

---

# DICE Embeddings

*Release 0.1.3.2*

**Caglar Demir**

**Nov 19, 2025**

## Contents:

<b>1</b>	<b>Dicee Manual</b>	<b>2</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	Installation from Source . . . . .	3
<b>3</b>	<b>Download Knowledge Graphs</b>	<b>3</b>
<b>4</b>	<b>Knowledge Graph Embedding Models</b>	<b>3</b>
<b>5</b>	<b>How to Train</b>	<b>3</b>
<b>6</b>	<b>Creating an Embedding Vector Database</b>	<b>5</b>
6.1	Learning Embeddings . . . . .	5
6.2	Loading Embeddings into Qdrant Vector Database . . . . .	6
6.3	Launching Webservice . . . . .	6
<b>7</b>	<b>Answering Complex Queries</b>	<b>6</b>
<b>8</b>	<b>Predicting Missing Links</b>	<b>8</b>
<b>9</b>	<b>Downloading Pretrained Models</b>	<b>8</b>
<b>10</b>	<b>How to Deploy</b>	<b>8</b>
<b>11</b>	<b>Docker</b>	<b>8</b>
<b>12</b>	<b>Coverage Report</b>	<b>8</b>
<b>13</b>	<b>How to cite</b>	<b>10</b>
<b>14</b>	<b>dicee</b>	<b>12</b>
14.1	Submodules . . . . .	12
14.2	Attributes . . . . .	179
14.3	Classes . . . . .	180
14.4	Functions . . . . .	181
14.5	Package Contents . . . . .	182
	<b>Python Module Index</b>	<b>228</b>

DICE Embeddings<sup>1</sup>: Hardware-agnostic Framework for Large-scale Knowledge Graph Embeddings:

## 1 Dicee Manual

**Version:** dicee 0.2.0

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

**Publisher and maintainer:** Caglar Demir<sup>2</sup>

**Contact:** [caglar.demir@upb.de](mailto: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**<sup>3</sup> & Co. to use parallelism at preprocessing a large knowledge graph,
2. **PyTorch**<sup>4</sup> & Co. to learn knowledge graph embeddings via multi-CPUs, GPUs, TPUs or computing cluster, and
3. **Huggingface**<sup>5</sup> to ease the deployment of pre-trained models.

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

**Why PyTorch<sup>7</sup> & 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**<sup>8</sup> & **PytorchLightning**<sup>9</sup>. Users can choose the trainer class (e.g., DDP by Pytorch) to train large knowledge graph embedding models with billions of parameters. PytorchLightning allows us to use state-of-the-art model parallelism techniques (e.g. Fully Sharded Training, FairScale, or DeepSpeed) without extra effort. With our framework, practitioners can directly use PytorchLightning for model parallelism to train gigantic embedding models.

**Why Hugging-face Gradio<sup>10</sup>?** Deploy a pre-trained embedding model without writing a single line of code.

<sup>1</sup> <https://github.com/dice-group/dice-embeddings>

<sup>2</sup> <https://github.com/Demirrr>

<sup>3</sup> <https://pandas.pydata.org/>

<sup>4</sup> <https://pytorch.org/>

<sup>5</sup> <https://huggingface.co/>

<sup>6</sup> <https://pandas.pydata.org/>

<sup>7</sup> <https://pytorch.org/>

<sup>8</sup> <https://pytorch.org/>

<sup>9</sup> <https://www.pytorchlightning.ai/>

<sup>10</sup> <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/](https://dice-research.org/projects/DiceEmbeddings/)<sup>11</sup>

## 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`<sup>12</sup>:

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

(continues on next page)

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

<sup>12</sup> <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](mailto: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** `dicce.callbacks.Perturb` (*level: str = 'input', ratio: float = 0.0, method: str = None, scaler: float = None, frequency=None*)

Bases: `dicce.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** `dicce.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: `dicce.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_idx
```

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.

forall  $y_i = 1$  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.

forall  $y_i = 1$  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_idx,
      relation_idx, 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\_idx**

mapping.

**relation\_idx**

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.CVDataModule(train_set_idx: numpy.ndarray, num_entities,
    num_relations, neg_sample_ratio, batch_size, num_workers)
Bases: pytorch_lightning.LightningDataModule
Create a Dataset for cross validation

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

Return type
    ?

train_set_idx
num_entities
num_relations
neg_sample_ratio
batch_size
num_workers

```

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

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **: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 :

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} ∩ (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.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:

- $\theta_t$ : parameter at step  $t$
- $g_t$ : gradient at step  $t$
- $m_t$ : first moment estimate (momentum)
- $v_t$ : second moment estimate (variance)
- $\alpha$ : learning rate
- $\beta_1, \beta_2$ : 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 ( $\beta_1, \beta_2$ ) for computing running averages of gradient and its square.  $\beta_1$  controls momentum,  $\beta_2$  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:

- $\theta$ : parameters
- $g$ : gradients
- $m$ : first moment (momentum of normalized gradients)
- $v$ : second moment (variance of raw gradients)
- $\alpha$ : learning rate
- $\beta_1, \beta_2$ : exponential decay rates
- $\epsilon$ : 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  $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_{i,r,k} - h_{k,r,i}) e_i e_k$

$\sigma_{pp}$  captures the interactions between along p bases For instance, let p e<sub>1</sub>, e<sub>2</sub>, e<sub>3</sub>, we compute interactions between e<sub>1</sub> e<sub>2</sub>, e<sub>1</sub> e<sub>3</sub>, and e<sub>2</sub> e<sub>3</sub> 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<sub>1</sub>e<sub>1</sub>, e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>,

e<sub>2</sub>e<sub>1</sub>, e<sub>2</sub>e<sub>2</sub>, e<sub>2</sub>e<sub>3</sub>, e<sub>3</sub>e<sub>1</sub>, e<sub>3</sub>e<sub>2</sub>, e<sub>3</sub>e<sub>3</sub>

Then select the triangular matrix without diagonals: e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>, e<sub>2</sub>e<sub>3</sub>.

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

Compute  $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_{j,r,k} - h_{k,r,j}) e_j e_k$   $\sigma_{qq}$  captures the interactions between along q bases For instance, let q e<sub>1</sub>, e<sub>2</sub>, e<sub>3</sub>, we compute interactions between e<sub>1</sub> e<sub>2</sub>, e<sub>1</sub> e<sub>3</sub>, and e<sub>2</sub> e<sub>3</sub> 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<sub>1</sub>e<sub>1</sub>, e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>,

e<sub>2</sub>e<sub>1</sub>, e<sub>2</sub>e<sub>2</sub>, e<sub>2</sub>e<sub>3</sub>, e<sub>3</sub>e<sub>1</sub>, e<sub>3</sub>e<sub>2</sub>, e<sub>3</sub>e<sub>3</sub>

Then select the triangular matrix without diagonals: e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>, e<sub>2</sub>e<sub>3</sub>.

**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

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

For different base vector interactions, we have

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

**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:

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

$\sigma_{pp}$  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 Em- beddings
<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>ACovQ</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 ( <a href="https://arxiv.org/abs/2101.09090">https://arxiv.org/abs/2101.09090</a> )
<i>Pyke</i>	A Physical Embedding Model for Knowledge Graphs

### Module Contents

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

Bases: *dicee.models.base\_model.BaseKGE*

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

**name** = 'DistMult'

**k\_vs\_all\_score** (*emb\_h: torch.FloatTensor, emb\_r: torch.FloatTensor, emb\_E: torch.FloatTensor*)

#### Parameters

- **emb\_h**
- **emb\_r**
- **emb\_E**

**forward\_k\_vs\_all** (*x: torch.LongTensor*)

**forward\_k\_vs\_sample** (*x: torch.LongTensor, target\_entity\_idx: torch.LongTensor*)

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

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

Bases: *dicee.models.base\_model.BaseKGE*

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

**name** = 'TransE'

**margin** = 4

```

    score(head_ent_emb, rel_ent_emb, tail_ent_emb)

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

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

    name = 'Shallom'

    shallom

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

    forward_k_vs_all(x) → torch.FloatTensor

    forward_triples(x) → torch.FloatTensor

```

#### Parameters

**x**

#### Returns

```

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

    name = 'Pyke'

    dist_func

    margin = 1.0

    forward_triples(x: torch.LongTensor)

```

#### Parameters

**x**

## `dicee.models.static_funcs`

### Functions

---

```

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

---

## Module Contents

```

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

```

## `dicee.models.transformers`

Full definition of a GPT Language Model, all of it in this single file. References: 1) the official GPT-2 TensorFlow implementation released by OpenAI: <https://github.com/openai/gpt-2/blob/master/src/model.py> 2) huggingface/transformers PyTorch implementation: [https://github.com/huggingface/transformers/blob/main/src/transformers/models/gpt2/modeling\\_gpt2.py](https://github.com/huggingface/transformers/blob/main/src/transformers/models/gpt2/modeling_gpt2.py)

## Classes

<i>Byte</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>CausalSelfAttention</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.Byte(*args, **kwargs)
```

Bases: *dicee.models.base\_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### Note

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

### Variables

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

```
name = 'Byte'
```

```
config
```

```

temperature = 0.5

topk = 2

transformer

lm_head

loss_function(yhat_batch, y_batch)

```

#### Parameters

- **yhat\_batch**
- **y\_batch**

**forward** (*x: torch.LongTensor*)

#### Parameters

**x** (*B by T tensor*)

**generate** (*idx, max\_new\_tokens, temperature=1.0, top\_k=None*)

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

**training\_step** (*batch, batch\_idx=None*)

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

#### Parameters

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

#### Returns

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

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

Example:

```

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

```

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

```

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

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

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

```

#### **Note**

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

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

Bases: `torch.nn.Module`

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

**weight**

**bias**

**forward**(input)

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

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the 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**

**flash = True**

**forward**(*x*)

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

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### **Note**

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

### **Variables**

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

**c\_fc**

**gelu**

**c\_proj**

**dropout**

**forward**(*x*)

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

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### **Note**

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

### **Variables**

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

**ln\_1**

```

    attn

    ln_2

    mlp

    forward(x)

```

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

```

    block_size: int = 1024

    vocab_size: int = 50304

    n_layer: int = 12

    n_head: int = 12

    n_embd: int = 768

    dropout: float = 0.0

    bias: bool = False

```

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

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

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

```

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

#### Note

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

#### Variables

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

**config**

**transformer**

**lm\_head**

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

**forward** (*idx, targets=None*)

**crop\_block\_size** (*block\_size*)

**classmethod from\_pretrained** (*model\_type, override\_args=None*)

**configure\_optimizers** (*weight\_decay, learning\_rate, betas, device\_type*)

**estimate\_mfu** (*fwdbwd\_per\_iter, dt*)  
 estimate model flops utilization (MFU) in units of A100 bfloat16 peak FLOPS

## Classes

<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>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 ( <a href="https://arxiv.org/abs/2101.09090">https://arxiv.org/abs/2101.09090</a> )
<i>Pyke</i>	A Physical Embedding Model for Knowledge Graphs
<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.

continues on next page

Table 1 – continued from previous page

<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 ( <a href="https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657">https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657</a> )

## 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:

- $\theta_t$ : parameter at step  $t$
- $g_t$ : gradient at step  $t$
- $m_t$ : first moment estimate (momentum)
- $v_t$ : second moment estimate (variance)
- $\alpha$ : learning rate
- $\beta_1, \beta_2$ : exponential decay rates
- `clip()`: optional gradient clipping function

#### Reference:

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

#### Parameters

- **params** (*ParamST*) – Iterable of parameters to optimize or dicts defining parameter groups.
- **lr** (*float or Tensor, optional*) – Learning rate. Can be a float or 1-element Tensor. Default: `1e-3`
- **betas** (*Tuple[float, float], optional*) – Coefficients ( $\beta_1, \beta_2$ ) for computing running averages of gradient and its square.  $\beta_1$  controls momentum,  $\beta_2$  controls variance. Default: `(0.9, 0.9999)`
- **eps** (*float, optional*) – Term added to denominator to improve numerical stability. Default: `1e-6`
- **clip\_lambda** (*Callable[[int], float], optional*) – Function that takes the step number and returns the gradient clipping threshold. Common choices: - `lambda step: step**0.25` (default, gradually increases clipping threshold) - `lambda step: 1.0` (constant clipping) - `None` (no clipping) Default: `lambda step: step**0.25`
- **weight\_decay** (*float, optional*) – Weight decay coefficient (L2 penalty). Default: `0.0`
- **decouple** (*bool, optional*) – If True, uses decoupled weight decay (AdamW-style), applying weight decay directly to parameters. If False, adds weight decay to gradients (L2 regularization). Default: `False`
- **foreach** (*bool, optional*) – If True, uses the faster foreach implementation for multi-tensor operations. Default: `None` (auto-select)
- **maximize** (*bool, optional*) – If True, maximizes parameters instead of minimizing. Useful for reinforcement learning. Default: `False`
- **capturable** (*bool, optional*) – If True, the optimizer is safe to capture in a CUDA graph. Requires learning rate as Tensor. Default: `False`
- **differentiable** (*bool, optional*) – If True, the optimization step can be differentiated. Useful for meta-learning. Default: `False`
- **fused** (*bool, optional*) – If True, uses fused kernel implementation (currently not supported). Default: `None`

#### Raises

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

## Example

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

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

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

### Note

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

## `clip_lambda`

### `__setstate__(state)`

Restore optimizer state from a checkpoint.

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

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

#### Parameters

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

### Note

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

### `step(closure=None)`

Perform a single optimization step.

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

1. Optionally evaluates a closure to recompute the loss (useful for algorithms like LBFGS or when loss needs multiple evaluations)

2. For each parameter group: - Collects parameters with gradients and their associated state - Extracts hyperparameters (betas, learning rate, etc.) - Calls the functional adopt() API to perform the actual update
3. Returns the loss value if a closure was provided

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

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

#### Parameters

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

#### Returns

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

#### Return type

Optional[Tensor]

#### Example

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

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

#### Note

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

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

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)

(continued from previous page)

