicee.executer, 49
 - dicee.knowledge_graph, 51
 - dicee.knowledge_graph_embeddings, 53
 - dicee.models, 57
 - dicee.models.adopt, 57
 - dicee.models.base_model, 66
 - dicee.models.clifford, 75
 - dicee.models.complex, 82
 - dicee.models.dualE, 84
 - dicee.models.ensemble, 85
 - dicee.models.function_space, 86
 - dicee.models.literal, 90
 - dicee.models.octonion, 91
 - dicee.models.pykeen_models, 94
 - dicee.models.quaternion, 96
 - dicee.models.real, 99
 - dicee.models.static_funcs, 102
 - dicee.models.transformers, 102

- dicee.query_generator, 161
- dicee.read_preprocess_save_load_kg, 163
- dicee.read_preprocess_save_load_kg.preprocess, 163
- dicee.read_preprocess_save_load_kg.read_from_disk, 163
- dicee.read_preprocess_save_load_kg.save_load_disk, 164
- dicee.read_preprocess_save_load_kg.util, 165
- dicee.sanity_checkers, 169
- dicee.scripts, 170
- dicee.scripts.index_serve, 170
- dicee.scripts.run, 172
- dicee.static_funcs, 172
- dicee.static_funcs_training, 175
- dicee.static_preprocess_funcs, 176
- dicee.trainer, 177
- dicee.trainer.dice_trainer, 177
- dicee.trainer.model_parallelism, 179
- dicee.trainer.torch_trainer, 179
- dicee.trainer.torch_trainer_ddp, 181
- modules () (*dicee.EnsembleKGE method*), 211
- modules () (*dicee.models.ensemble.EnsembleKGE method*), 86
- moving_average () (*dicee.callbacks.SWA static method*), 29
- MultiClassClassificationDataset (*class in dicee*), 219
- MultiClassClassificationDataset (*class in dicee.dataset_classes*), 34
- MultiLabelDataset (*class in dicee*), 218
- MultiLabelDataset (*class in dicee.dataset_classes*), 34

N

- n (*dicee.models.FMult2 attribute*), 158
- n (*dicee.models.function_space.FMult2 attribute*), 88
- n_embd (*dicee.models.CoKEConfig attribute*), 127
- n_embd (*dicee.models.real.CoKEConfig attribute*), 100
- n_embd (*dicee.models.transformers.GPTConfig attribute*), 107
- n_embd (*dicee.models.transformers.SelfAttention attribute*), 105
- n_epochs_eval_model (*dicee.callbacks.PeriodicEvalCallback attribute*), 27
- n_epochs_eval_model (*dicee.config.Namespace attribute*), 32
- n_head (*dicee.models.CoKEConfig attribute*), 127
- n_head (*dicee.models.real.CoKEConfig attribute*), 100
- n_head (*dicee.models.transformers.GPTConfig attribute*), 107
- n_head (*dicee.models.transformers.SelfAttention attribute*), 105
- n_layer (*dicee.models.CoKEConfig attribute*), 127
- n_layer (*dicee.models.real.CoKEConfig attribute*), 100
- n_layer (*dicee.models.transformers.GPTConfig attribute*), 107
- n_layers (*dicee.models.FMult2 attribute*), 158
- n_layers (*dicee.models.function_space.FMult2 attribute*), 88
- name (*dicee.abstracts.BaseInteractiveKGE property*), 15
- name (*dicee.AConEx attribute*), 196
- name (*dicee.AConvO attribute*), 196
- name (*dicee.AConvQ attribute*), 197
- name (*dicee.BytE attribute*), 206
- name (*dicee.CKeci attribute*), 187
- name (*dicee.CoKE attribute*), 205
- name (*dicee.ComplEx attribute*), 196
- name (*dicee.ConEx attribute*), 199
- name (*dicee.ConvO attribute*), 199
- name (*dicee.ConvQ attribute*), 198
- name (*dicee.DeCaL attribute*), 191
- name (*dicee.DistMult attribute*), 187
- name (*dicee.DualE attribute*), 194
- name (*dicee.EnsembleKGE attribute*), 210
- name (*dicee.Keci attribute*), 188
- name (*dicee.LFMult attribute*), 203
- name (*dicee.models.AConEx attribute*), 131
- name (*dicee.models.AConvO attribute*), 144
- name (*dicee.models.AConvQ attribute*), 138
- name (*dicee.models.CKeci attribute*), 147
- name (*dicee.models.clifford.CKeci attribute*), 78
- name (*dicee.models.clifford.DeCaL attribute*), 79

name (*dicee.models.clifford.Keci* attribute), 75
 name (*dicee.models.CoKE* attribute), 127
 name (*dicee.models.ComplEx* attribute), 132
 name (*dicee.models.complex.AConEx* attribute), 83
 name (*dicee.models.complex.ComplEx* attribute), 84
 name (*dicee.models.complex.ConEx* attribute), 82
 name (*dicee.models.ConEx* attribute), 130
 name (*dicee.models.ConvO* attribute), 143
 name (*dicee.models.ConvQ* attribute), 137
 name (*dicee.models.DeCaL* attribute), 148
 name (*dicee.models.DistMult* attribute), 125
 name (*dicee.models.DualE* attribute), 160
 name (*dicee.models.dualE.DualE* attribute), 84
 name (*dicee.models.ensemble.EnsembleKGE* attribute), 86
 name (*dicee.models.FMult* attribute), 157
 name (*dicee.models.FMult2* attribute), 158
 name (*dicee.models.function_space.FMult* attribute), 87
 name (*dicee.models.function_space.FMult2* attribute), 88
 name (*dicee.models.function_space.GFMult* attribute), 87
 name (*dicee.models.function_space.LFMult* attribute), 89
 name (*dicee.models.function_space.LFMult1* attribute), 88
 name (*dicee.models.GFMult* attribute), 158
 name (*dicee.models.Keci* attribute), 145
 name (*dicee.models.LFMult* attribute), 159
 name (*dicee.models.LFMult1* attribute), 159
 name (*dicee.models.octonion.AConvO* attribute), 94
 name (*dicee.models.octonion.ConvO* attribute), 93
 name (*dicee.models.octonion.OMult* attribute), 92
 name (*dicee.models.OMult* attribute), 142
 name (*dicee.models.Pyke* attribute), 126
 name (*dicee.models.pykeen_models.PykeenKGE* attribute), 95
 name (*dicee.models.PykeenKGE* attribute), 154
 name (*dicee.models.QMult* attribute), 136
 name (*dicee.models.quaternion.AConvQ* attribute), 98
 name (*dicee.models.quaternion.ConvQ* attribute), 98
 name (*dicee.models.quaternion.QMult* attribute), 97
 name (*dicee.models.real.CoKE* attribute), 101
 name (*dicee.models.real.DistMult* attribute), 99
 name (*dicee.models.real.Pyke* attribute), 100
 name (*dicee.models.real.Shallom* attribute), 100
 name (*dicee.models.real.TransE* attribute), 99
 name (*dicee.models.Shallom* attribute), 126
 name (*dicee.models.TransE* attribute), 126
 name (*dicee.models.transformers.BytE* attribute), 103
 name (*dicee.OMult* attribute), 202
 name (*dicee.Pyke* attribute), 186
 name (*dicee.PykeenKGE* attribute), 205
 name (*dicee.QMult* attribute), 201
 name (*dicee.Shallom* attribute), 202
 name (*dicee.TransE* attribute), 190
 named_children() (*dicee.EnsembleKGE* method), 211
 named_children() (*dicee.models.ensemble.EnsembleKGE* method), 86
 Namespace (class in *dicee.config*), 29
 neg_ratio (*dicee.BPE_NegativeSamplingDataset* attribute), 218
 neg_ratio (*dicee.config.Namespace* attribute), 30
 neg_ratio (*dicee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 34
 neg_ratio (*dicee.dataset_classes.KvsSampleDataset* attribute), 39
 neg_ratio (*dicee.KvsSampleDataset* attribute), 224
 neg_sample_ratio (*dicee.CVDDataModule* attribute), 226
 neg_sample_ratio (*dicee.dataset_classes.CVDDataModule* attribute), 41
 neg_sample_ratio (*dicee.dataset_classes.NegSampleDataset* attribute), 40
 neg_sample_ratio (*dicee.dataset_classes.OnevsSample* attribute), 38
 neg_sample_ratio (*dicee.dataset_classes.TriplePredictionDataset* attribute), 41
 neg_sample_ratio (*dicee.NegSampleDataset* attribute), 224
 neg_sample_ratio (*dicee.OnevsSample* attribute), 222, 223
 neg_sample_ratio (*dicee.TriplePredictionDataset* attribute), 225
 negnorm() (*dicee.abstracts.InteractiveQueryDecomposition* method), 16
 NegSampleDataset (class in *dicee*), 224