```
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'
```

(continues on next page)

(continued from previous page)

```
# is available when the scheduler is updated, thus stopping
# training if not found. If set to `False`, it will only produce a warning
"strict": True,
# If using the `LearningRateMonitor` callback to monitor the
# learning rate progress, this keyword can be used to specify
# a custom logged name
"name": None,
}
```

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

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

#### Note

Some things to know:

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

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(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

        •  $\mathbf{x}$ 

        •  $\mathbf{y\_idx}$ 

        •  $\mathbf{ordered\_bpe\_entities}$ 

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

        • ( $\mathbf{b} (x \text{ shape})$ )

        • 3

        •  $\mathbf{t}$ )

```

`get_bpe_head_and_relation_representation(x: torch.LongTensor)`  
 $\rightarrow$  Tuple[torch.FloatTensor, torch.FloatTensor]

#### Parameters

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

`get_embeddings()`  $\rightarrow$  Tuple[numpy.ndarray, numpy.ndarray]

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### Note

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

#### Variables

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

**args** = None

**\_\_call\_\_**(x)

**static forward**(x)

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

Bases: 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

        •  $\mathbf{x}$ 

        •  $\mathbf{y\_idx}$ 

        •  $\mathbf{ordered\_bpe\_entities}$ 

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**

- $(b(x \text{ shape}))$
- 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.DistMult(args)`

Bases: `dicee.models.base_model.BaseKGE`

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

`name = 'DistMult'`

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

**Parameters**

- `emb_h`
- `emb_r`
- `emb_E`

`forward_k_vs_all(x: torch.LongTensor)`

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

`score(h, r, t)`

`class dicee.models.TransE(args)`

Bases: `dicee.models.base_model.BaseKGE`

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

`name = 'TransE'`

`margin = 4`

`score(head_ent_emb, rel_ent_emb, tail_ent_emb)`

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

`class dicee.models.Shallom(args)`

Bases: `dicee.models.base_model.BaseKGE`

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

```

name = 'Shallom'

shallom

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

forward_k_vs_all(x) → torch.FloatTensor

forward_triples(x) → torch.FloatTensor

```

#### Parameters

**x**

#### Returns

```

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

```

#### Parameters

**x**

```

class dicee.models.BaseKGE(args: dict)

```

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the 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.models.ConEx(args)
    Bases: dicee.models.base_model.BaseKGE
    Convolutional ComplEx Knowledge Graph Embeddings
    name = 'ConEx'
    conv2d
    fc_num_input
    fc1
    norm_fc1
    bn_conv2d
    feature_map_dropout
    residual_convolution(C_1: Tuple[torch.Tensor, torch.Tensor],
        C_2: Tuple[torch.Tensor, torch.Tensor]) → torch.FloatTensor
        Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors
        that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds
        complex-valued embeddings :return:
    forward_k_vs_all(x: torch.Tensor) → torch.FloatTensor
    forward_triples(x: torch.Tensor) → torch.FloatTensor

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

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

```

**forward\_triples** (*x*: torch.Tensor) → torch.FloatTensor

#### Parameters

**x**

**forward\_k\_vs\_sample** (*x*: torch.Tensor, *target\_entity\_idx*: torch.Tensor)

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

Bases: *dicee.models.base\_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### Note

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

#### Variables

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

**name** = 'Complex'

**static score** (*head\_ent\_emb*: torch.FloatTensor, *rel\_ent\_emb*: torch.FloatTensor,  
*tail\_ent\_emb*: torch.FloatTensor)

**static k\_vs\_all\_score** (*emb\_h*: torch.FloatTensor, *emb\_r*: torch.FloatTensor,  
*emb\_E*: torch.FloatTensor)

#### Parameters

- **emb\_h**
- **emb\_r**
- **emb\_E**

**forward\_k\_vs\_all** (*x: torch.LongTensor*) → torch.FloatTensor

**forward\_k\_vs\_sample** (*x: torch.LongTensor, target\_entity\_idx: torch.LongTensor*)

dicee.models.**quaternion\_mul** (\*, *Q\_1, Q\_2*)  
→ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Perform quaternion multiplication :param Q\_1: :param Q\_2: :return:

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### Note

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

### Variables

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

**args**

**embedding\_dim** = None

**num\_entities** = None

**num\_relations** = None

**num\_tokens** = None

**learning\_rate** = None

**apply\_unit\_norm** = None

```

input_dropout_rate = None
hidden_dropout_rate = None
optimizer_name = None
feature_map_dropout_rate = None
kernel_size = None
num_of_output_channels = None
weight_decay = None
loss
selected_optimizer = None
normalizer_class = None
normalize_head_entity_embeddings
normalize_relation_embeddings
normalize_tail_entity_embeddings
hidden_normalizer
param_init
input_dp_ent_real
input_dp_rel_real
hidden_dropout
loss_history = []
byte_pair_encoding
max_length_subword_tokens
block_size
forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

    Parameters
         $\mathbf{x} (B \times 2 \times T)$ 
forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors

    Parameters
    -----
init_params_with_sanity_checking()

```

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

#### Parameters

- **x**
- **y\_idx**
- **ordered\_bpe\_entities**

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

#### Parameters

**x**

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

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

```
get_triple_representation (idx_hrt)
```

```
get_head_relation_representation (indexed_triple)
```

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

#### Parameters

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

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

#### Parameters

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

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

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)

(continued from previous page)

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

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

#### **Note**

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

#### **Variables**

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

`args = None`

`__call__(x)`

`static forward(x)`

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

`class dicee.models.QMult(args)`

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### **Note**

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

### Variables

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

**name** = 'QMult'

**explicit** = True

**quaternion\_multiplication\_followed\_by\_inner\_product** (*h, r, t*)

### Parameters

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

### Returns

Triple scores.

**static quaternion\_normalizer** (*x: torch.FloatTensor*) → torch.FloatTensor

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

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

### Parameters

**x** – The vector.

### Returns

The normalized vector.

**score** (*head\_ent\_emb: torch.FloatTensor, rel\_ent\_emb: torch.FloatTensor, tail\_ent\_emb: torch.FloatTensor*)

**k\_vs\_all\_score** (*bpe\_head\_ent\_emb, bpe\_rel\_ent\_emb, E*)

### Parameters

- **bpe\_head\_ent\_emb**
- **bpe\_rel\_ent\_emb**
- **E**

**forward\_k\_vs\_all** (*x*)

### Parameters

**x**

**forward\_k\_vs\_sample** (*x, target\_entity\_idx*)

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

```

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

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

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

```

**forward\_triples** (*indexed\_triple: torch.Tensor*) → torch.Tensor

### Parameters

**x**

**forward\_k\_vs\_all** (*x: torch.Tensor*)

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

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### Note

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

### Variables

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

**args**

**embedding\_dim** = None

**num\_entities** = None

**num\_relations** = None

**num\_tokens** = None

**learning\_rate** = None

```

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all(x: torch.LongTensor)

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

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

    Parameters
    -----

init_params_with_sanity_checking()

```

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

#### Parameters

- **x**
- **y\_idx**
- **ordered\_bpe\_entities**

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

#### Parameters

**x**

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

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

```
get_triple_representation (idx_hrt)
```

```
get_head_relation_representation (indexed_triple)
```

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

#### Parameters

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

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

#### Parameters

**x** (*B x 2 x T*)

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

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)

(continued from previous page)

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

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

### **Note**

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

### **Variables**

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

**args** = `None`

**\_\_call\_\_** (*x*)

**static forward** (*x*)

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

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

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### Note

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

### Variables

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

`name = 'OMult'`

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

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

`k_vs_all_score (bpe_head_ent_emb, bpe_rel_ent_emb, E)`

`forward_k_vs_all (x)`

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

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### Note

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

### Variables

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

**name** = 'ConvO'

**conv2d**

**fc\_num\_input**

**fc1**

**bn\_conv2d**

**norm\_fc1**

**feature\_map\_dropout**

**static octonion\_normalizer** (*emb\_rel\_e0, emb\_rel\_e1, emb\_rel\_e2, emb\_rel\_e3, emb\_rel\_e4, emb\_rel\_e5, emb\_rel\_e6, emb\_rel\_e7*)

**residual\_convolution** (*O\_1, O\_2*)

**forward\_triples** (*x: torch.Tensor*) → torch.Tensor

### Parameters

**x**

**forward\_k\_vs\_all** (*x: torch.Tensor*)

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

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

Bases: *dicee.models.base\_model.BaseKGE*

Additive Convolutional Octonion Knowledge Graph Embeddings

**name** = 'AConvO'

**conv2d**

**fc\_num\_input**

**fc1**

**bn\_conv2d**

**norm\_fc1**

**feature\_map\_dropout**

**static octonion\_normalizer** (*emb\_rel\_e0, emb\_rel\_e1, emb\_rel\_e2, emb\_rel\_e3, emb\_rel\_e4, emb\_rel\_e5, emb\_rel\_e6, emb\_rel\_e7*)

**residual\_convolution** (*O\_1, O\_2*)

**forward\_triples** (*x: torch.Tensor*) → torch.Tensor

### Parameters

**x**

**forward\_k\_vs\_all** (*x*: torch.Tensor)

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

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

Bases: *dicee.models.base\_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### Note

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

### Variables

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

**name** = 'Keci'

**p**

**q**

**r**

**requires\_grad\_for\_interactions** = True

**compute\_sigma\_pp** (*hp, rp*)

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

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

Compute  $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_{j \ r \ k} - h_{k \ r \ j}) e_j e_k$  captures the interactions between along q bases For instance, let q e\_1, e\_2, e\_3, we compute interactions between e\_1 e\_2, e\_1 e\_3, and e\_2 e\_3 This can be implemented with a nested two for loops

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

**compute\_sigma\_pq** (\*, hp, hq, rp, rq)

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

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

**for j in range(q):**

        sigma\_pq[:, :, i, j] = hp[:, :, i] \* rq[:, :, j] - hq[:, :, j] \* rp[:, :, i]

print(sigma\_pq.shape)

**apply\_coefficients** (hp, hq, rp, rq)

Multiplying a base vector with its scalar coefficient

**clifford\_multiplication** (h0, hp, hq, r0, rp, rq)

Compute our CL multiplication

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

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

$e_j$

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

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

(2)  $\sigma_p = \sum_{i=1}^p (h_0 \ r_{i \ r \ i} + h_{i \ r \ i} r_0) e_i$

(3)  $\sigma_q = \sum_{j=p+1}^{p+q} (h_0 \ r_{j \ r \ j} + h_{j \ r \ j} r_0) e_j$

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

**construct\_batch\_selected\_cl\_multivector** (*x: torch.FloatTensor, r: int, p: int, q: int*)  
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

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

### Parameter

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

#### returns

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

**forward\_k\_vs\_sample** (*x: torch.LongTensor, target\_entity\_idx: torch.LongTensor*) → torch.FloatTensor

### Parameter

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

target\_entity\_idx: torch.LongTensor with (n, k ) shape k denotes the selected number of examples.

#### rtype

torch.FloatTensor with (n, k) shape

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

**forward\_triples** (*x: torch.Tensor*) → torch.FloatTensor

### Parameter

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

**rtype**

torch.FloatTensor with (n) shape

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

Bases: *Keci*

Without learning dimension scaling

**name** = 'CKeci'

**requires\_grad\_for\_interactions** = False

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

Bases: *dicee.models.base\_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### Note

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

### Variables

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

**name** = 'DeCaL'

**entity\_embeddings**

**relation\_embeddings**

**p**

**q**

**r**

**re**

**forward\_triples** ( $x$ : *torch.Tensor*)  $\rightarrow$  *torch.FloatTensor*

### Parameter

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

**rtype**

*torch.FloatTensor* with (n) shape

**cl\_pqr** ( $a$ : *torch.tensor*)  $\rightarrow$  *torch.tensor*

Input: *tensor*(batch\_size, emb\_dim)  $\rightarrow$  output: *tensor* with 1+p+q+r components with size (batch\_size, emb\_dim/(1+p+q+r)) each.

1) takes a *tensor* of size (batch\_size, emb\_dim), split it into 1 + p + q + r components, hence 1+p+q+r must be a divisor of the emb\_dim. 2) Return a list of the 1+p+q+r components vectors, each are *tensors* of size (batch\_size, emb\_dim/(1+p+q+r))

**compute\_sigmas\_single** (*list\_h\_emb*, *list\_r\_emb*, *list\_t\_emb*)

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

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

and return:

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

**compute\_sigmas\_multivect** (*list\_h\_emb*, *list\_r\_emb*)

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

For same bases vectors interaction we have

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

For different base vector interactions, we have

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

**forward\_k\_vs\_all** (*x*: torch.Tensor) → torch.FloatTensor

Kvsall training

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

forward\_k\_vs\_with\_explicit and this functions are identical Parameter ——— *x*: torch.LongTensor with (n, ) shape :rtype: torch.FloatTensor with (n, **IE**) shape

**apply\_coefficients** (*h0, hp, hq, hk, r0, rp, rq, rk*)

Multiplying a base vector with its scalar coefficient

**construct\_cl\_multivector** (*x: torch.FloatTensor, re: int, p: int, q: int, r: int*)  
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

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

## Parameter

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

### returns

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

**compute\_sigma\_pp** (*hp, rp*)

Compute .. math:

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

$\sigma_{\{pp\}}$  captures the interactions between along *p* bases For instance, let *p* e<sub>1</sub>, e<sub>2</sub>, e<sub>3</sub>, we compute interactions between e<sub>1</sub> e<sub>2</sub>, e<sub>1</sub> e<sub>3</sub>, and e<sub>2</sub> e<sub>3</sub> 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<sub>1</sub>e<sub>1</sub>, e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>,

e<sub>2</sub>e<sub>1</sub>, e<sub>2</sub>e<sub>2</sub>, e<sub>2</sub>e<sub>3</sub>, e<sub>3</sub>e<sub>1</sub>, e<sub>3</sub>e<sub>2</sub>, e<sub>3</sub>e<sub>3</sub>

Then select the triangular matrix without diagonals: e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>, e<sub>2</sub>e<sub>3</sub>.

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

Compute

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

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

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

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

Yet, this computation would be quite inefficient. Instead, we compute interactions along all  $p$ , e.g.,  $e_1 e_1, e_1 e_2, e_1 e_3$ ,

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

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

**compute\_sigma\_rr** ( $hk, rk$ )

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

**compute\_sigma\_pq** ( $*, hp, hq, rp, rq$ )

Compute

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

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

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

print(sigma\_pq.shape)

**compute\_sigma\_pr** ( $*, hp, hk, rp, rk$ )

Compute

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

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

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

print(sigma\_pq.shape)

**compute\_sigma\_qr** ( $*, hq, hk, rq, rk$ )

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

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

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

print(sigma\_pq.shape)

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### Note

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

#### Variables

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

**args**

**embedding\_dim** = None

**num\_entities** = None

**num\_relations** = None

**num\_tokens** = None

**learning\_rate** = None

**apply\_unit\_norm** = None

**input\_dropout\_rate** = None

**hidden\_dropout\_rate** = None

**optimizer\_name** = None

**feature\_map\_dropout\_rate** = None

```

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

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

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

    Parameters
    -----

init_params_with_sanity_checking()

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

    Parameters
        •  $\mathbf{x}$ 
        •  $\mathbf{y\_idx}$ 
        •  $\mathbf{ordered\_bpe\_entities}$ 

```

**forward\_triples** (*x*: torch.LongTensor) → torch.Tensor

**Parameters**

**x**

**forward\_k\_vs\_all** (\*args, \*\*kwargs)

**forward\_k\_vs\_sample** (\*args, \*\*kwargs)

**get\_triple\_representation** (idx\_hrt)

**get\_head\_relation\_representation** (indexed\_triple)

**get\_sentence\_representation** (*x*: torch.LongTensor)

**Parameters**

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

**get\_bpe\_head\_and\_relation\_representation** (*x*: torch.LongTensor)  
→ Tuple[torch.FloatTensor, torch.FloatTensor]

**Parameters**

**x** (*B* × 2 × *T*)

**get\_embeddings** () → Tuple[numpy.ndarray, numpy.ndarray]

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

Bases: [dicee.models.base\\_model.BaseKGE](#)

A class for using knowledge graph embedding models implemented in Pykeen

Notes: Pykeen\_DistMult: C Pykeen\_ComplEx: Pykeen\_QuatE: Pykeen\_MuRE: Pykeen\_CP: Pykeen\_HolE: Pykeen\_HolE: Pykeen\_HolE: Pykeen\_TransD: Pykeen\_TransE: Pykeen\_TransF: Pykeen\_TransH: Pykeen\_TransR:

**model\_kwargs**

**name**

**model**

**loss\_history** = []

**args**

**entity\_embeddings** = None

**relation\_embeddings** = None

**forward\_k\_vs\_all** (*x*: torch.LongTensor)

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

# (1) Retrieve embeddings of heads and relations + apply Dropout & Normalization if given. h, r = self.get\_head\_relation\_representation(x) # (2) Reshape (1). if self.last\_dim > 0:

h = h.reshape(len(x), self.embedding\_dim, self.last\_dim) r = r.reshape(len(x), self.embedding\_dim, self.last\_dim)

# (3) Reshape all entities. if self.last\_dim > 0:

```

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

    else:
        t = self.entity_embeddings.weight

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

forward_triples (x: torch.LongTensor) → torch.FloatTensor
    # => Explicit version by this we can apply bn and dropout

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

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

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
    • x
    • y_idx
    • ordered_bpe_entities

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

    Parameters
    • x

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

get_sentence_representation (x: torch.LongTensor)

    Parameters
    • (b (x shape)
    • 3
    • t)

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

    Parameters
    • x (B x 2 x T)

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

class dicee.models.FMult (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    name = 'FMult'
    entity_embeddings
    relation_embeddings
    k

```

```

num_sample = 50

gamma

roots

weights

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

chain_func (weights, x: torch.FloatTensor)

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

```

#### Parameters

**x**

```

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

```

#### Parameters

**x**

```

class dicee.models.FMult2 (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    name = 'FMult2'
    n_layers = 3
    k
    n = 50
    score_func = 'compositional'
    discrete_points

```

```

entity_embeddings

relation_embeddings

build_func (Vec)

build_chain_funcs (list_Vec)

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

function (list_W, list_b)

trapezoid (list_W, list_b)

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

```

#### Parameters

**x**

```
class dicee.models.LFMult1 (args)
```

Bases: *[dicee.models.base\\_model.BaseKGE](#)*

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

```
name = 'LFMult1'
```

```
entity_embeddings
```

```
relation_embeddings
```

```
forward_triples (idx_triple)
```

#### Parameters

**x**

```
tri_score (h, r, t)
```

```
vtp_score (h, r, t)
```

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

Bases: *[dicee.models.base\\_model.BaseKGE](#)*

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

```
name = 'LFMult'
```

```
entity_embeddings
```

```
relation_embeddings
```

```
degree
```

```
m
```

```
x_values
```

**forward\_triples** (*idx\_triple*)

#### Parameters

**x**

**construct\_multi\_coeff** (*x*)

**poly\_NN** (*x, coefh, coefr, coeft*)

Constructing a 2 layers NN to represent the embeddings.  $h = \text{sigma}(w_h^T x + b_h)$ ,  $r = \text{sigma}(w_r^T x + b_r)$ ,  $t = \text{sigma}(w_t^T x + b_t)$

**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: str, val_path: str, test_path: str,
    ent2id: Dict = None, rel2id: Dict = None, seed: int = 1, gen_valid: bool = False,
    gen_test: bool = True)

    train_path

    val_path

    test_path

    gen_valid = False

    gen_test = True

    seed = 1

    max_ans_num = 1000000.0

    mode

    ent2id = None

    rel2id: Dict = None

    ent_in: Dict

    ent_out: Dict

    query_name_to_struct

    list2tuple(list_data)

    tuple2list(x: List | Tuple) → List | Tuple
        Convert a nested tuple to a nested list.

    set_global_seed(seed: int)
        Set seed

    construct_graph(paths: List[str]) → Tuple[Dict, Dict]
        Construct graph from triples Returns dicts with incoming and outgoing edges

    fill_query(query_structure: List[str | List], ent_in: Dict, ent_out: Dict, answer: int) → bool
        Private method for fill_query logic.

    achieve_answer(query: List[str | List], ent_in: Dict, ent_out: Dict) → set
        Private method for achieve_answer logic. @TODO: Document the code

    write_links(ent_out, small_ent_out)

    ground_queries(query_structure: List[str | List], ent_in: Dict, ent_out: Dict, small_ent_in: Dict,
        small_ent_out: Dict, gen_num: int, query_name: str)
        Generating queries and achieving answers

    unmap(query_type, queries, tp_answers, fp_answers, fn_answers)

    unmap_query(query_structure, query, id2ent, id2rel)
```

**generate\_queries** (*query\_struct: List, gen\_num: int, query\_type: str*)

Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting queries and answers in return @ TODO: create a class for each single query struct

**save\_queries** (*query\_type: str, gen\_num: int, save\_path: str*)

**abstract load\_queries** (*path*)

**get\_queries** (*query\_type: str, gen\_num: int*)

**static save\_queries\_and\_answers** (*path: str, data: List[Tuple[str, Tuple[collections.defaultdict]]]*)  
→ None

Save Queries into Disk

**static load\_queries\_and\_answers** (*path: str*) → List[Tuple[str, Tuple[collections.defaultdict]]]

Load Queries from Disk to Memory

## **dicee.read\_preprocess\_save\_load\_kg**

### **Submodules**

#### **dicee.read\_preprocess\_save\_load\_kg.preprocess**

### **Classes**

*PreprocessKG*

Preprocess the data in memory

### **Module Contents**

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

Preprocess the data in memory

**kg**

**start** () → None

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

#### **Parameter**

**rtype**

None

**preprocess\_with\_byte\_pair\_encoding** ()

**preprocess\_with\_byte\_pair\_encoding\_with\_padding** () → None

Preprocess with byte pair encoding and add padding

**preprocess\_with\_pandas** () → None

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

**preprocess\_with\_polars** () → None

Preprocess with polars: add reciprocal triples and create indexed datasets

`sequential_vocabulary_construction()` → None

- (1) Read input data into memory
- (2) Remove triples with a condition
- (3) **Serialize vocabularies in a pandas dataframe where**  
=> the index is integer and => a single column is string (e.g. URI)

`dicee.read_preprocess_save_load_kg.read_from_disk`

## Classes

---

*ReadFromDisk*

Read the data from disk into memory

---

## Module Contents

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

Read the data from disk into memory

**kg**

**start()** → None

Read a knowledge graph from disk into memory

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

### Parameter

None

**rtype**

None

**add\_noisy\_triples\_into\_training()**

`dicee.read_preprocess_save_load_kg.save_load_disk`

## Classes

---

*LoadSaveToDisk*

## Module Contents

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

**kg**

**save()**

**load()**

## dicee.read\_preprocess\_save\_load\_kg.util

### Functions

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

### Module Contents

`dicee.read_preprocess_save_load_kg.util.polars_dataframe_indexer` (  
    *df\_polars: polars.DataFrame, idx\_entity: polars.DataFrame, idx\_relation: polars.DataFrame*  
     $\rightarrow$  polars.DataFrame

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

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

### Parameters:

#### **df\_polars**

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

#### **idx\_entity**

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

#### **idx\_relation**

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

### Returns:

#### **polars.DataFrame**

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

### Example Usage:

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

### Steps:

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

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

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

## Parameters:

### **df\_pandas**

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

### **idx\_entity**

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

### **idx\_relation**

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

## Returns:

### **pd.DataFrame**

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

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

Add reciprocal triples if conditions are met

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

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

Load and Preprocess via Polars

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

Load and Preprocess via Pandas

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

```
dicee.read_preprocess_save_load_kg.util.count_triples (endpoint: str) → int
```

Returns the total number of triples in the triple store.

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

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

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

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

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

Read triples from triple store into pandas dataframe

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

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

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

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

(1) Extract domains and ranges of relations

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

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

Deserialize data

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

`dicee.read_preprocess_save_load_kg.util.load_numpy_ndarray (*, file_path: str)`

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

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

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

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

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

#### Parameters

- **train\_set**
- **num\_entities**
- **num\_relations**

#### Returns

## Classes

<code>PreprocessKG</code>	Preprocess the data in memory
<code>LoadSaveToDisk</code>	
<code>ReadFromDisk</code>	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> ([description])	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> (x)	Add inverse triples into dask 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>timeit</code> (func)	
<code>save_pickle</code> (*[, data, file_path])	
<code>load_pickle</code> ([file_path])	
<code>load_term_mapping</code> ([file_path])	
<code>select_model</code> (args[, is_continual_training, storage_path])	
<code>load_model</code> (→ Tuple[object, Tuple[dict, dict]])	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> (*[, data, file_path])	
<code>numpy_data_type_changer</code> (→ numpy.ndarray)	Detect most efficient data type for a given triples
<code>save_checkpoint_model</code> (→ None)	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(</code> <code>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</code> ( $\rightarrow$ Iterable)
--

### Module Contents

`dicee.trainer.torch_trainer_ddp.make_iterable_verbose` (*iterable\_object*, *verbose*,  
*desc*='Default', *position*=None, *leave*=True)  $\rightarrow$  Iterable

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

Bases: `dicee.abstracts.AbstractTrainer`

A Trainer based on torch.nn.parallel.DistributedDataParallel

Arguments

**entity\_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 ( <a href="https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657">https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657</a> )
<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 ( <a href="https://arxiv.org/abs/2101.09090">https://arxiv.org/abs/2101.09090</a> )
<i>LFMult</i>	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
<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
<i>CVDDataModule</i>	Create a Dataset for cross validation

continues on next page

Table 3 – continued from previous page

<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, ...)	
<i>deploy_head_entity_prediction</i> (pre_trained_kge, ...)	