NegSampleDataset (class in *dicее.dataset_classes*), 39
 neural_searcher (in module *dicее.scripts.index_serve*), 171
 NeuralSearcher (class in *dicее.scripts.index_serve*), 171
 NodeTrainer (class in *dicее.trainer.torch_trainer_ddp*), 181
 norm_fc1 (*dicее.AConEx* attribute), 196
 norm_fc1 (*dicее.AConvO* attribute), 197
 norm_fc1 (*dicее.ConEx* attribute), 200
 norm_fc1 (*dicее.ConvO* attribute), 199
 norm_fc1 (*dicее.models.AConEx* attribute), 131
 norm_fc1 (*dicее.models.AConvO* attribute), 144
 norm_fc1 (*dicее.models.complex.AConEx* attribute), 83
 norm_fc1 (*dicее.models.complex.ConEx* attribute), 82
 norm_fc1 (*dicее.models.ConEx* attribute), 130
 norm_fc1 (*dicее.models.ConvO* attribute), 143
 norm_fc1 (*dicее.models.octonion.AConvO* attribute), 94
 norm_fc1 (*dicее.models.octonion.ConvO* attribute), 93
 normalization (*dicее.analyse_experiments.Experiment* attribute), 20
 normalization (*dicее.config.Namespace* attribute), 30
 normalization (*dicее.dataset_classes.LiteralDataset* attribute), 45
 normalization (*dicее.LiteralDataset* attribute), 229
 normalization_params (*dicее.dataset_classes.LiteralDataset* attribute), 45, 46
 normalization_params (*dicее.LiteralDataset* attribute), 229, 230
 normalization_type (*dicее.dataset_classes.LiteralDataset* attribute), 46
 normalization_type (*dicее.LiteralDataset* attribute), 230
 normalize_head_entity_embeddings (*dicее.BaseKGE* attribute), 209
 normalize_head_entity_embeddings (*dicее.models.base_model.BaseKGE* attribute), 73
 normalize_head_entity_embeddings (*dicее.models.BaseKGE* attribute), 120, 123, 129, 134, 140, 152, 156
 normalize_relation_embeddings (*dicее.BaseKGE* attribute), 209
 normalize_relation_embeddings (*dicее.models.base_model.BaseKGE* attribute), 73
 normalize_relation_embeddings (*dicее.models.BaseKGE* attribute), 120, 123, 129, 134, 140, 152, 156
 normalize_tail_entity_embeddings (*dicее.BaseKGE* attribute), 209
 normalize_tail_entity_embeddings (*dicее.models.base_model.BaseKGE* attribute), 73
 normalize_tail_entity_embeddings (*dicее.models.BaseKGE* attribute), 120, 123, 129, 134, 140, 152, 156
 normalizer_class (*dicее.BaseKGE* attribute), 209
 normalizer_class (*dicее.models.base_model.BaseKGE* attribute), 73
 normalizer_class (*dicее.models.BaseKGE* attribute), 120, 123, 129, 134, 140, 152, 156
 num_bpe_entities (*dicее.BPE_NegativeSamplingDataset* attribute), 218
 num_bpe_entities (*dicее.dataset_classes.BPE_NegativeSamplingDataset* attribute), 34
 num_bpe_entities (*dicее.knowledge_graph.KG* attribute), 52
 num_core (*dicее.config.Namespace* attribute), 31
 num_data_properties (*dicее.dataset_classes.LiteralDataset* attribute), 45
 num_data_properties (*dicее.LiteralDataset* attribute), 230
 num_datapoints (*dicее.BPE_NegativeSamplingDataset* attribute), 218
 num_datapoints (*dicее.dataset_classes.BPE_NegativeSamplingDataset* attribute), 34
 num_datapoints (*dicее.dataset_classes.MultiLabelDataset* attribute), 34
 num_datapoints (*dicее.MultiLabelDataset* attribute), 219
 num_ent (*dicее.DualE* attribute), 194
 num_ent (*dicее.models.DualE* attribute), 160
 num_ent (*dicее.models.dualE.DualE* attribute), 85
 num_entities (*dicее.BaseKGE* attribute), 209
 num_entities (*dicее.CVDDataModule* attribute), 226
 num_entities (*dicее.dataset_classes.CVDDataModule* attribute), 41
 num_entities (*dicее.dataset_classes.KvsSampleDataset* attribute), 39
 num_entities (*dicее.dataset_classes.LiteralDataset* attribute), 45, 46
 num_entities (*dicее.dataset_classes.NegSampleDataset* attribute), 40
 num_entities (*dicее.dataset_classes.OnevsSample* attribute), 37, 38
 num_entities (*dicее.dataset_classes.TriplePredictionDataset* attribute), 41
 num_entities (*dicее.evaluator.Evaluator* attribute), 48
 num_entities (*dicее.knowledge_graph.KG* attribute), 52
 num_entities (*dicее.KvsSampleDataset* attribute), 224
 num_entities (*dicее.LiteralDataset* attribute), 230
 num_entities (*dicее.models.base_model.BaseKGE* attribute), 72
 num_entities (*dicее.models.BaseKGE* attribute), 119, 123, 128, 133, 139, 152, 155
 num_entities (*dicее.NegSampleDataset* attribute), 224
 num_entities (*dicее.OnevsSample* attribute), 222, 223
 num_entities (*dicее.TriplePredictionDataset* attribute), 225
 num_epochs (*dicее.abstracts.AbstractPPECallback* attribute), 17
 num_epochs (*dicее.analyse_experiments.Experiment* attribute), 19