continues on next page

Table 4 – continued from previous page

<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`

```
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 sub-modules as regular attributes:

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

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

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

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

### Note

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

### Variables

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

**name** = 'Keci'

**p**

**q**

**r**

**requires\_grad\_for\_interactions** = True

**compute\_sigma\_pp** (*hp, rp*)

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

$\sigma_{pp}$  captures the interactions between along p bases For instance, let p e<sub>1</sub>, e<sub>2</sub>, e<sub>3</sub>, we compute interactions between e<sub>1</sub> e<sub>2</sub>, e<sub>1</sub> e<sub>3</sub>, and e<sub>2</sub> e<sub>3</sub> 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<sub>1</sub>e<sub>1</sub>, e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>,

e<sub>2</sub>e<sub>1</sub>, e<sub>2</sub>e<sub>2</sub>, e<sub>2</sub>e<sub>3</sub>, e<sub>3</sub>e<sub>1</sub>, e<sub>3</sub>e<sub>2</sub>, e<sub>3</sub>e<sub>3</sub>

Then select the triangular matrix without diagonals: e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>, e<sub>2</sub>e<sub>3</sub>.

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

Compute  $\sigma_{qq} = \sum_{j=1}^{q-1} \sum_{k=j+1}^q (h_{jr_k} - h_{kr_j}) e_j e_k$  captures the interactions between along q bases For instance, let q e<sub>1</sub>, e<sub>2</sub>, e<sub>3</sub>, we compute interactions between e<sub>1</sub> e<sub>2</sub>, e<sub>1</sub> e<sub>3</sub>, and e<sub>2</sub> e<sub>3</sub> 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<sub>1</sub>e<sub>1</sub>, e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>,

e<sub>2</sub>e<sub>1</sub>, e<sub>2</sub>e<sub>2</sub>, e<sub>2</sub>e<sub>3</sub>, e<sub>3</sub>e<sub>1</sub>, e<sub>3</sub>e<sub>2</sub>, e<sub>3</sub>e<sub>3</sub>

Then select the triangular matrix without diagonals: e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>, e<sub>2</sub>e<sub>3</sub>.

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


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


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

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

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

For different base vector interactions, we have

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

**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:

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

$\sigma_{pp}$  captures the interactions between along  $p$  bases For instance, let  $p$  e\_1, e\_2, e\_3, we compute interactions between e\_1 e\_2, e\_1 e\_3, and e\_2 e\_3 This can be implemented with a nested two for loops

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

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

Compute

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

$\sigma_{qq}$  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, **!Entities!**) Given a batch of head entities and  
relations => shape (size of batch,! Entities!)

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

    A class for using knowledge graph embedding models implemented in Pykeen

    Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Py-
    keen_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.  $(h, r) \in KG$

#### 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  $N = |E| \times |R|$   $y_i$  denotes a multi-label vector in  $[0, 1]^{|E|}$  is a binary label.

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

#### Note

**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
  - `adopt`, 57
  - `base_model`, 66
  - `clifford`, 75
  - `complex`, 82
  - `dualE`, 84
  - `ensemble`, 85
  - `function_space`, 86
  - `literal`, 90
  - `octonion`, 91
  - `pykeen_models`, 94
  - `quaternion`, 96
  - `real`, 99
  - `static_funcs`, 100
  - `transformers`, 100
- `dicee.query_generator`, 157
- `dicee.read_preprocess_save_load_kg`, 159
  - `read_preprocess_save_load_kg.preprocess`, 159
  - `read_preprocess_save_load_kg.read_from_disk`, 160
  - `read_preprocess_save_load_kg.save_load_disk`, 160
  - `read_preprocess_save_load_kg.util`, 161
- `dicee.sanity_checkers`, 165
- `dicee.scripts`, 166
  - `index_serve`, 166
  - `run`, 168
- `dicee.static_funcs`, 168
- `dicee.static_funcs_training`, 171
- `dicee.static_preprocess_funcs`, 172
- `dicee.trainer`, 173
  - `dice_trainer`, 173
  - `model_parallelism`, 175
  - `torch_trainer`, 175
  - `torch_trainer_ddp`, 177

# Index

## Non-alphabetical

`__call__` () (*dicee.EnsembleKGE method*), 206  
`__call__` () (*dicee.models.base\_model.IdentityClass method*), 75  
`__call__` () (*dicee.models.ensemble.EnsembleKGE method*), 86  
`__call__` () (*dicee.models.IdentityClass method*), 120, 132, 138  
`__getitem__` () (*dicee.AllvsAll method*), 217  
`__getitem__` () (*dicee.BPE\_NegativeSamplingDataset method*), 213  
`__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*), 216  
`__getitem__` () (*dicee.KvsSampleDataset method*), 219  
`__getitem__` () (*dicee.LiteralDataset method*), 225  
`__getitem__` () (*dicee.MultiClassClassificationDataset method*), 215  
`__getitem__` () (*dicee.MultiLabelDataset method*), 214  
`__getitem__` () (*dicee.NegSampleDataset method*), 220  
`__getitem__` () (*dicee.OnevsAllDataset method*), 215  
`__getitem__` () (*dicee.OnevsSample method*), 218  
`__getitem__` () (*dicee.TriplePredictionDataset method*), 220  
`__iter__` () (*dicee.config.Namespace method*), 32  
`__iter__` () (*dicee.EnsembleKGE method*), 206  
`__iter__` () (*dicee.knowledge\_graph.KG method*), 53  
`__iter__` () (*dicee.models.ensemble.EnsembleKGE method*), 86  
`__len__` () (*dicee.AllvsAll method*), 217  
`__len__` () (*dicee.BPE\_NegativeSamplingDataset method*), 213  
`__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*), 206  
`__len__` () (*dicee.knowledge\_graph.KG method*), 53  
`__len__` () (*dicee.KvsAll method*), 216  
`__len__` () (*dicee.KvsSampleDataset method*), 219  
`__len__` () (*dicee.LiteralDataset method*), 225  
`__len__` () (*dicee.models.ensemble.EnsembleKGE method*), 86  
`__len__` () (*dicee.MultiClassClassificationDataset method*), 215  
`__len__` () (*dicee.MultiLabelDataset method*), 214  
`__len__` () (*dicee.NegSampleDataset method*), 219  
`__len__` () (*dicee.OnevsAllDataset method*), 215  
`__len__` () (*dicee.OnevsSample method*), 218  
`__len__` () (*dicee.TriplePredictionDataset method*), 220  
`__setstate__` () (*dicee.models.ADOPT method*), 110  
`__setstate__` () (*dicee.models.adapt.ADOPT method*), 60  
`__str__` () (*dicee.EnsembleKGE method*), 206  
`__str__` () (*dicee.KGE method*), 209  
`__str__` () (*dicee.knowledge\_graph\_embeddings.KGE method*), 53  
`__str__` () (*dicee.models.ensemble.EnsembleKGE method*), 86  
`__version__` (in module *dicee*), 227

## 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), 158  
achieve\_answer() (*dicee.QueryGenerator* method), 226  
AConEx (class in *dicee*), 192  
AConEx (class in *dicee.models*), 127  
AConEx (class in *dicee.models.complex*), 83  
AConvO (class in *dicee*), 192  
AConvO (class in *dicee.models*), 140  
AConvO (class in *dicee.models.octonion*), 94  
AConvQ (class in *dicee*), 193  
AConvQ (class in *dicee.models*), 134  
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*), 207  
add\_noisy\_triples() (in module *dicee.static\_funcs*), 170  
add\_noisy\_triples\_into\_training() (*dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk.ReadFromDisk* method), 160  
add\_noisy\_triples\_into\_training() (*dicee.read\_preprocess\_save\_load\_kg.ReadFromDisk* method), 165  
add\_reciprocal (*dicee.knowledge\_graph.KG* attribute), 52  
ADOPT (class in *dicee.models*), 108  
ADOPT (class in *dicee.models.adopt*), 58  
adopt() (in module *dicee.models.adopt*), 62  
AllvsAll (class in *dicee*), 216  
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), 211  
answer\_multi\_hop\_query() (*dicee.knowledge\_graph\_embeddings.KGE* method), 55  
app (in module *dicee.scripts.index\_serve*), 167  
apply\_coefficients() (*dicee.DeCaL* method), 188  
apply\_coefficients() (*dicee.Keci* method), 185  
apply\_coefficients() (*dicee.models.clifford.DeCaL* method), 80  
apply\_coefficients() (*dicee.models.clifford.Keci* method), 76  
apply\_coefficients() (*dicee.models.DeCaL* method), 146  
apply\_coefficients() (*dicee.models.Keci* method), 142  
apply\_reciprocal\_or\_noise() (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 163  
apply\_semantic\_constraint (*dicee.abstracts.BaseInteractiveKGE* attribute), 14  
apply\_unit\_norm (*dicee.BaseKGE* attribute), 204  
apply\_unit\_norm (*dicee.models.base\_model.BaseKGE* attribute), 72  
apply\_unit\_norm (*dicee.models.BaseKGE* attribute), 118, 121, 125, 129, 135, 148, 152  
args (*dicee.BaseKGE* attribute), 204  
args (*dicee.DICE\_Trainer* attribute), 208  
args (*dicee.EnsembleKGE* attribute), 206  
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), 118, 121, 125, 129, 135, 148, 151  
args (*dicee.models.ensemble.EnsembleKGE* attribute), 86  
args (*dicee.models.IdentityClass* attribute), 120, 132, 138  
args (*dicee.models.pykeen\_models.PykeenKGE* attribute), 95  
args (*dicee.models.PykeenKGE* attribute), 150  
args (*dicee.PykeenKGE* attribute), 200  
args (*dicee.trainer.DICE\_Trainer* attribute), 178  
args (*dicee.trainer.dice\_trainer.DICE\_Trainer* attribute), 173  
ASWA (class in *dicee.callbacks*), 24  
aswa (*dicee.analyse\_experiments.Experiment* attribute), 19  
attn (*dicee.models.transformers.Block* attribute), 105  
attn\_dropout (*dicee.models.transformers.CausalSelfAttention* attribute), 104  
attributes (*dicee.abstracts.AbstractTrainer* attribute), 13

auto\_batch\_finding (*dicее.config.Namespace attribute*), 32

## B

backend (*dicее.config.Namespace attribute*), 30

backend (*dicее.knowledge\_graph.KG attribute*), 52

BaseInteractiveKGE (*class in dicее.abstracts*), 14

BaseInteractiveTrainKGE (*class in dicее.abstracts*), 18

BaseKGE (*class in dicее*), 203

BaseKGE (*class in dicее.models*), 117, 120, 124, 129, 135, 147, 151

BaseKGE (*class in dicее.models.base\_model*), 71

BaseKGELightning (*class in dicее.models*), 111

BaseKGELightning (*class in dicее.models.base\_model*), 66

batch\_kronecker\_product () (*dicее.callbacks.KronE static method*), 26

batch\_size (*dicее.analyse\_experiments.Experiment attribute*), 19

batch\_size (*dicее.callbacks.PseudoLabellingCallback attribute*), 24

batch\_size (*dicее.config.Namespace attribute*), 30

batch\_size (*dicее.CVDataModule attribute*), 221

batch\_size (*dicее.dataset\_classes.CVDataModule attribute*), 41

batches\_per\_epoch (*dicее.callbacks.LRScheduler attribute*), 28

bias (*dicее.models.transformers.GPTConfig attribute*), 106

bias (*dicее.models.transformers.LayerNorm attribute*), 103

Block (*class in dicее.models.transformers*), 105

block\_size (*dicее.BaseKGE attribute*), 205

block\_size (*dicее.config.Namespace attribute*), 32

block\_size (*dicее.dataset\_classes.MultiClassClassificationDataset attribute*), 35

block\_size (*dicее.models.base\_model.BaseKGE attribute*), 73

block\_size (*dicее.models.BaseKGE attribute*), 119, 122, 126, 130, 136, 149, 152

block\_size (*dicее.models.transformers.GPTConfig attribute*), 106

block\_size (*dicее.MultiClassClassificationDataset attribute*), 214

bn\_conv1 (*dicее.AConvQ attribute*), 193

bn\_conv1 (*dicее.ConvQ attribute*), 194

bn\_conv1 (*dicее.models.AConvQ attribute*), 134

bn\_conv1 (*dicее.models.ConvQ attribute*), 134

bn\_conv1 (*dicее.models.quaternion.AConvQ attribute*), 98

bn\_conv1 (*dicее.models.quaternion.ConvQ attribute*), 98

bn\_conv2 (*dicее.AConvQ attribute*), 193

bn\_conv2 (*dicее.ConvQ attribute*), 194

bn\_conv2 (*dicее.models.AConvQ attribute*), 134

bn\_conv2 (*dicее.models.ConvQ attribute*), 134

bn\_conv2 (*dicее.models.quaternion.AConvQ attribute*), 98

bn\_conv2 (*dicее.models.quaternion.ConvQ attribute*), 98

bn\_conv2d (*dicее.AConEx attribute*), 192

bn\_conv2d (*dicее.AConvO attribute*), 193

bn\_conv2d (*dicее.ConEx attribute*), 196

bn\_conv2d (*dicее.ConvO attribute*), 195

bn\_conv2d (*dicее.models.AConEx attribute*), 127

bn\_conv2d (*dicее.models.AConvO attribute*), 140

bn\_conv2d (*dicее.models.complex.AConEx attribute*), 83

bn\_conv2d (*dicее.models.complex.ConEx attribute*), 82

bn\_conv2d (*dicее.models.ConEx attribute*), 127

bn\_conv2d (*dicее.models.ConvO attribute*), 140

bn\_conv2d (*dicее.models.octonion.AConvO attribute*), 94

bn\_conv2d (*dicее.models.octonion.ConvO attribute*), 93

BPE\_NegativeSamplingDataset (*class in dicее*), 213

BPE\_NegativeSamplingDataset (*class in dicее.dataset\_classes*), 33

build\_chain\_funcs () (*dicее.models.FMult2 method*), 155

build\_chain\_funcs () (*dicее.models.function\_space.FMult2 method*), 88

build\_func () (*dicее.models.FMult2 method*), 155

build\_func () (*dicее.models.function\_space.FMult2 method*), 88

Byte (*class in dicее*), 201

Byte (*class in dicее.models.transformers*), 101

byte\_pair\_encoding (*dicее.analyse\_experiments.Experiment attribute*), 19

byte\_pair\_encoding (*dicее.BaseKGE attribute*), 204

byte\_pair\_encoding (*dicее.config.Namespace attribute*), 31

byte\_pair\_encoding (*dicее.knowledge\_graph.KG attribute*), 52

byte\_pair\_encoding (*dicее.models.base\_model.BaseKGE attribute*), 73

byte\_pair\_encoding (*dicее.models.BaseKGE attribute*), 119, 122, 125, 130, 136, 149, 152

## C

`c_attn` (*dicee.models.transformers.CausalSelfAttention* attribute), 104  
`c_fc` (*dicee.models.transformers.MLP* attribute), 105  
`c_proj` (*dicee.models.transformers.CausalSelfAttention* attribute), 104  
`c_proj` (*dicee.models.transformers.MLP* 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), 178  
`CausalSelfAttention` (class in *dicee.models.transformers*), 103  
`chain_func()` (*dicee.models.FMult* method), 154  
`chain_func()` (*dicee.models.function\_space.FMult* method), 87  
`chain_func()` (*dicee.models.function\_space.GFMult* method), 87  
`chain_func()` (*dicee.models.GFMult* method), 154  
`CKeci` (class in *dicee*), 183  
`CKeci` (class in *dicee.models*), 144  
`CKeci` (class in *dicee.models.clifford*), 78  
`cl_pqr()` (*dicee.DeCaL* method), 187  
`cl_pqr()` (*dicee.models.clifford.DeCaL* method), 79  
`cl_pqr()` (*dicee.models.DeCaL* method), 145  
`cleanup()` (*dicee.executer.Execute* method), 50  
`clifford_multiplication()` (*dicee.Keci* method), 185  
`clifford_multiplication()` (*dicee.models.clifford.Keci* method), 76  
`clifford_multiplication()` (*dicee.models.Keci* method), 142  
`clip_lambda` (*dicee.models.ADOPT* attribute), 110  
`clip_lambda` (*dicee.models.adopt.ADOPT* attribute), 60  
`collate_fn` (*dicee.AllvsAll* attribute), 216  
`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), 216  
`collate_fn` (*dicee.KvsSampleDataset* attribute), 219  
`collate_fn` (*dicee.MultiClassClassificationDataset* attribute), 214  
`collate_fn` (*dicee.MultiLabelDataset* attribute), 214  
`collate_fn` (*dicee.OnevsAllDataset* attribute), 215  
`collate_fn` (*dicee.OnevsSample* attribute), 217, 218  
`collate_fn()` (*dicee.BPE\_NegativeSamplingDataset* method), 213  
`collate_fn()` (*dicee.dataset\_classes.BPE\_NegativeSamplingDataset* method), 34  
`collate_fn()` (*dicee.dataset\_classes.TriplePredictionDataset* method), 41  
`collate_fn()` (*dicee.TriplePredictionDataset* method), 220  
`collection_name` (*dicee.scripts.index\_serve.NeuralSearcher* attribute), 167  
`comp_func()` (*dicee.LFMult* method), 200  
`comp_func()` (*dicee.models.function\_space.LFMult* method), 89  
`comp_func()` (*dicee.models.LFMult* method), 156  
`Complex` (class in *dicee*), 191  
`Complex` (class in *dicee.models*), 128  
`Complex` (class in *dicee.models.complex*), 83  
`compute_convergence()` (in module *dicee.callbacks*), 24  
`compute_func()` (*dicee.models.FMult* method), 154  
`compute_func()` (*dicee.models.FMult2* method), 155  
`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), 154  
`compute_mrr()` (*dicee.callbacks.ASWA* static method), 25  
`compute_sigma_pp()` (*dicee.DeCaL* method), 188  
`compute_sigma_pp()` (*dicee.Keci* method), 184  
`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), 146  
`compute_sigma_pp()` (*dicee.models.Keci* method), 141  
`compute_sigma_pq()` (*dicee.DeCaL* method), 189  
`compute_sigma_pq()` (*dicee.Keci* method), 184  
`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*), 147  
`compute_sigma_pq()` (*dicee.models.Keci method*), 142  
`compute_sigma_pr()` (*dicee.DeCaL method*), 190  
`compute_sigma_pr()` (*dicee.models.clifford.DeCaL method*), 81  
`compute_sigma_pr()` (*dicee.models.DeCaL method*), 147  
`compute_sigma_qq()` (*dicee.DeCaL method*), 189  
`compute_sigma_qq()` (*dicee.Keci method*), 184  
`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*), 146  
`compute_sigma_qq()` (*dicee.models.Keci method*), 142  
`compute_sigma_qr()` (*dicee.DeCaL method*), 190  
`compute_sigma_qr()` (*dicee.models.clifford.DeCaL method*), 82  
`compute_sigma_qr()` (*dicee.models.DeCaL method*), 147  
`compute_sigma_rr()` (*dicee.DeCaL method*), 189  
`compute_sigma_rr()` (*dicee.models.clifford.DeCaL method*), 81  
`compute_sigma_rr()` (*dicee.models.DeCaL method*), 147  
`compute_sigmas_multivect()` (*dicee.DeCaL method*), 188  
`compute_sigmas_multivect()` (*dicee.models.clifford.DeCaL method*), 79  
`compute_sigmas_multivect()` (*dicee.models.DeCaL method*), 145  
`compute_sigmas_single()` (*dicee.DeCaL method*), 187  
`compute_sigmas_single()` (*dicee.models.clifford.DeCaL method*), 79  
`compute_sigmas_single()` (*dicee.models.DeCaL method*), 145  
`ConEx` (*class in dicee*), 195  
`ConEx` (*class in dicee.models*), 126  
`ConEx` (*class in dicee.models.complex*), 82  
`config` (*dicee.BytE attribute*), 202  
`config` (*dicee.models.transformers.BytE attribute*), 101  
`config` (*dicee.models.transformers.GPT attribute*), 107  
`configs` (*dicee.abstracts.BaseInteractiveKGE attribute*), 14  
`configure_optimizers()` (*dicee.models.base\_model.BaseKGELightning method*), 70  
`configure_optimizers()` (*dicee.models.BaseKGELightning method*), 116  
`configure_optimizers()` (*dicee.models.transformers.GPT method*), 107  
`construct_batch_selected_cl_multivector()` (*dicee.Keci method*), 185  
`construct_batch_selected_cl_multivector()` (*dicee.models.clifford.Keci method*), 77  
`construct_batch_selected_cl_multivector()` (*dicee.models.Keci method*), 143  
`construct_cl_multivector()` (*dicee.DeCaL method*), 188  
`construct_cl_multivector()` (*dicee.Keci method*), 185  
`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*), 146  
`construct_cl_multivector()` (*dicee.models.Keci method*), 143  
`construct_dataset()` (*in module dicee*), 213  
`construct_dataset()` (*in module dicee.dataset\_classes*), 33  
`construct_ensemble` (*dicee.abstracts.BaseInteractiveKGE attribute*), 14  
`construct_graph()` (*dicee.query\_generator.QueryGenerator method*), 158  
`construct_graph()` (*dicee.QueryGenerator method*), 226  
`construct_input_and_output()` (*dicee.abstracts.BaseInteractiveKGE method*), 16  
`construct_multi_coeff()` (*dicee.LFMMult method*), 199  
`construct_multi_coeff()` (*dicee.models.function\_space.LFMMult method*), 89  
`construct_multi_coeff()` (*dicee.models.LFMMult method*), 156  
`continual_learning` (*dicee.config.Namespace attribute*), 32  
`continual_start()` (*dicee.DICE\_Trainer method*), 208  
`continual_start()` (*dicee.executer.ContinuousExecute method*), 51  
`continual_start()` (*dicee.trainer.DICE\_Trainer method*), 178  
`continual_start()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 174  
`continual_training_setup_executor()` (*in module dicee*), 207  
`continual_training_setup_executor()` (*in module dicee.static\_funcs*), 171  
`ContinuousExecute` (*class in dicee.executer*), 51  
`conv2d` (*dicee.AConEx attribute*), 192  
`conv2d` (*dicee.AConvO attribute*), 193  
`conv2d` (*dicee.AConvQ attribute*), 193  
`conv2d` (*dicee.ConEx attribute*), 195  
`conv2d` (*dicee.ConvO attribute*), 195  
`conv2d` (*dicee.ConvQ attribute*), 194  
`conv2d` (*dicee.models.AConEx attribute*), 127  
`conv2d` (*dicee.models.AConvO attribute*), 140

`conv2d` (*dicee.models.AConvQ attribute*), 134  
`conv2d` (*dicee.models.complex.AConEx attribute*), 83  
`conv2d` (*dicee.models.complex.ConEx attribute*), 82  
`conv2d` (*dicee.models.ConEx attribute*), 127  
`conv2d` (*dicee.models.ConvO attribute*), 140  
`conv2d` (*dicee.models.ConvQ attribute*), 134  
`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*), 194  
`ConvO` (*class in dicee.models*), 139  
`ConvO` (*class in dicee.models.octonion*), 93  
`ConvQ` (*class in dicee*), 194  
`ConvQ` (*class in dicee.models*), 133  
`ConvQ` (*class in dicee.models.quaternion*), 97  
`count_triples()` (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 163  
`create_and_store_kg()` (*dicee.executer.Execute method*), 50  
`create_constraints()` (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 163  
`create_constraints()` (*in module dicee.static\_preprocess\_funcs*), 172  
`create_experiment_folder()` (*in module dicee*), 207  
`create_experiment_folder()` (*in module dicee.static\_funcs*), 171  
`create_random_data()` (*dicee.callbacks.PseudoLabellingCallback method*), 24  
`create_recipriocal_triples()` (*in module dicee*), 206  
`create_recipriocal_triples()` (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 164  
`create_recipriocal_triples()` (*in module dicee.static\_funcs*), 169  
`create_vector_database()` (*dicee.KGE method*), 209  
`create_vector_database()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 53  
`crop_block_size()` (*dicee.models.transformers.GPT method*), 107  
`ctx` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 178  
`current_epoch` (*dicee.callbacks.SWA attribute*), 29  
`CVDDataModule` (*class in dicee*), 220  
`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*), 225  
`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*), 164  
`DeCaL` (*class in dicee*), 186  
`DeCaL` (*class in dicee.models*), 144  
`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*), 199  
`degree` (*dicee.models.function\_space.LFMult attribute*), 89  
`degree` (*dicee.models.LFMult attribute*), 155  
`denormalize()` (*dicee.dataset\_classes.LiteralDataset static method*), 46  
`denormalize()` (*dicee.LiteralDataset static method*), 225  
`deploy()` (*dicee.KGE method*), 212  
`deploy()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 56  
`deploy_head_entity_prediction()` (*in module dicee*), 207  
`deploy_head_entity_prediction()` (*in module dicee.static\_funcs*), 170  
`deploy_relation_prediction()` (*in module dicee*), 207  
`deploy_relation_prediction()` (*in module dicee.static\_funcs*), 171  
`deploy_tail_entity_prediction()` (*in module dicee*), 207  
`deploy_tail_entity_prediction()` (*in module dicee.static\_funcs*), 170  
`deploy_triple_prediction()` (*in module dicee*), 207  
`deploy_triple_prediction()` (*in module dicee.static\_funcs*), 170  
`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*), 208

- DICE\_Trainer (*class in dicee.trainer*), 178
- DICE\_Trainer (*class in dicee.trainer.dice\_trainer*), 173
- 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, 100
- dicee.models.transformers
  - module, 100
- dicee.query\_generator
  - module, 157
- dicee.read\_preprocess\_save\_load\_kg
  - module, 159
- dicee.read\_preprocess\_save\_load\_kg.preprocess
  - module, 159
- dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk
  - module, 160
- dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk
  - module, 160
- dicee.read\_preprocess\_save\_load\_kg.util
  - module, 161

- `dicee.sanity_checkers`
  - module, 165
- `dicee.scripts`
  - module, 166
- `dicee.scripts.index_serve`
  - module, 166
- `dicee.scripts.run`
  - module, 168
- `dicee.static_funcs`
  - module, 168
- `dicee.static_funcs_training`
  - module, 171
- `dicee.static_preprocess_funcs`
  - module, 172
- `dicee.trainer`
  - module, 173
- `dicee.trainer.dice_trainer`
  - module, 173
- `dicee.trainer.model_parallelism`
  - module, 175
- `dicee.trainer.torch_trainer`
  - module, 175
- `dicee.trainer.torch_trainer_ddp`
  - module, 177
- `discrete_points` (*dicee.models.FMult2 attribute*), 154