- `num_epochs` (*dicee.callbacks.ASWA attribute*), 24
- `num_epochs` (*dicee.config.Namespace attribute*), 30
- `num_epochs` (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 182
- `num_folds_for_cv` (*dicee.config.Namespace attribute*), 31
- `num_of_data_points` (*dicee.dataset_classes.MultiClassClassificationDataset attribute*), 35
- `num_of_data_points` (*dicee.MultiClassClassificationDataset attribute*), 219
- `num_of_data_properties` (*dicee.models.literal.LiteralEmbeddings attribute*), 90, 91
- `num_of_epochs` (*dicee.callbacks.PseudoLabellingCallback attribute*), 24
- `num_of_output_channels` (*dicee.BaseKGE attribute*), 209
- `num_of_output_channels` (*dicee.config.Namespace attribute*), 31
- `num_of_output_channels` (*dicee.models.base_model.BaseKGE attribute*), 72
- `num_of_output_channels` (*dicee.models.BaseKGE attribute*), 120, 123, 129, 133, 139, 152, 156
- `num_params` (*dicee.analyse_experiments.Experiment attribute*), 19
- `num_relations` (*dicee.BaseKGE attribute*), 209
- `num_relations` (*dicee.CVDDataModule attribute*), 226
- `num_relations` (*dicee.dataset_classes.CVDDataModule attribute*), 41
- `num_relations` (*dicee.dataset_classes.NegSampleDataset attribute*), 40
- `num_relations` (*dicee.dataset_classes.OnevsSample attribute*), 38
- `num_relations` (*dicee.dataset_classes.TriplePredictionDataset attribute*), 41
- `num_relations` (*dicee.evaluator.Evaluator attribute*), 48
- `num_relations` (*dicee.knowledge_graph.KG attribute*), 52
- `num_relations` (*dicee.models.base_model.BaseKGE attribute*), 72
- `num_relations` (*dicee.models.BaseKGE attribute*), 119, 123, 128, 133, 139, 152, 155
- `num_relations` (*dicee.NegSampleDataset attribute*), 224
- `num_relations` (*dicee.OnevsSample attribute*), 222, 223
- `num_relations` (*dicee.TriplePredictionDataset attribute*), 225
- `num_sample` (*dicee.models.FMulti attribute*), 157
- `num_sample` (*dicee.models.function_space.FMulti attribute*), 87
- `num_sample` (*dicee.models.function_space.GFMulti attribute*), 87
- `num_sample` (*dicee.models.GFMulti attribute*), 158
- `num_tokens` (*dicee.BaseKGE attribute*), 209
- `num_tokens` (*dicee.knowledge_graph.KG attribute*), 52
- `num_tokens` (*dicee.models.base_model.BaseKGE attribute*), 72
- `num_tokens` (*dicee.models.BaseKGE attribute*), 120, 123, 128, 133, 139, 152, 155
- `num_workers` (*dicee.CVDDataModule attribute*), 226
- `num_workers` (*dicee.dataset_classes.CVDDataModule attribute*), 41
- `numpy_data_type_changer()` (in module *dicee*), 212
- `numpy_data_type_changer()` (in module *dicee.static_funcs*), 174

O

- `octonion_mul()` (in module *dicee.models*), 142
- `octonion_mul()` (in module *dicee.models.octonion*), 92
- `octonion_mul_norm()` (in module *dicee.models*), 142
- `octonion_mul_norm()` (in module *dicee.models.octonion*), 92
- `octonion_normalizer()` (*dicee.AConvO static method*), 197
- `octonion_normalizer()` (*dicee.ConvO static method*), 199
- `octonion_normalizer()` (*dicee.models.AConvO static method*), 144
- `octonion_normalizer()` (*dicee.models.ConvO static method*), 143
- `octonion_normalizer()` (*dicee.models.octonion.AConvO static method*), 94
- `octonion_normalizer()` (*dicee.models.octonion.ConvO static method*), 94
- `octonion_normalizer()` (*dicee.models.octonion.OMult static method*), 92
- `octonion_normalizer()` (*dicee.models.OMult static method*), 142
- `octonion_normalizer()` (*dicee.OMult static method*), 202
- `OMult` (class in *dicee*), 201
- `OMult` (class in *dicee.models*), 142
- `OMult` (class in *dicee.models.octonion*), 92
- `on_epoch_end()` (*dicee.callbacks.KGESaveCallback method*), 24
- `on_epoch_end()` (*dicee.callbacks.PseudoLabellingCallback method*), 24
- `on_fit_end()` (*dicee.abstracts.AbstractCallback method*), 17
- `on_fit_end()` (*dicee.abstracts.AbstractPPECallback method*), 18
- `on_fit_end()` (*dicee.abstracts.AbstractTrainer method*), 13
- `on_fit_end()` (*dicee.callbacks.AccumulateEpochLossCallback method*), 21
- `on_fit_end()` (*dicee.callbacks.ASWA method*), 24
- `on_fit_end()` (*dicee.callbacks.Eval method*), 26
- `on_fit_end()` (*dicee.callbacks.KGESaveCallback method*), 23
- `on_fit_end()` (*dicee.callbacks.LRScheduler method*), 28
- `on_fit_end()` (*dicee.callbacks.PeriodicEvalCallback method*), 28

- `on_fit_end()` (*dicee.callbacks.PrintCallback* method), 22
- `on_fit_end()` (*dicee.callbacks.SWA* method), 29
- `on_fit_start()` (*dicee.abstracts.AbstractCallback* method), 16
- `on_fit_start()` (*dicee.abstracts.AbstractPPECallback* method), 18
- `on_fit_start()` (*dicee.abstracts.AbstractTrainer* method), 13
- `on_fit_start()` (*dicee.callbacks.Eval* method), 25
- `on_fit_start()` (*dicee.callbacks.KGESaveCallback* method), 23
- `on_fit_start()` (*dicee.callbacks.KronE* method), 27
- `on_fit_start()` (*dicee.callbacks.PrintCallback* method), 22
- `on_init_end()` (*dicee.abstracts.AbstractCallback* method), 16
- `on_init_start()` (*dicee.abstracts.AbstractCallback* method), 16
- `on_train_batch_end()` (*dicee.abstracts.AbstractCallback* method), 17
- `on_train_batch_end()` (*dicee.abstracts.AbstractTrainer* method), 13
- `on_train_batch_end()` (*dicee.callbacks.Eval* method), 26
- `on_train_batch_end()` (*dicee.callbacks.KGESaveCallback* method), 23
- `on_train_batch_end()` (*dicee.callbacks.LRScheduler* method), 28
- `on_train_batch_end()` (*dicee.callbacks.PrintCallback* method), 22
- `on_train_batch_start()` (*dicee.callbacks.Perturb* method), 27
- `on_train_epoch_end()` (*dicee.abstracts.AbstractCallback* method), 17
- `on_train_epoch_end()` (*dicee.abstracts.AbstractTrainer* method), 13
- `on_train_epoch_end()` (*dicee.callbacks.ASWA* method), 25
- `on_train_epoch_end()` (*dicee.callbacks.Eval* method), 26
- `on_train_epoch_end()` (*dicee.callbacks.KGESaveCallback* method), 23
- `on_train_epoch_end()` (*dicee.callbacks.PeriodicEvalCallback* method), 28
- `on_train_epoch_end()` (*dicee.callbacks.PrintCallback* method), 22
- `on_train_epoch_end()` (*dicee.callbacks.SWA* method), 29
- `on_train_epoch_end()` (*dicee.models.base_model.BaseKGELightning* method), 68
- `on_train_epoch_end()` (*dicee.models.BaseKGELightning* method), 115
- `on_train_epoch_start()` (*dicee.abstracts.AbstractTrainer* method), 13
- `on_train_epoch_start()` (*dicee.callbacks.SWA* method), 29
- `on_train_start()` (*dicee.callbacks.LRScheduler* method), 28
- `OnevsAllDataset` (class in *dicee*), 220
- `OnevsAllDataset` (class in *dicee.dataset_classes*), 35
- `OnevsSample` (class in *dicee*), 222
- `OnevsSample` (class in *dicee.dataset_classes*), 37
- `optim` (*dicee.config.Namespace* attribute), 30
- `optimizer` (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 181
- `optimizer` (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 180
- `optimizer_name` (*dicee.BaseKGE* attribute), 209
- `optimizer_name` (*dicee.models.base_model.BaseKGE* attribute), 72
- `optimizer_name` (*dicee.models.BaseKGE* attribute), 120, 123, 129, 133, 139, 152, 155
- `ordered_bpe_entities` (*dicee.BPE_NegativeSamplingDataset* attribute), 218
- `ordered_bpe_entities` (*dicee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 34
- `ordered_bpe_entities` (*dicee.knowledge_graph.KG* attribute), 53
- `ordered_shaped_bpe_tokens` (*dicee.knowledge_graph.KG* attribute), 52

P

- `p` (*dicee.config.Namespace* attribute), 31
- `p` (*dicee.DeCaL* attribute), 191
- `p` (*dicee.Keci* attribute), 188
- `p` (*dicee.models.clifford.DeCaL* attribute), 79
- `p` (*dicee.models.clifford.Keci* attribute), 75
- `p` (*dicee.models.DeCaL* attribute), 148
- `p` (*dicee.models.Keci* attribute), 145
- `padding` (*dicee.knowledge_graph.KG* attribute), 52
- `pandas_dataframe_indexer()` (in module *dicee.read_preprocess_save_load_kg.util*), 166
- `param_init` (*dicee.BaseKGE* attribute), 209
- `param_init` (*dicee.models.base_model.BaseKGE* attribute), 73
- `param_init` (*dicee.models.BaseKGE* attribute), 120, 123, 129, 134, 140, 152, 156
- `parameters()` (*dicee.abstracts.BaseInteractiveKGE* method), 16
- `parameters()` (*dicee.EnsembleKGE* method), 211
- `parameters()` (*dicee.models.ensemble.EnsembleKGE* method), 86
- `path` (*dicee.abstracts.AbstractPPECallback* attribute), 17
- `path` (*dicee.callbacks.AccumulateEpochLossCallback* attribute), 21
- `path` (*dicee.callbacks.ASWA* attribute), 24
- `path` (*dicee.callbacks.Eval* attribute), 25
- `path` (*dicee.callbacks.KGESaveCallback* attribute), 23

path_dataset_folder (*dicee.analyse_experiments.Experiment* attribute), 19
 path_for_deserialization (*dicee.knowledge_graph.KG* attribute), 52
 path_for_serialization (*dicee.knowledge_graph.KG* attribute), 52
 path_single_kg (*dicee.config.Namespace* attribute), 30
 path_single_kg (*dicee.knowledge_graph.KG* attribute), 52
 path_to_store_single_run (*dicee.config.Namespace* attribute), 29
 PeriodicEvalCallback (*class in dicee.callbacks*), 27
 Perturb (*class in dicee.callbacks*), 27
 polars_dataframe_indexer() (*in module dicee.read_preprocess_save_load_kg.util*), 165
 poly_NN() (*dicee.LFMMult* method), 203
 poly_NN() (*dicee.models.function_space.LFMMult* method), 89
 poly_NN() (*dicee.models.LFMMult* method), 159
 polynomial() (*dicee.LFMMult* method), 204
 polynomial() (*dicee.models.function_space.LFMMult* method), 89
 polynomial() (*dicee.models.LFMMult* method), 160
 pop() (*dicee.LFMMult* method), 204
 pop() (*dicee.models.function_space.LFMMult* method), 90
 pop() (*dicee.models.LFMMult* method), 160
 pos_emb (*dicee.CoKE* attribute), 205
 pos_emb (*dicee.models.CoKE* attribute), 127
 pos_emb (*dicee.models.real.CoKE* attribute), 101
 pq (*dicee.analyse_experiments.Experiment* attribute), 20
 predict() (*dicee.KGE* method), 216
 predict() (*dicee.knowledge_graph_embeddings.KGE* method), 55
 predict_data_loader() (*dicee.models.base_model.BaseKGELightning* method), 69
 predict_data_loader() (*dicee.models.BaseKGELightning* method), 117
 predict_literals() (*dicee.KGE* method), 217
 predict_literals() (*dicee.knowledge_graph_embeddings.KGE* method), 56
 predict_missing_head_entity() (*dicee.KGE* method), 214
 predict_missing_head_entity() (*dicee.knowledge_graph_embeddings.KGE* method), 53
 predict_missing_relations() (*dicee.KGE* method), 215
 predict_missing_relations() (*dicee.knowledge_graph_embeddings.KGE* method), 54
 predict_missing_tail_entity() (*dicee.KGE* method), 215
 predict_missing_tail_entity() (*dicee.knowledge_graph_embeddings.KGE* method), 54
 predict_topk() (*dicee.KGE* method), 216
 predict_topk() (*dicee.knowledge_graph_embeddings.KGE* method), 55
 prepare_data() (*dicee.CVDataModule* method), 228
 prepare_data() (*dicee.dataset_classes.CVDataModule* method), 43
 preprocess_with_byte_pair_encoding() (*dicee.read_preprocess_save_load_kg.PreprocessKG* method), 168
 preprocess_with_byte_pair_encoding() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 163
 preprocess_with_byte_pair_encoding_with_padding() (*dicee.read_preprocess_save_load_kg.PreprocessKG* method), 169
 preprocess_with_byte_pair_encoding_with_padding() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 163
 preprocess_with_pandas() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 163
 preprocess_with_polars() (*dicee.read_preprocess_save_load_kg.PreprocessKG* method), 169
 preprocess_with_polars() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 163
 preprocesses_input_args() (*in module dicee.static_preprocess_funcs*), 176
 PreprocessKG (*class in dicee.read_preprocess_save_load_kg*), 168
 PreprocessKG (*class in dicee.read_preprocess_save_load_kg.preprocess*), 163
 PrintCallback (*class in dicee.callbacks*), 21
 process (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 180
 PseudoLabellingCallback (*class in dicee.callbacks*), 24
 Pyke (*class in dicee*), 186
 Pyke (*class in dicee.models*), 126
 Pyke (*class in dicee.models.real*), 100
 pykeen_model_kwargs (*dicee.config.Namespace* attribute), 31
 PykeenKGE (*class in dicee*), 205
 PykeenKGE (*class in dicee.models*), 154
 PykeenKGE (*class in dicee.models.pykeen_models*), 95

Q

q (*dicee.config.Namespace* attribute), 31
 q (*dicee.DeCaL* attribute), 191
 q (*dicee.Keci* attribute), 188
 q (*dicee.models.clifford.DeCaL* attribute), 79
 q (*dicee.models.clifford.Keci* attribute), 76
 q (*dicee.models.DeCaL* attribute), 148

- `q` (*dicee.models.Keci* attribute), 145
- `qdrant_client` (*dicee.scripts.index_serve.NeuralSearcher* attribute), 171
- `QMult` (class in *dicee*), 200
- `QMult` (class in *dicee.models*), 136
- `QMult` (class in *dicee.models.quaternion*), 96
- `quaternion_mul()` (in module *dicee.models*), 132
- `quaternion_mul()` (in module *dicee.models.static_funcs*), 102
- `quaternion_mul_with_unit_norm()` (in module *dicee.models*), 136
- `quaternion_mul_with_unit_norm()` (in module *dicee.models.quaternion*), 96
- `quaternion_multiplication_followed_by_inner_product()` (*dicee.models.QMult* method), 136
- `quaternion_multiplication_followed_by_inner_product()` (*dicee.models.quaternion.QMult* method), 97
- `quaternion_multiplication_followed_by_inner_product()` (*dicee.QMult* method), 201
- `quaternion_normalizer()` (*dicee.models.QMult* static method), 136
- `quaternion_normalizer()` (*dicee.models.quaternion.QMult* static method), 97
- `quaternion_normalizer()` (*dicee.QMult* static method), 201
- `queries` (*dicee.scripts.index_serve.StringListRequest* attribute), 171
- `query_name_to_struct` (*dicee.query_generator.QueryGenerator* attribute), 162
- `query_name_to_struct` (*dicee.QueryGenerator* attribute), 231
- `QueryGenerator` (class in *dicee*), 231
- `QueryGenerator` (class in *dicee.query_generator*), 161