- `discrete_points` (*dicee.models.function\_space.FMult2 attribute*), 88
- `dist_func` (*dicee.models.Pyke attribute*), 124
- `dist_func` (*dicee.models.real.Pyke attribute*), 100
- `dist_func` (*dicee.Pyke attribute*), 182
- `DistMult` (*class in dicee*), 182
- `DistMult` (*class in dicee.models*), 123
- `DistMult` (*class in dicee.models.real*), 99
- `distributed` (*dicee.executer.Execute attribute*), 50
- `download_file()` (*in module dicee*), 207
- `download_file()` (*in module dicee.static\_funcs*), 171
- `download_files_from_url()` (*in module dicee*), 207
- `download_files_from_url()` (*in module dicee.static\_funcs*), 171
- `download_pretrained_model()` (*in module dicee*), 208
- `download_pretrained_model()` (*in module dicee.static\_funcs*), 171
- `dropout` (*dicee.models.literal.LiteralEmbeddings attribute*), 90, 91
- `dropout` (*dicee.models.transformers.CausalSelfAttention attribute*), 104
- `dropout` (*dicee.models.transformers.GPTConfig attribute*), 106
- `dropout` (*dicee.models.transformers.MLP attribute*), 105
- `DualE` (*class in dicee*), 190
- `DualE` (*class in dicee.models*), 156
- `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*), 172
- `embedding_dim` (*dicee.analyse\_experiments.Experiment attribute*), 19
- `embedding_dim` (*dicee.BaseKGE attribute*), 204
- `embedding_dim` (*dicee.config.Namespace attribute*), 30
- `embedding_dim` (*dicee.models.base\_model.BaseKGE attribute*), 72
- `embedding_dim` (*dicee.models.BaseKGE attribute*), 118, 121, 125, 129, 135, 148, 152
- `embedding_dim` (*dicee.models.literal.LiteralEmbeddings attribute*), 91
- `embedding_dims` (*dicee.models.literal.LiteralEmbeddings attribute*), 90
- `enable_log` (*in module dicee.static\_preprocess\_funcs*), 172
- `enc` (*dicee.knowledge\_graph.KG attribute*), 52
- `end()` (*dicee.executer.Execute method*), 50
- `EnsembleKGE` (*class in dicee*), 205
- `EnsembleKGE` (*class in dicee.models.ensemble*), 86
- `ent2id` (*dicee.query\_generator.QueryGenerator attribute*), 158
- `ent2id` (*dicee.QueryGenerator attribute*), 226

ent\_in (*dicee.query\_generator.QueryGenerator* attribute), 158  
ent\_in (*dicee.QueryGenerator* attribute), 226  
ent\_out (*dicee.query\_generator.QueryGenerator* attribute), 158  
ent\_out (*dicee.QueryGenerator* attribute), 226  
entities\_str (*dicee.knowledge\_graph.KG* property), 53  
entity\_embeddings (*dicee.AConvQ* attribute), 193  
entity\_embeddings (*dicee.ConvQ* attribute), 194  
entity\_embeddings (*dicee.DeCaL* attribute), 187  
entity\_embeddings (*dicee.DualE* attribute), 190  
entity\_embeddings (*dicee.LFMult* attribute), 199  
entity\_embeddings (*dicee.models.AConvQ* attribute), 134  
entity\_embeddings (*dicee.models.clifford.DeCaL* attribute), 79  
entity\_embeddings (*dicee.models.ConvQ* attribute), 134  
entity\_embeddings (*dicee.models.DeCaL* attribute), 144  
entity\_embeddings (*dicee.models.DualE* attribute), 157  
entity\_embeddings (*dicee.models.dualE.DualE* attribute), 84  
entity\_embeddings (*dicee.models.FMult* attribute), 153  
entity\_embeddings (*dicee.models.FMult2* attribute), 154  
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), 154  
entity\_embeddings (*dicee.models.LFMult* attribute), 155  
entity\_embeddings (*dicee.models.LFMult1* attribute), 155  
entity\_embeddings (*dicee.models.literal.LiteralEmbeddings* attribute), 90, 91  
entity\_embeddings (*dicee.models.pykeen\_models.PykeenKGE* attribute), 95  
entity\_embeddings (*dicee.models.PykeenKGE* attribute), 150  
entity\_embeddings (*dicee.models.quaternion.AConvQ* attribute), 98  
entity\_embeddings (*dicee.models.quaternion.ConvQ* attribute), 98  
entity\_embeddings (*dicee.PykeenKGE* attribute), 200  
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), 224, 225  
entity\_to\_idx (*dicee.scripts.index\_serve.NeuralSearcher* attribute), 167  
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), 107  
estimate\_q () (in module *dicee.callbacks*), 24  
Eval (class in *dicee.callbacks*), 25  
eval () (*dicee.EnsembleKGE* method), 206  
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), 209  
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*), 207  
evaluate () (in module *dicee.static\_funcs*), 171  
evaluate\_bpe\_lp () (in module *dicee.static\_funcs\_training*), 172  
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*), 171  
 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), 208  
 evaluator (*dicee.executer.Execute* attribute), 50  
 evaluator (*dicee.trainer.DICE\_Trainer* attribute), 178  
 evaluator (*dicee.trainer.dice\_trainer.DICE\_Trainer* attribute), 174  
 every\_x\_epoch (*dicee.callbacks.KGESaveCallback* attribute), 23  
 example\_input\_array (*dicee.EnsembleKGE* property), 206  
 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), 133  
 explicit (*dicee.models.quaternion.QMult* attribute), 97  
 explicit (*dicee.QMult* attribute), 197  
 exponential\_function() (in module *dicee*), 207  
 exponential\_function() (in module *dicee.static\_funcs*), 171  
 extract\_input\_outputs() (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* method), 178  
 extract\_input\_outputs() (in module *dicee.trainer.model\_parallelism*), 175  
 extract\_input\_outputs\_set\_device() (*dicee.trainer.torch\_trainer.TorchTrainer* method), 176

## F

f (*dicee.callbacks.KronE* attribute), 26  
 fc (*dicee.models.literal.LiteralEmbeddings* attribute), 91  
 fc1 (*dicee.AConEx* attribute), 192  
 fc1 (*dicee.AConvO* attribute), 193  
 fc1 (*dicee.AConvQ* attribute), 193  
 fc1 (*dicee.ConEx* attribute), 195  
 fc1 (*dicee.ConvO* attribute), 195  
 fc1 (*dicee.ConvQ* attribute), 194  
 fc1 (*dicee.models.AConEx* attribute), 127  
 fc1 (*dicee.models.AConvO* attribute), 140  
 fc1 (*dicee.models.AConvQ* attribute), 134  
 fc1 (*dicee.models.complex.AConEx* attribute), 83  
 fc1 (*dicee.models.complex.ConEx* attribute), 82  
 fc1 (*dicee.models.ConEx* attribute), 127  
 fc1 (*dicee.models.ConvO* attribute), 140  
 fc1 (*dicee.models.ConvQ* attribute), 134  
 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), 192  
 fc\_num\_input (*dicee.AConvO* attribute), 193  
 fc\_num\_input (*dicee.AConvQ* attribute), 193  
 fc\_num\_input (*dicee.ConEx* attribute), 195  
 fc\_num\_input (*dicee.ConvO* attribute), 195  
 fc\_num\_input (*dicee.ConvQ* attribute), 194  
 fc\_num\_input (*dicee.models.AConEx* attribute), 127  
 fc\_num\_input (*dicee.models.AConvO* attribute), 140  
 fc\_num\_input (*dicee.models.AConvQ* attribute), 134  
 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), 127  
 fc\_num\_input (*dicee.models.ConvO* attribute), 140  
 fc\_num\_input (*dicee.models.ConvQ* attribute), 134  
 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), 192  
`feature_map_dropout` (*dicee.AConvO* attribute), 193  
`feature_map_dropout` (*dicee.AConvQ* attribute), 193  
`feature_map_dropout` (*dicee.ConEx* attribute), 196  
`feature_map_dropout` (*dicee.ConvO* attribute), 195  
`feature_map_dropout` (*dicee.ConvQ* attribute), 194  
`feature_map_dropout` (*dicee.models.AConEx* attribute), 127  
`feature_map_dropout` (*dicee.models.AConvO* attribute), 140  
`feature_map_dropout` (*dicee.models.AConvQ* attribute), 134  
`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), 127  
`feature_map_dropout` (*dicee.models.ConvO* attribute), 140  
`feature_map_dropout` (*dicee.models.ConvQ* attribute), 134  
`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), 204  
`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), 118, 121, 125, 130, 136, 148, 152  
`fetch_worker` () (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 163  
`fill_query` () (*dicee.query\_generator.QueryGenerator* method), 158  
`fill_query` () (*dicee.QueryGenerator* method), 226  
`find_good_batch_size` () (in module *dicee.trainer.model\_parallelism*), 175  
`find_missing_triples` () (*dicee.KGE* method), 212  
`find_missing_triples` () (*dicee.knowledge\_graph\_embeddings.KGE* method), 56  
`fit` () (*dicee.trainer.model\_parallelism.TensorParallel* method), 175  
`fit` () (*dicee.trainer.torch\_trainer\_ddp.TorchDDPTrainer* method), 177  
`fit` () (*dicee.trainer.torch\_trainer.TorchTrainer* method), 176  
`flash` (*dicee.models.transformers.CausalSelfAttention* attribute), 104  
`FMult` (class in *dicee.models*), 153  
`FMult` (class in *dicee.models.function\_space*), 87  
`FMult2` (class in *dicee.models*), 154  
`FMult2` (class in *dicee.models.function\_space*), 87  
`form_of_labelling` (*dicee.DICE\_Trainer* attribute), 208  
`form_of_labelling` (*dicee.trainer.DICE\_Trainer* attribute), 178  
`form_of_labelling` (*dicee.trainer.dice\_trainer.DICE\_Trainer* attribute), 174  
`forward` () (*dicee.BaseKGE* method), 205  
`forward` () (*dicee.BytE* method), 202  
`forward` () (*dicee.models.base\_model.BaseKGE* method), 73  
`forward` () (*dicee.models.base\_model.IdentityClass* static method), 75  
`forward` () (*dicee.models.BaseKGE* method), 119, 122, 126, 130, 136, 149, 153  
`forward` () (*dicee.models.IdentityClass* static method), 120, 132, 138  
`forward` () (*dicee.models.literal.LiteralEmbeddings* method), 91  
`forward` () (*dicee.models.transformers.Block* method), 106  
`forward` () (*dicee.models.transformers.BytE* method), 102  
`forward` () (*dicee.models.transformers.CausalSelfAttention* method), 104  
`forward` () (*dicee.models.transformers.GPT* method), 107  
`forward` () (*dicee.models.transformers.LayerNorm* method), 103  
`forward` () (*dicee.models.transformers.MLP* method), 105  
`forward_backward_update` () (*dicee.trainer.torch\_trainer.TorchTrainer* method), 176  
`forward_backward_update_loss` () (in module *dicee.trainer.model\_parallelism*), 175  
`forward_byte_pair_encoded_k_vs_all` () (*dicee.BaseKGE* method), 205  
`forward_byte_pair_encoded_k_vs_all` () (*dicee.models.base\_model.BaseKGE* method), 73  
`forward_byte_pair_encoded_k_vs_all` () (*dicee.models.BaseKGE* method), 119, 122, 126, 130, 136, 149, 152  
`forward_byte_pair_encoded_triple` () (*dicee.BaseKGE* method), 205  
`forward_byte_pair_encoded_triple` () (*dicee.models.base\_model.BaseKGE* method), 73  
`forward_byte_pair_encoded_triple` () (*dicee.models.BaseKGE* method), 119, 122, 126, 130, 136, 149, 152  
`forward_k_vs_all` () (*dicee.AConEx* method), 192  
`forward_k_vs_all` () (*dicee.AConvO* method), 193  
`forward_k_vs_all` () (*dicee.AConvQ* method), 193  
`forward_k_vs_all` () (*dicee.BaseKGE* method), 205

`forward_k_vs_all()` (*dicee.ComplEx method*), 192  
`forward_k_vs_all()` (*dicee.ConEx method*), 196  
`forward_k_vs_all()` (*dicee.ConvO method*), 195  
`forward_k_vs_all()` (*dicee.ConvQ method*), 194  
`forward_k_vs_all()` (*dicee.DeCaL method*), 188  
`forward_k_vs_all()` (*dicee.DistMult method*), 183  
`forward_k_vs_all()` (*dicee.DualE method*), 191  
`forward_k_vs_all()` (*dicee.Keci method*), 185  
`forward_k_vs_all()` (*dicee.models.AConEx method*), 127  
`forward_k_vs_all()` (*dicee.models.AConvO method*), 140  
`forward_k_vs_all()` (*dicee.models.AConvQ method*), 135  
`forward_k_vs_all()` (*dicee.models.base\_model.BaseKGE method*), 73  
`forward_k_vs_all()` (*dicee.models.BaseKGE method*), 119, 122, 126, 131, 137, 150, 153  
`forward_k_vs_all()` (*dicee.models.clifford.DeCaL method*), 80  
`forward_k_vs_all()` (*dicee.models.clifford.Keci method*), 77  
`forward_k_vs_all()` (*dicee.models.ComplEx method*), 128  
`forward_k_vs_all()` (*dicee.models.complex.AConEx method*), 83  
`forward_k_vs_all()` (*dicee.models.complex.ComplEx method*), 84  
`forward_k_vs_all()` (*dicee.models.complex.ConEx method*), 82  
`forward_k_vs_all()` (*dicee.models.ConEx method*), 127  
`forward_k_vs_all()` (*dicee.models.ConvO method*), 140  
`forward_k_vs_all()` (*dicee.models.ConvQ method*), 134  
`forward_k_vs_all()` (*dicee.models.DeCaL method*), 145  
`forward_k_vs_all()` (*dicee.models.DistMult method*), 123  
`forward_k_vs_all()` (*dicee.models.DualE method*), 157  
`forward_k_vs_all()` (*dicee.models.dualE.DualE method*), 85  
`forward_k_vs_all()` (*dicee.models.Keci method*), 143  
`forward_k_vs_all()` (*dicee.models.octonion.AConvO method*), 94  
`forward_k_vs_all()` (*dicee.models.octonion.ConvO method*), 94  
`forward_k_vs_all()` (*dicee.models.octonion.OMult method*), 93  
`forward_k_vs_all()` (*dicee.models.OMult method*), 139  