R

- `r` (*dicee.DeCaL* attribute), 191
- `r` (*dicee.Keci* attribute), 188
- `r` (*dicee.models.clifford.DeCaL* attribute), 79
- `r` (*dicee.models.clifford.Keci* attribute), 76
- `r` (*dicee.models.DeCaL* attribute), 148
- `r` (*dicee.models.Keci* attribute), 145
- `random_prediction()` (in module *dicee*), 212
- `random_prediction()` (in module *dicee.static_funcs*), 174
- `random_seed` (*dicee.config.Namespace* attribute), 31
- `ratio` (*dicee.callbacks.Perturb* attribute), 27
- `re` (*dicee.DeCaL* attribute), 191
- `re` (*dicee.models.clifford.DeCaL* attribute), 79
- `re` (*dicee.models.DeCaL* attribute), 148
- `re_vocab` (*dicee.evaluator.Evaluator* attribute), 48
- `read_from_disk()` (in module *dicee.read_preprocess_save_load_kg.util*), 167
- `read_from_triple_store_with_pandas()` (in module *dicee.read_preprocess_save_load_kg.util*), 167
- `read_from_triple_store_with_polars()` (in module *dicee.read_preprocess_save_load_kg.util*), 167
- `read_only_few` (*dicee.config.Namespace* attribute), 31
- `read_only_few` (*dicee.knowledge_graph.KG* attribute), 52
- `read_or_load_kg()` (in module *dicee*), 212
- `read_or_load_kg()` (in module *dicee.static_funcs*), 174
- `read_with_pandas()` (in module *dicee.read_preprocess_save_load_kg.util*), 167
- `read_with_polars()` (in module *dicee.read_preprocess_save_load_kg.util*), 167
- `ReadFromDisk` (class in *dicee.read_preprocess_save_load_kg*), 169
- `ReadFromDisk` (class in *dicee.read_preprocess_save_load_kg.read_from_disk*), 164
- `reducer` (*dicee.scripts.index_serve.StringListRequest* attribute), 171
- `rel2id` (*dicee.query_generator.QueryGenerator* attribute), 162
- `rel2id` (*dicee.QueryGenerator* attribute), 231
- `relation_embeddings` (*dicee.AConvQ* attribute), 197
- `relation_embeddings` (*dicee.ConvQ* attribute), 198
- `relation_embeddings` (*dicee.DeCaL* attribute), 191
- `relation_embeddings` (*dicee.DualE* attribute), 194
- `relation_embeddings` (*dicee.LFMult* attribute), 203
- `relation_embeddings` (*dicee.models.AConvQ* attribute), 138
- `relation_embeddings` (*dicee.models.clifford.DeCaL* attribute), 79
- `relation_embeddings` (*dicee.models.ConvQ* attribute), 137
- `relation_embeddings` (*dicee.models.DeCaL* attribute), 148
- `relation_embeddings` (*dicee.models.DualE* attribute), 160
- `relation_embeddings` (*dicee.models.dualE.DualE* attribute), 85
- `relation_embeddings` (*dicee.models.FMult* attribute), 157
- `relation_embeddings` (*dicee.models.FMult2* attribute), 158
- `relation_embeddings` (*dicee.models.function_space.FMult* attribute), 87
- `relation_embeddings` (*dicee.models.function_space.FMult2* attribute), 88
- `relation_embeddings` (*dicee.models.function_space.GFMult* attribute), 87
- `relation_embeddings` (*dicee.models.function_space.LFMult* attribute), 89

- `relation_embeddings` (*dicee.models.function_space.LFMultI attribute*), 88
- `relation_embeddings` (*dicee.models.GFMult attribute*), 158
- `relation_embeddings` (*dicee.models.LFMult attribute*), 159
- `relation_embeddings` (*dicee.models.LFMultI attribute*), 159
- `relation_embeddings` (*dicee.models.pykeen_models.PykeenKGE attribute*), 95
- `relation_embeddings` (*dicee.models.PykeenKGE attribute*), 154
- `relation_embeddings` (*dicee.models.quaternion.AConvQ attribute*), 98
- `relation_embeddings` (*dicee.models.quaternion.ConvQ attribute*), 98
- `relation_embeddings` (*dicee.PykeenKGE attribute*), 205
- `relation_to_idx` (*dicee.knowledge_graph.KG attribute*), 52
- `relations_str` (*dicee.knowledge_graph.KG property*), 53
- `reload_dataset()` (*in module dicee*), 218
- `reload_dataset()` (*in module dicee.dataset_classes*), 33
- `report` (*dicee.DICE_Trainer attribute*), 213
- `report` (*dicee.evaluator.Evaluator attribute*), 48
- `report` (*dicee.executer.Execute attribute*), 50
- `report` (*dicee.trainer.DICE_Trainer attribute*), 182
- `report` (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 177
- `reports` (*dicee.callbacks.Eval attribute*), 25
- `reports` (*dicee.callbacks.PeriodicEvalCallback attribute*), 27
- `requires_grad_for_interactions` (*dicee.CKeci attribute*), 187
- `requires_grad_for_interactions` (*dicee.Keci attribute*), 188
- `requires_grad_for_interactions` (*dicee.models.CKeci attribute*), 147
- `requires_grad_for_interactions` (*dicee.models.clifford.CKeci attribute*), 78
- `requires_grad_for_interactions` (*dicee.models.clifford.Keci attribute*), 76
- `requires_grad_for_interactions` (*dicee.models.Keci attribute*), 145
- `resid_dropout` (*dicee.models.transformers.SelfAttention attribute*), 105
- `residual_convolution()` (*dicee.AConEx method*), 196
- `residual_convolution()` (*dicee.AConvO method*), 197
- `residual_convolution()` (*dicee.AConvQ method*), 197
- `residual_convolution()` (*dicee.ConEx method*), 200
- `residual_convolution()` (*dicee.ConvO method*), 199
- `residual_convolution()` (*dicee.ConvQ method*), 198
- `residual_convolution()` (*dicee.models.AConEx method*), 131
- `residual_convolution()` (*dicee.models.AConvO method*), 144
- `residual_convolution()` (*dicee.models.AConvQ method*), 138
- `residual_convolution()` (*dicee.models.complex.AConEx method*), 83
- `residual_convolution()` (*dicee.models.complex.ConEx method*), 82
- `residual_convolution()` (*dicee.models.ConEx method*), 130
- `residual_convolution()` (*dicee.models.ConvO method*), 143
- `residual_convolution()` (*dicee.models.ConvQ method*), 138
- `residual_convolution()` (*dicee.models.octonion.AConvO method*), 94
- `residual_convolution()` (*dicee.models.octonion.ConvO method*), 94
- `residual_convolution()` (*dicee.models.quaternion.AConvQ method*), 98
- `residual_convolution()` (*dicee.models.quaternion.ConvQ method*), 98
- `retrieve_embedding()` (*dicee.scripts.index_serve.NeuralSearcher method*), 171
- `retrieve_embeddings()` (*in module dicee.scripts.index_serve*), 171
- `return_multi_hop_query_results()` (*dicee.KGE method*), 216
- `return_multi_hop_query_results()` (*dicee.knowledge_graph_embeddings.KGE method*), 55
- `root()` (*in module dicee.scripts.index_serve*), 171
- `roots` (*dicee.models.FMult attribute*), 157
- `roots` (*dicee.models.function_space.FMult attribute*), 87
- `roots` (*dicee.models.function_space.GFMult attribute*), 87
- `roots` (*dicee.models.GFMult attribute*), 158
- `runtime` (*dicee.analyse_experiments.Experiment attribute*), 20

S

- `sample_counter` (*dicee.abstracts.AbstractPPECallback attribute*), 17
- `sample_entity()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
- `sample_relation()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
- `sample_triples_ratio` (*dicee.config.Namespace attribute*), 31
- `sample_triples_ratio` (*dicee.knowledge_graph.KG attribute*), 52
- `sampling_ratio` (*dicee.dataset_classes.LiteralDataset attribute*), 45, 46
- `sampling_ratio` (*dicee.LiteralDataset attribute*), 229, 230
- `sanity_check_callback_args()` (*in module dicee.sanity_checkers*), 170
- `sanity_checking_with_arguments()` (*in module dicee.sanity_checkers*), 170
- `save()` (*dicee.abstracts.BaseInteractiveKGE method*), 15

`save()` (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk method*), 169
`save()` (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk method*), 164
`save_checkpoint()` (*dicee.abstracts.AbstractTrainer static method*), 14
`save_checkpoint_model()` (*in module dicee*), 212
`save_checkpoint_model()` (*in module dicee.static_funcs*), 174
`save_embeddings()` (*in module dicee*), 212
`save_embeddings()` (*in module dicee.static_funcs*), 174
`save_embeddings_as_csv()` (*dicee.config.Namespace attribute*), 29
`save_every_n_epochs()` (*dicee.config.Namespace attribute*), 32
`save_experiment()` (*dicee.analyse_experiments.Experiment method*), 20
`save_model_at_every_epoch()` (*dicee.config.Namespace attribute*), 31
`save_model_every_n_epoch()` (*dicee.callbacks.PeriodicEvalCallback attribute*), 27
`save_numpy_ndarray()` (*in module dicee*), 212
`save_numpy_ndarray()` (*in module dicee.read_preprocess_save_load_kg.util*), 168
`save_numpy_ndarray()` (*in module dicee.static_funcs*), 174
`save_pickle()` (*in module dicee*), 211
`save_pickle()` (*in module dicee.read_preprocess_save_load_kg.util*), 168
`save_pickle()` (*in module dicee.static_funcs*), 174
`save_queries()` (*dicee.query_generator.QueryGenerator method*), 162
`save_queries()` (*dicee.QueryGenerator method*), 232
`save_queries_and_answers()` (*dicee.query_generator.QueryGenerator static method*), 162
`save_queries_and_answers()` (*dicee.QueryGenerator static method*), 232
`save_trained_model()` (*dicee.executer.Execute method*), 50
`scalar_batch_NN()` (*dicee.LFMMult method*), 203
`scalar_batch_NN()` (*dicee.models.function_space.LFMMult method*), 89
`scalar_batch_NN()` (*dicee.models.LFMMult method*), 159
`scaler` (*dicee.callbacks.Perturb attribute*), 27
`scaler` (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 182
`scheduler` (*dicee.callbacks.LRScheduler attribute*), 28
`score()` (*dicee.CoKE method*), 205
`score()` (*dicee.ComplEx static method*), 196
`score()` (*dicee.DistMult method*), 187
`score()` (*dicee.Keci method*), 190
`score()` (*dicee.models.clifford.Keci method*), 78
`score()` (*dicee.models.CoKE method*), 128
`score()` (*dicee.models.ComplEx static method*), 132
`score()` (*dicee.models.complex.ComplEx static method*), 84
`score()` (*dicee.models.DistMult method*), 126
`score()` (*dicee.models.Keci method*), 147
`score()` (*dicee.models.octonion.OMult method*), 92
`score()` (*dicee.models.OMult method*), 142
`score()` (*dicee.models.QMult method*), 137
`score()` (*dicee.models.quaternion.QMult method*), 97
`score()` (*dicee.models.real.CoKE method*), 101
`score()` (*dicee.models.real.DistMult method*), 99
`score()` (*dicee.models.real.TransE method*), 100
`score()` (*dicee.models.TransE method*), 126
`score()` (*dicee.OMult method*), 202
`score()` (*dicee.QMult method*), 201
`score()` (*dicee.TransE method*), 190
`score_func` (*dicee.models.FMMult attribute*), 158
`score_func` (*dicee.models.function_space.FMMult attribute*), 88
`scoring_technique` (*dicee.analyse_experiments.Experiment attribute*), 20
`scoring_technique` (*dicee.config.Namespace attribute*), 30
`search()` (*dicee.scripts.index_serve.NeuralSearcher method*), 171
`search_embeddings()` (*in module dicee.scripts.index_serve*), 171
`search_embeddings_batch()` (*in module dicee.scripts.index_serve*), 171
`seed` (*dicee.query_generator.QueryGenerator attribute*), 162
`seed` (*dicee.QueryGenerator attribute*), 231
`select_model()` (*in module dicee*), 211
`select_model()` (*in module dicee.static_funcs*), 174
`selected_optimizer` (*dicee.BaseKGE attribute*), 209
`selected_optimizer` (*dicee.models.base_model.BaseKGE attribute*), 73
`selected_optimizer` (*dicee.models.BaseKGE attribute*), 120, 123, 129, 133, 139, 152, 156
`SelfAttention` (*class in dicee.models.transformers*), 104
`separator` (*dicee.config.Namespace attribute*), 30
`separator` (*dicee.knowledge_graph.KG attribute*), 53
`sequential_vocabulary_construction()` (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 169