`forward_k_vs_all()` (*dicee.models.pykeen\_models.PykeenKGE method*), 95  
`forward_k_vs_all()` (*dicee.models.PykeenKGE method*), 150  
`forward_k_vs_all()` (*dicee.models.QMult method*), 133  
`forward_k_vs_all()` (*dicee.models.quaternion.AConvQ method*), 99  
`forward_k_vs_all()` (*dicee.models.quaternion.ConvQ method*), 98  
`forward_k_vs_all()` (*dicee.models.quaternion.QMult method*), 97  
`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*), 124  
`forward_k_vs_all()` (*dicee.models.TransE method*), 123  
`forward_k_vs_all()` (*dicee.OMult method*), 198  
`forward_k_vs_all()` (*dicee.PykeenKGE method*), 200  
`forward_k_vs_all()` (*dicee.QMult method*), 197  
`forward_k_vs_all()` (*dicee.Shallom method*), 199  
`forward_k_vs_all()` (*dicee.TransE method*), 186  
`forward_k_vs_sample()` (*dicee.AConEx method*), 192  
`forward_k_vs_sample()` (*dicee.BaseKGE method*), 205  
`forward_k_vs_sample()` (*dicee.ComplEx method*), 192  
`forward_k_vs_sample()` (*dicee.ConEx method*), 196  
`forward_k_vs_sample()` (*dicee.DistMult method*), 183  
`forward_k_vs_sample()` (*dicee.Keci method*), 186  
`forward_k_vs_sample()` (*dicee.models.AConEx method*), 128  
`forward_k_vs_sample()` (*dicee.models.base\_model.BaseKGE method*), 73  
`forward_k_vs_sample()` (*dicee.models.BaseKGE method*), 119, 122, 126, 131, 137, 150, 153  
`forward_k_vs_sample()` (*dicee.models.clifford.Keci method*), 78  
`forward_k_vs_sample()` (*dicee.models.ComplEx method*), 129  
`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*), 127  
`forward_k_vs_sample()` (*dicee.models.DistMult method*), 123  
`forward_k_vs_sample()` (*dicee.models.Keci method*), 143  
`forward_k_vs_sample()` (*dicee.models.pykeen\_models.PykeenKGE method*), 95  
`forward_k_vs_sample()` (*dicee.models.PykeenKGE method*), 151  
`forward_k_vs_sample()` (*dicee.models.QMult method*), 133  
`forward_k_vs_sample()` (*dicee.models.quaternion.QMult method*), 97

`forward_k_vs_sample()` (*dicee.models.real.DistMult method*), 99  
`forward_k_vs_sample()` (*dicee.PykeenKGE method*), 201  
`forward_k_vs_sample()` (*dicee.QMult method*), 197  
`forward_k_vs_with_explicit()` (*dicee.Keci method*), 185  
`forward_k_vs_with_explicit()` (*dicee.models.clifford.Keci method*), 77  
`forward_k_vs_with_explicit()` (*dicee.models.Keci method*), 143  
`forward_triples()` (*dicee.AConEx method*), 192  
`forward_triples()` (*dicee.AConvO method*), 193  
`forward_triples()` (*dicee.AConvQ method*), 193  
`forward_triples()` (*dicee.BaseKGE method*), 205  
`forward_triples()` (*dicee.ConEx method*), 196  
`forward_triples()` (*dicee.ConvO method*), 195  
`forward_triples()` (*dicee.ConvQ method*), 194  
`forward_triples()` (*dicee.DeCaL method*), 187  
`forward_triples()` (*dicee.DualE method*), 190  
`forward_triples()` (*dicee.Keci method*), 186  
`forward_triples()` (*dicee.LFMult method*), 199  
`forward_triples()` (*dicee.models.AConEx method*), 127  
`forward_triples()` (*dicee.models.AConvO method*), 140  
`forward_triples()` (*dicee.models.AConvQ method*), 134  
`forward_triples()` (*dicee.models.base\_model.BaseKGE method*), 73  
`forward_triples()` (*dicee.models.BaseKGE method*), 119, 122, 126, 131, 137, 149, 153  
`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*), 127  
`forward_triples()` (*dicee.models.ConvO method*), 140  
`forward_triples()` (*dicee.models.ConvQ method*), 134  
`forward_triples()` (*dicee.models.DeCaL method*), 145  
`forward_triples()` (*dicee.models.DualE method*), 157  
`forward_triples()` (*dicee.models.dualE.DualE method*), 85  
`forward_triples()` (*dicee.models.FMult method*), 154  
`forward_triples()` (*dicee.models.FMult2 method*), 155  
`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*), 154  
`forward_triples()` (*dicee.models.Keci method*), 144  
`forward_triples()` (*dicee.models.LFMult method*), 155  
`forward_triples()` (*dicee.models.LFMult1 method*), 155  
`forward_triples()` (*dicee.models.octonion.AConvO method*), 94  
`forward_triples()` (*dicee.models.octonion.ConvO method*), 94  
`forward_triples()` (*dicee.models.Pyke method*), 124  
`forward_triples()` (*dicee.models.pykeen\_models.PykeenKGE method*), 95  
`forward_triples()` (*dicee.models.PykeenKGE method*), 151  
`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*), 124  
`forward_triples()` (*dicee.Pyke method*), 182  
`forward_triples()` (*dicee.PykeenKGE method*), 201  
`forward_triples()` (*dicee.Shallom method*), 199  
`freeze_entity_embeddings` (*dicee.models.literal.LiteralEmbeddings attribute*), 91  
`frequency` (*dicee.callbacks.Perturb attribute*), 27  
`from_pretrained()` (*dicee.models.transformers.GPT class method*), 107  
`from_pretrained_model_write_embeddings_into_csv()` (*in module dicee*), 208  
`from_pretrained_model_write_embeddings_into_csv()` (*in module dicee.static\_funcs*), 171  
`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*), 155  
`function()` (*dicee.models.function\_space.FMult2 method*), 88

## G

`gamma` (*dicee.models.FMult* attribute), 154  
`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), 105  
`gen_test` (*dicee.query\_generator.QueryGenerator* attribute), 158  
`gen_test` (*dicee.QueryGenerator* attribute), 226  
`gen_valid` (*dicee.query\_generator.QueryGenerator* attribute), 158  
`gen_valid` (*dicee.QueryGenerator* attribute), 226  
`generate` () (*dicee.BytE* method), 202  
`generate` () (*dicee.KGE* method), 209  
`generate` () (*dicee.knowledge\_graph\_embeddings.KGE* method), 53  
`generate` () (*dicee.models.transformers.BytE* method), 102  
`generate_queries` () (*dicee.query\_generator.QueryGenerator* method), 158  
`generate_queries` () (*dicee.QueryGenerator* method), 227  
`get_aswa_state_dict` () (*dicee.callbacks.ASWA* method), 25  
`get_bpe_head_and_relation_representation` () (*dicee.BaseKGE* method), 205  
`get_bpe_head_and_relation_representation` () (*dicee.models.base\_model.BaseKGE* method), 74  
`get_bpe_head_and_relation_representation` () (*dicee.models.BaseKGE* method), 119, 123, 126, 131, 137, 150, 153  
`get_bpe_token_representation` () (*dicee.abstracts.BaseInteractiveKGE* method), 14  
`get_callbacks` () (in module *dicee.trainer.dice\_trainer*), 173  
`get_default_arguments` () (in module *dicee.analyse\_experiments*), 19  
`get_default_arguments` () (in module *dicee.scripts.index\_serve*), 167  
`get_default_arguments` () (in module *dicee.scripts.run*), 168  
`get_ee_vocab` () (in module *dicee*), 206  
`get_ee_vocab` () (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 163  
`get_ee_vocab` () (in module *dicee.static\_funcs*), 169  
`get_ee_vocab` () (in module *dicee.static\_preprocess\_funcs*), 172  
`get_embeddings` () (*dicee.BaseKGE* method), 205  
`get_embeddings` () (*dicee.EnsembleKGE* method), 206  
`get_embeddings` () (*dicee.models.base\_model.BaseKGE* method), 74  
`get_embeddings` () (*dicee.models.BaseKGE* method), 120, 123, 126, 131, 137, 150, 153  
`get_embeddings` () (*dicee.models.ensemble.EnsembleKGE* method), 86  
`get_embeddings` () (*dicee.models.real.Shallom* method), 100  
`get_embeddings` () (*dicee.models.Shallom* method), 124  
`get_embeddings` () (*dicee.Shallom* method), 199  
`get_entity_embeddings` () (*dicee.abstracts.BaseInteractiveKGE* method), 15  
`get_entity_index` () (*dicee.abstracts.BaseInteractiveKGE* method), 15  
`get_er_vocab` () (in module *dicee*), 206  
`get_er_vocab` () (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 163  
`get_er_vocab` () (in module *dicee.static\_funcs*), 169  
`get_er_vocab` () (in module *dicee.static\_preprocess\_funcs*), 172  
`get_eval_report` () (*dicee.abstracts.BaseInteractiveKGE* method), 14  
`get_head_relation_representation` () (*dicee.BaseKGE* method), 205  
`get_head_relation_representation` () (*dicee.models.base\_model.BaseKGE* method), 74  
`get_head_relation_representation` () (*dicee.models.BaseKGE* method), 119, 122, 126, 131, 137, 150, 153  
`get_kronecker_triple_representation` () (*dicee.callbacks.KronE* method), 26  
`get_num_params` () (*dicee.models.transformers.GPT* method), 107  
`get_padded_bpe_triple_representation` () (*dicee.abstracts.BaseInteractiveKGE* method), 14  
`get_queries` () (*dicee.query\_generator.QueryGenerator* method), 159  
`get_queries` () (*dicee.QueryGenerator* method), 227  
`get_re_vocab` () (in module *dicee*), 206  
`get_re_vocab` () (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 163  
`get_re_vocab` () (in module *dicee.static\_funcs*), 169  
`get_re_vocab` () (in module *dicee.static\_preprocess\_funcs*), 172  
`get_relation_embeddings` () (*dicee.abstracts.BaseInteractiveKGE* method), 15  
`get_relation_index` () (*dicee.abstracts.BaseInteractiveKGE* method), 15  
`get_sentence_representation` () (*dicee.BaseKGE* method), 205  
`get_sentence_representation` () (*dicee.models.base\_model.BaseKGE* method), 74  
`get_sentence_representation` () (*dicee.models.BaseKGE* method), 119, 123, 126, 131, 137, 150, 153  
`get_transductive_entity_embeddings` () (*dicee.KGE* method), 209  
`get_transductive_entity_embeddings` () (*dicee.knowledge\_graph\_embeddings.KGE* method), 53  
`get_triple_representation` () (*dicee.BaseKGE* method), 205  
`get_triple_representation` () (*dicee.models.base\_model.BaseKGE* method), 73  
`get_triple_representation` () (*dicee.models.BaseKGE* method), 119, 122, 126, 131, 137, 150, 153  
`GFMult` (class in *dicee.models*), 154  
`GFMult` (class in *dicee.models.function\_space*), 87

global\_rank (*dicee.abstracts.AbstractTrainer attribute*), 13  
 global\_rank (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 177  
 GPT (*class in dicee.models.transformers*), 106  
 GPTConfig (*class in dicee.models.transformers*), 106  
 gpus (*dicee.config.Namespace attribute*), 30  
 gradient\_accumulation\_steps (*dicee.config.Namespace attribute*), 31  
 ground\_queries () (*dicee.query\_generator.QueryGenerator method*), 158  
 ground\_queries () (*dicee.QueryGenerator method*), 226

## H

hidden\_dim (*dicee.models.literal.LiteralEmbeddings attribute*), 91  
 hidden\_dropout (*dicee.BaseKGE attribute*), 204  
 hidden\_dropout (*dicee.models.base\_model.BaseKGE attribute*), 73  
 hidden\_dropout (*dicee.models.BaseKGE attribute*), 119, 122, 125, 130, 136, 149, 152  
 hidden\_dropout\_rate (*dicee.BaseKGE attribute*), 204  
 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*), 118, 121, 125, 130, 136, 148, 152  
 hidden\_normalizer (*dicee.BaseKGE attribute*), 204  
 hidden\_normalizer (*dicee.models.base\_model.BaseKGE attribute*), 73  
 hidden\_normalizer (*dicee.models.BaseKGE attribute*), 118, 122, 125, 130, 136, 149, 152

## I

IdentityClass (*class in dicee.models*), 120, 131, 137  
 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*), 167  
 index\_triple () (*dicee.abstracts.BaseInteractiveKGE method*), 15  
 init\_data\_loader () (*dicee.DICE\_Trainer method*), 209  
 init\_data\_loader () (*dicee.trainer.DICE\_Trainer method*), 179  
 init\_data\_loader () (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 174  
 init\_dataset () (*dicee.DICE\_Trainer method*), 209  
 init\_dataset () (*dicee.trainer.DICE\_Trainer method*), 179  
 init\_dataset () (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 174  
 init\_param (*dicee.config.Namespace attribute*), 31  
 init\_params\_with\_sanity\_checking () (*dicee.BaseKGE method*), 205  
 init\_params\_with\_sanity\_checking () (*dicee.models.base\_model.BaseKGE method*), 73  
 init\_params\_with\_sanity\_checking () (*dicee.models.BaseKGE method*), 119, 122, 126, 130, 136, 149, 153  
 initial\_eval\_setting (*dicee.callbacks.ASWA attribute*), 24  
 initialize\_or\_load\_model () (*dicee.DICE\_Trainer method*), 209  
 initialize\_or\_load\_model () (*dicee.trainer.DICE\_Trainer method*), 179  
 initialize\_or\_load\_model () (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 174  
 initialize\_trainer () (*dicee.DICE\_Trainer method*), 208  
 initialize\_trainer () (*dicee.trainer.DICE\_Trainer method*), 179  
 initialize\_trainer () (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 174  
 initialize\_trainer () (*in module dicee.trainer.dice\_trainer*), 173  
 input\_dp\_ent\_real (*dicee.BaseKGE attribute*), 204  
 input\_dp\_ent\_real (*dicee.models.base\_model.BaseKGE attribute*), 73  