`sequential_vocabulary_construction()` (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 163
`serve()` (in module *dicee.scripts.index_serve*), 172
`set_global_seed()` (*dicee.query_generator.QueryGenerator method*), 162
`set_global_seed()` (*dicee.QueryGenerator method*), 231
`set_model_eval_mode()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`set_model_train_mode()` (*dicee.abstracts.BaseInteractiveKGE method*), 14
`setup()` (*dicee.CVDataModule method*), 227
`setup()` (*dicee.dataset_classes.CVDataModule method*), 42
`setup_executor()` (*dicee.executor.Execute method*), 50
`Shallom` (class in *dicee*), 202
`Shallom` (class in *dicee.models*), 126
`Shallom` (class in *dicee.models.real*), 100
`shallom` (*dicee.models.real.Shallom attribute*), 100
`shallom` (*dicee.models.Shallom attribute*), 126
`shallom` (*dicee.Shallom attribute*), 202
`single_hop_query_answering()` (*dicee.KGE method*), 216
`single_hop_query_answering()` (*dicee.knowledge_graph_embeddings.KGE method*), 55
`snapshot_dir` (*dicee.callbacks.LRScheduler attribute*), 28
`snapshot_loss` (*dicee.callbacks.LRScheduler attribute*), 28
`sparql_endpoint` (*dicee.config.Namespace attribute*), 30
`sparql_endpoint` (*dicee.knowledge_graph.KG attribute*), 52
`start()` (*dicee.DICE_Trainer method*), 214
`start()` (*dicee.executor.Execute method*), 51
`start()` (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 168
`start()` (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 163
`start()` (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method*), 164
`start()` (*dicee.read_preprocess_save_load_kg.ReadFromDisk method*), 169
`start()` (*dicee.trainer.DICE_Trainer method*), 183
`start()` (*dicee.trainer.dice_trainer.DICE_Trainer method*), 178
`start_time` (*dicee.callbacks.PrintCallback attribute*), 22
`start_time` (*dicee.executor.Execute attribute*), 50
`state_dict()` (*dicee.EnsembleKGE method*), 211
`state_dict()` (*dicee.models.ensemble.EnsembleKGE method*), 86
`step()` (*dicee.EnsembleKGE method*), 211
`step()` (*dicee.models.ADOPT method*), 112
`step()` (*dicee.models.adopt.ADOPT method*), 60
`step()` (*dicee.models.ensemble.EnsembleKGE method*), 86
`step_count` (*dicee.callbacks.LRScheduler attribute*), 28
`storage_path` (*dicee.config.Namespace attribute*), 29
`storage_path` (*dicee.DICE_Trainer attribute*), 213
`storage_path` (*dicee.trainer.DICE_Trainer attribute*), 182
`storage_path` (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 178
`store()` (in module *dicee*), 212
`store()` (in module *dicee.static_funcs*), 174
`store_ensemble()` (*dicee.abstracts.AbstractPPECallback method*), 18
`strategy` (*dicee.abstracts.AbstractTrainer attribute*), 13
`StringListRequest` (class in *dicee.scripts.index_serve*), 171
`SWA` (class in *dicee.callbacks*), 28
`swa` (*dicee.config.Namespace attribute*), 32
`swa_c_epochs` (*dicee.callbacks.SWA attribute*), 29
`swa_lr` (*dicee.callbacks.SWA attribute*), 29
`swa_model` (*dicee.callbacks.SWA attribute*), 29
`swa_n` (*dicee.callbacks.SWA attribute*), 29
`swa_start_epoch` (*dicee.callbacks.SWA attribute*), 29
`swa_start_epoch` (*dicee.config.Namespace attribute*), 32

T

`T()` (*dicee.DualE method*), 195
`T()` (*dicee.models.DualE method*), 161
`T()` (*dicee.models.dualE.DualE method*), 85
`t_conorm()` (*dicee.abstracts.InteractiveQueryDecomposition method*), 16
`t_norm()` (*dicee.abstracts.InteractiveQueryDecomposition method*), 16
`target_dim` (*dicee.AllvsAll attribute*), 221
`target_dim` (*dicee.dataset_classes.AllvsAll attribute*), 37
`target_dim` (*dicee.dataset_classes.MultiLabelDataset attribute*), 34
`target_dim` (*dicee.dataset_classes.OnevsAllDataset attribute*), 35
`target_dim` (*dicee.knowledge_graph.KG attribute*), 52

target_dim (*dicee.MultiLabelDataset* attribute), 219
 target_dim (*dicee.OnevsAllDataset* attribute), 220
 temperature (*dicee.BytE* attribute), 207
 temperature (*dicee.models.transformers.BytE* attribute), 103
 tensor_t_norm() (*dicee.abstracts.InteractiveQueryDecomposition* method), 16
 TensorParallel (class in *dicee.trainer.model_parallelism*), 179
 test_data_loader() (*dicee.models.base_model.BaseKGELightning* method), 68
 test_data_loader() (*dicee.models.BaseKGELightning* method), 115
 test_epoch_end() (*dicee.models.base_model.BaseKGELightning* method), 68
 test_epoch_end() (*dicee.models.BaseKGELightning* method), 115
 test_h1 (*dicee.analyse_experiments.Experiment* attribute), 20
 test_h3 (*dicee.analyse_experiments.Experiment* attribute), 20
 test_h10 (*dicee.analyse_experiments.Experiment* attribute), 20
 test_mrr (*dicee.analyse_experiments.Experiment* attribute), 20
 test_path (*dicee.query_generator.QueryGenerator* attribute), 161
 test_path (*dicee.QueryGenerator* attribute), 231
 timeit() (in module *dicee*), 211, 218
 timeit() (in module *dicee.read_preprocess_save_load_kg.util*), 167
 timeit() (in module *dicee.static_funcs*), 173
 timeit() (in module *dicee.static_preprocess_funcs*), 176
 to() (*dicee.EnsembleKGE* method), 211
 to() (*dicee.KGE* method), 214
 to() (*dicee.knowledge_graph_embeddings.KGE* method), 53
 to() (*dicee.models.ensemble.EnsembleKGE* method), 86
 to_df() (*dicee.analyse_experiments.Experiment* method), 20
 topk (*dicee.BytE* attribute), 207
 topk (*dicee.models.transformers.BytE* attribute), 103
 topk (*dicee.scripts.index_serve.NeuralSearcher* attribute), 171
 torch_ordered_shaped_bpe_entities (*dicee.dataset_classes.MultiLabelDataset* attribute), 34
 torch_ordered_shaped_bpe_entities (*dicee.MultiLabelDataset* attribute), 219
 TorchDDPTrainer (class in *dicee.trainer.torch_trainer_ddp*), 181
 TorchTrainer (class in *dicee.trainer.torch_trainer*), 180
 total_epochs (*dicee.callbacks.LRScheduler* attribute), 28
 total_steps (*dicee.callbacks.LRScheduler* attribute), 28
 train() (*dicee.abstracts.BaseInteractiveTrainKGE* method), 18
 train() (*dicee.trainer.torch_trainer_ddp.NodeTrainer* method), 182
 train_data (*dicee.AllvsAll* attribute), 221
 train_data (*dicee.dataset_classes.AllvsAll* attribute), 37
 train_data (*dicee.dataset_classes.KvsAll* attribute), 36
 train_data (*dicee.dataset_classes.KvsSampleDataset* attribute), 39
 train_data (*dicee.dataset_classes.MultiClassClassificationDataset* attribute), 35
 train_data (*dicee.dataset_classes.OnevsAllDataset* attribute), 35
 train_data (*dicee.dataset_classes.OnevsSample* attribute), 37, 38
 train_data (*dicee.KvsAll* attribute), 221
 train_data (*dicee.KvsSampleDataset* attribute), 224
 train_data (*dicee.MultiClassClassificationDataset* attribute), 219
 train_data (*dicee.OnevsAllDataset* attribute), 220
 train_data (*dicee.OnevsSample* attribute), 222, 223
 train_data_loader() (*dicee.CVDataModule* method), 226
 train_data_loader() (*dicee.dataset_classes.CVDataModule* method), 41
 train_data_loader() (*dicee.models.base_model.BaseKGELightning* method), 69
 train_data_loader() (*dicee.models.BaseKGELightning* method), 117
 train_data_loaders (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 180
 train_dataset_loader (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 182
 train_file_path (*dicee.dataset_classes.LiteralDataset* attribute), 45
 train_file_path (*dicee.LiteralDataset* attribute), 229, 230
 train_h1 (*dicee.analyse_experiments.Experiment* attribute), 20
 train_h3 (*dicee.analyse_experiments.Experiment* attribute), 20
 train_h10 (*dicee.analyse_experiments.Experiment* attribute), 20
 train_indices_target (*dicee.dataset_classes.MultiLabelDataset* attribute), 34
 train_indices_target (*dicee.MultiLabelDataset* attribute), 219
 train_k_vs_all() (*dicee.abstracts.BaseInteractiveTrainKGE* method), 18
 train_literals() (*dicee.abstracts.BaseInteractiveTrainKGE* method), 18
 train_mode (*dicee.EnsembleKGE* attribute), 210
 train_mode (*dicee.models.ensemble.EnsembleKGE* attribute), 86
 train_mrr (*dicee.analyse_experiments.Experiment* attribute), 20
 train_path (*dicee.query_generator.QueryGenerator* attribute), 161
 train_path (*dicee.QueryGenerator* attribute), 231

train_set (*dicee.BPE_NegativeSamplingDataset* attribute), 218
train_set (*dicee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 34
train_set (*dicee.dataset_classes.MultiLabelDataset* attribute), 34
train_set (*dicee.dataset_classes.NegSampleDataset* attribute), 40
train_set (*dicee.dataset_classes.TriplePredictionDataset* attribute), 41
train_set (*dicee.MultiLabelDataset* attribute), 219
train_set (*dicee.NegSampleDataset* attribute), 224
train_set (*dicee.TriplePredictionDataset* attribute), 225
train_set_idx (*dicee.CVDataModule* attribute), 226
train_set_idx (*dicee.dataset_classes.CVDataModule* attribute), 41
train_set_target (*dicee.knowledge_graph.KG* attribute), 52
train_target (*dicee.AllvsAll* attribute), 221
train_target (*dicee.dataset_classes.AllvsAll* attribute), 37
train_target (*dicee.dataset_classes.KvsAll* attribute), 36
train_target (*dicee.dataset_classes.KvsSampleDataset* attribute), 39
train_target (*dicee.KvsAll* attribute), 221
train_target (*dicee.KvsSampleDataset* attribute), 224
train_target_indices (*dicee.knowledge_graph.KG* attribute), 53
train_triples (*dicee.dataset_classes.NegSampleDataset* attribute), 40
train_triples (*dicee.NegSampleDataset* attribute), 224
train_triples() (*dicee.abstracts.BaseInteractiveTrainKGE* method), 18
trained_model (*dicee.executer.Execute* attribute), 50
trainer (*dicee.config.Namespace* attribute), 30
trainer (*dicee.DICE_Trainer* attribute), 213
trainer (*dicee.executer.Execute* attribute), 50
trainer (*dicee.trainer.DICE_Trainer* attribute), 182
trainer (*dicee.trainer.dice_trainer.DICE_Trainer* attribute), 177
trainer (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 181
training_step (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 180
training_step() (*dicee.BytE* method), 207
training_step() (*dicee.models.base_model.BaseKGELightning* method), 66
training_step() (*dicee.models.BaseKGELightning* method), 114
training_step() (*dicee.models.transformers.BytE* method), 103
training_step_outputs (*dicee.models.base_model.BaseKGELightning* attribute), 66
training_step_outputs (*dicee.models.BaseKGELightning* attribute), 114
training_technique (*dicee.knowledge_graph.KG* attribute), 52
TransE (class in *dicee*), 190
TransE (class in *dicee.models*), 126
TransE (class in *dicee.models.real*), 99
transfer_batch_to_device() (*dicee.CVDataModule* method), 227
transfer_batch_to_device() (*dicee.dataset_classes.CVDataModule* method), 42
transformer (*dicee.BytE* attribute), 207
transformer (*dicee.models.transformers.BytE* attribute), 103
transformer (*dicee.models.transformers.GPT* attribute), 108
trapezoid() (*dicee.models.FMult2* method), 158
trapezoid() (*dicee.models.function_space.FMult2* method), 88
tri_score() (*dicee.LFMult* method), 203
tri_score() (*dicee.models.function_space.LFMult* method), 89
tri_score() (*dicee.models.function_space.LFMult1* method), 88
tri_score() (*dicee.models.LFMult* method), 160
tri_score() (*dicee.models.LFMult1* method), 159
triple_score() (*dicee.KGE* method), 216
triple_score() (*dicee.knowledge_graph_embeddings.KGE* method), 55
TriplePredictionDataset (class in *dicee*), 225
TriplePredictionDataset (class in *dicee.dataset_classes*), 40
tuple2list() (*dicee.query_generator.QueryGenerator* method), 162
tuple2list() (*dicee.QueryGenerator* method), 231

U

unlabelled_size (*dicee.callbacks.PseudoLabellingCallback* attribute), 24
unmap() (*dicee.query_generator.QueryGenerator* method), 162
unmap() (*dicee.QueryGenerator* method), 232
unmap_query() (*dicee.query_generator.QueryGenerator* method), 162
unmap_query() (*dicee.QueryGenerator* method), 232

V

val_aswa (*dicee.callbacks.ASWA* attribute), 24

val_data_loader() (*dictee.models.base_model.BaseKGELighting method*), 69
 val_data_loader() (*dictee.models.BaseKGELighting method*), 116
 val_h1 (*dictee.analyse_experiments.Experiment attribute*), 20
 val_h3 (*dictee.analyse_experiments.Experiment attribute*), 20
 val_h10 (*dictee.analyse_experiments.Experiment attribute*), 20
 val_mrr (*dictee.analyse_experiments.Experiment attribute*), 20
 val_path (*dictee.query_generator.QueryGenerator attribute*), 161
 val_path (*dictee.QueryGenerator attribute*), 231
 validate_knowledge_graph() (*in module dictee.sanity_checkers*), 170
 vocab_preparation() (*dictee.evaluator.Evaluator method*), 49
 vocab_size (*dictee.models.CoKEConfig attribute*), 127
 vocab_size (*dictee.models.real.CoKEConfig attribute*), 100
 vocab_size (*dictee.models.transformers.GPTConfig attribute*), 107
 vocab_to_parquet() (*in module dictee*), 212
 vocab_to_parquet() (*in module dictee.static_funcs*), 175
 vtp_score() (*dictee.LFMMult method*), 203
 vtp_score() (*dictee.models.function_space.LFMMult method*), 89
 vtp_score() (*dictee.models.function_space.LFMMult1 method*), 88
 vtp_score() (*dictee.models.LFMMult method*), 160
 vtp_score() (*dictee.models.LFMMult1 method*), 159

W

warmup_steps (*dictee.callbacks.LRScheduler attribute*), 28
 weight (*dictee.models.transformers.LayerNorm attribute*), 104
 weight_decay (*dictee.BaseKGE attribute*), 209
 weight_decay (*dictee.config.Namespace attribute*), 30
 weight_decay (*dictee.models.base_model.BaseKGE attribute*), 72
 weight_decay (*dictee.models.BaseKGE attribute*), 120, 123, 129, 133, 139, 152, 156
 weights (*dictee.models.FMMult attribute*), 157
 weights (*dictee.models.function_space.FMMult attribute*), 87
 weights (*dictee.models.function_space.GFMMult attribute*), 87
 weights (*dictee.models.GFMMult attribute*), 158
 write_csv_from_model_parallel() (*in module dictee*), 213
 write_csv_from_model_parallel() (*in module dictee.static_funcs*), 175
 write_links() (*dictee.query_generator.QueryGenerator method*), 162
 write_links() (*dictee.QueryGenerator method*), 231
 write_report() (*dictee.executer.Execute method*), 51

X

x_values (*dictee.LFMMult attribute*), 203
 x_values (*dictee.models.function_space.LFMMult attribute*), 89
 x_values (*dictee.models.LFMMult attribute*), 159