 input\_dp\_ent\_real (*dicee.models.BaseKGE attribute*), 119, 122, 125, 130, 136, 149, 152  
 input\_dp\_rel\_real (*dicee.BaseKGE attribute*), 204  
 input\_dp\_rel\_real (*dicee.models.base\_model.BaseKGE attribute*), 73  
 input\_dp\_rel\_real (*dicee.models.BaseKGE attribute*), 119, 122, 125, 130, 136, 149, 152  
 input\_dropout\_rate (*dicee.BaseKGE attribute*), 204  
 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*), 118, 121, 125, 129, 136, 148, 152  
 InteractiveQueryDecomposition (*class in dicee.abstracts*), 16  
 initialize\_model () (*in module dicee*), 207  
 initialize\_model () (*in module dicee.static\_funcs*), 170  
 is\_continual\_training (*dicee.DICE\_Trainer attribute*), 208  
 is\_continual\_training (*dicee.evaluator.Evaluator attribute*), 48  
 is\_continual\_training (*dicee.executer.Execute attribute*), 50  
 is\_continual\_training (*dicee.trainer.DICE\_Trainer attribute*), 178  
 is\_continual\_training (*dicee.trainer.dice\_trainer.DICE\_Trainer attribute*), 173  
 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`), 166

## K

`k` (*dicee.models.FMult* attribute), 153  
`k` (*dicee.models.FMult2* attribute), 154  
`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), 154  
`k_fold_cross_validation()` (*dicee.DICE\_Trainer* method), 209  
`k_fold_cross_validation()` (*dicee.trainer.DICE\_Trainer* method), 179  
`k_fold_cross_validation()` (*dicee.trainer.dice\_trainer.DICE\_Trainer* method), 174  
`k_vs_all_score()` (*dicee.ComplEx* static method), 192  
`k_vs_all_score()` (*dicee.DistMult* method), 183  
`k_vs_all_score()` (*dicee.Keci* method), 185  
`k_vs_all_score()` (*dicee.models.clifford.Keci* method), 77  
`k_vs_all_score()` (*dicee.models.ComplEx* static method), 128  
`k_vs_all_score()` (*dicee.models.complex.ComplEx* static method), 84  
`k_vs_all_score()` (*dicee.models.DistMult* method), 123  
`k_vs_all_score()` (*dicee.models.Keci* method), 143  
`k_vs_all_score()` (*dicee.models.octonion.OMult* method), 93  
`k_vs_all_score()` (*dicee.models.OMult* method), 139  
`k_vs_all_score()` (*dicee.models.QMult* method), 133  
`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), 198  
`k_vs_all_score()` (*dicee.QMult* method), 197  
*Keci* (class in *dicee*), 183  
*Keci* (class in *dicee.models*), 141  
*Keci* (class in *dicee.models.clifford*), 75  
`kernel_size` (*dicee.BaseKGE* attribute), 204  
`kernel_size` (*dicee.config.Namespace* attribute), 31  
`kernel_size` (*dicee.models.base\_model.BaseKGE* attribute), 72  
`kernel_size` (*dicee.models.BaseKGE* attribute), 118, 121, 125, 130, 136, 148, 152  
*KG* (class in *dicee.knowledge\_graph*), 51  
`kg` (*dicee.callbacks.PseudoLabellingCallback* attribute), 24  
`kg` (*dicee.read\_preprocess\_save\_load\_kg.LoadSaveToDisk* attribute), 165  
`kg` (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG* attribute), 164  
`kg` (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG* attribute), 159  
`kg` (*dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk.ReadFromDisk* attribute), 160  
`kg` (*dicee.read\_preprocess\_save\_load\_kg.ReadFromDisk* attribute), 165  
`kg` (*dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk.LoadSaveToDisk* attribute), 160  
*KGE* (class in *dicee*), 209  
*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*), 215  
*KvsAll* (class in *dicee.dataset\_classes*), 35  
`kvsall_score()` (*dicee.DualE* method), 190  
`kvsall_score()` (*dicee.models.DualE* method), 157  
`kvsall_score()` (*dicee.models.dualE.DualE* method), 85  
*KvsSampleDataset* (class in *dicee*), 218  
*KvsSampleDataset* (class in *dicee.dataset\_classes*), 39

## L

`label_smoothing_rate` (*dicee.AllvsAll* attribute), 216  
`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), 216  
`label_smoothing_rate` (*dicee.KvsSampleDataset* attribute), 219  
`label_smoothing_rate` (*dicee.OnevsSample* attribute), 217, 218  
`label_smoothing_rate` (*dicee.TriplePredictionDataset* attribute), 220

- labels (*dicee.dataset\_classes.NegSampleDataset* attribute), 40
- labels (*dicee.NegSampleDataset* attribute), 219
- layer\_norm (*dicee.models.literal.LiteralEmbeddings* attribute), 91
- LayerNorm (*class in dicee.models.transformers*), 103
- learning\_rate (*dicee.BaseKGE* attribute), 204
- learning\_rate (*dicee.models.base\_model.BaseKGE* attribute), 72
- learning\_rate (*dicee.models.BaseKGE* attribute), 118, 121, 125, 129, 135, 148, 152
- length (*dicee.dataset\_classes.NegSampleDataset* attribute), 40
- length (*dicee.dataset\_classes.TriplePredictionDataset* attribute), 41
- length (*dicee.NegSampleDataset* attribute), 219
- length (*dicee.TriplePredictionDataset* attribute), 220
- level (*dicee.callbacks.Perturb* attribute), 27
- LFMult (*class in dicee*), 199
- LFMult (*class in dicee.models*), 155
- LFMult (*class in dicee.models.function\_space*), 88
- LFMult1 (*class in dicee.models*), 155
- LFMult1 (*class in dicee.models.function\_space*), 88
- linear() (*dicee.LFMult* method), 199
- linear() (*dicee.models.function\_space.LFMult* method), 89
- linear() (*dicee.models.LFMult* method), 156
- list2tuple() (*dicee.query\_generator.QueryGenerator* method), 158
- list2tuple() (*dicee.QueryGenerator* method), 226
- LiteralDataset (*class in dicee*), 224
- LiteralDataset (*class in dicee.dataset\_classes*), 44
- LiteralEmbeddings (*class in dicee.models.literal*), 90
- lm\_head (*dicee.BytE* attribute), 202
- lm\_head (*dicee.models.transformers.BytE* attribute), 102
- lm\_head (*dicee.models.transformers.GPT* attribute), 107
- ln\_1 (*dicee.models.transformers.Block* attribute), 105
- ln\_2 (*dicee.models.transformers.Block* attribute), 106
- load() (*dicee.read\_preprocess\_save\_load\_kg.LoadSaveToDisk* method), 165
- load() (*dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk.LoadSaveToDisk* method), 160
- load\_and\_validate\_literal\_data() (*dicee.dataset\_classes.LiteralDataset* static method), 46
- load\_and\_validate\_literal\_data() (*dicee.LiteralDataset* static method), 225
- load\_from\_memmap() (*dicee.executer.Execute* method), 50
- load\_json() (*in module dicee*), 207
- load\_json() (*in module dicee.static\_funcs*), 170
- load\_model() (*in module dicee*), 206
- load\_model() (*in module dicee.static\_funcs*), 170
- load\_model\_ensemble() (*in module dicee*), 206
- load\_model\_ensemble() (*in module dicee.static\_funcs*), 170
- load\_numpy() (*in module dicee*), 207
- load\_numpy() (*in module dicee.static\_funcs*), 171
- load\_numpy\_ndarray() (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 164
- load\_pickle() (*in module dicee*), 206
- load\_pickle() (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 164
- load\_pickle() (*in module dicee.static\_funcs*), 170
- load\_queries() (*dicee.query\_generator.QueryGenerator* method), 159
- load\_queries() (*dicee.QueryGenerator* method), 227
- load\_queries\_and\_answers() (*dicee.query\_generator.QueryGenerator* static method), 159
- load\_queries\_and\_answers() (*dicee.QueryGenerator* static method), 227
- load\_state\_dict() (*dicee.EnsembleKGE* method), 206
- load\_state\_dict() (*dicee.models.ensemble.EnsembleKGE* method), 86
- load\_term\_mapping() (*in module dicee*), 206, 213
- load\_term\_mapping() (*in module dicee.static\_funcs*), 170
- load\_term\_mapping() (*in module dicee.trainer.dice\_trainer*), 173
- load\_with\_pandas() (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 164
- loader\_backend (*dicee.dataset\_classes.LiteralDataset* attribute), 45, 46
- loader\_backend (*dicee.LiteralDataset* attribute), 225
- LoadSaveToDisk (*class in dicee.read\_preprocess\_save\_load\_kg*), 165
- LoadSaveToDisk (*class in dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk*), 160
- local\_rank (*dicee.abstracts.AbstractTrainer* attribute), 13
- local\_rank (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 177
- loss (*dicee.BaseKGE* attribute), 204
- loss (*dicee.models.base\_model.BaseKGE* attribute), 72
- loss (*dicee.models.BaseKGE* attribute), 118, 121, 125, 130, 136, 149, 152
- loss\_func (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 178
- loss\_function (*dicee.trainer.torch\_trainer.TorchTrainer* attribute), 176

- `loss_function()` (*dicee.BytE* method), 202
- `loss_function()` (*dicee.models.base\_model.BaseKGELightning* method), 67
- `loss_function()` (*dicee.models.BaseKGELightning* method), 113
- `loss_function()` (*dicee.models.transformers.BytE* method), 102
- `loss_history` (*dicee.BaseKGE* attribute), 204
- `loss_history` (*dicee.models.base\_model.BaseKGE* attribute), 73
- `loss_history` (*dicee.models.BaseKGE* attribute), 119, 122, 125, 130, 136, 149, 152
- `loss_history` (*dicee.models.pykeen\_models.PykeenKGE* attribute), 95
- `loss_history` (*dicee.models.PykeenKGE* attribute), 150
- `loss_history` (*dicee.PykeenKGE* attribute), 200
- `loss_history` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 178
- `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), 199
- `m` (*dicee.models.function\_space.LFMult* attribute), 89
- `m` (*dicee.models.LFMult* attribute), 155
- `main()` (in module *dicee.scripts.index\_serve*), 168
- `main()` (in module *dicee.scripts.run*), 168
- `make_iterable_verbose()` (in module *dicee.static\_funcs\_training*), 171
- `make_iterable_verbose()` (in module *dicee.trainer.torch\_trainer\_ddp*), 177
- `mapping_from_first_two_cols_to_third()` (in module *dicee*), 213
- `mapping_from_first_two_cols_to_third()` (in module *dicee.static\_preprocess\_funcs*), 173
- `margin` (*dicee.models.Pyke* attribute), 124
- `margin` (*dicee.models.real.Pyke* attribute), 100
- `margin` (*dicee.models.real.TransE* attribute), 99
- `margin` (*dicee.models.TransE* attribute), 123
- `margin` (*dicee.Pyke* attribute), 182
- `margin` (*dicee.TransE* attribute), 186
- `max_ans_num` (*dicee.query\_generator.QueryGenerator* attribute), 158
- `max_ans_num` (*dicee.QueryGenerator* attribute), 226
- `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), 204
- `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), 119, 122, 126, 130, 136, 149, 152
- `max_num_of_classes` (*dicee.dataset\_classes.KvsSampleDataset* attribute), 39
- `max_num_of_classes` (*dicee.KvsSampleDataset* attribute), 219
- `mem_of_model()` (*dicee.EnsembleKGE* method), 206
- `mem_of_model()` (*dicee.models.base\_model.BaseKGELightning* method), 66
- `mem_of_model()` (*dicee.models.BaseKGELightning* method), 112
- `mem_of_model()` (*dicee.models.ensemble.EnsembleKGE* method), 86
- `method` (*dicee.callbacks.Perturb* attribute), 27
- MLP* (class in *dicee.models.transformers*), 104
- `mlp` (*dicee.models.transformers.Block* attribute), 106
- `mode` (*dicee.query\_generator.QueryGenerator* attribute), 158
- `mode` (*dicee.QueryGenerator* attribute), 226
- `model` (*dicee.config.Namespace* attribute), 30
- `model` (*dicee.models.pykeen\_models.PykeenKGE* attribute), 95
- `model` (*dicee.models.PykeenKGE* attribute), 150
- `model` (*dicee.PykeenKGE* attribute), 200
- `model` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 178
- `model` (*dicee.trainer.torch\_trainer.TorchTrainer* attribute), 176
- `model_kwargs` (*dicee.models.pykeen\_models.PykeenKGE* attribute), 95
- `model_kwargs` (*dicee.models.PykeenKGE* attribute), 150
- `model_kwargs` (*dicee.PykeenKGE* attribute), 200
- `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`, 100
  - `dicee.models.transformers`, 100
- `dicee.query_generator`, 157
- `dicee.read_preprocess_save_load_kg`, 159
  - `dicee.read_preprocess_save_load_kg.preprocess`, 159
  - `dicee.read_preprocess_save_load_kg.read_from_disk`, 160
  - `dicee.read_preprocess_save_load_kg.save_load_disk`, 160
  - `dicee.read_preprocess_save_load_kg.util`, 161
- `dicee.sanity_checkers`, 165
- `dicee.scripts`, 166
  - `dicee.scripts.index_serve`, 166
  - `dicee.scripts.run`, 168
- `dicee.static_funcs`, 168
- `dicee.static_funcs_training`, 171
- `dicee.static_preprocess_funcs`, 172
- `dicee.trainer`, 173
  - `dicee.trainer.dice_trainer`, 173
  - `dicee.trainer.model_parallelism`, 175
  - `dicee.trainer.torch_trainer`, 175
  - `dicee.trainer.torch_trainer_ddp`, 177
- `modules()` (*dicee.EnsembleKGE method*), 206
- `modules()` (*dicee.models.ensemble.EnsembleKGE method*), 86
- `moving_average()` (*dicee.callbacks.SWA static method*), 29
- `MultiClassClassificationDataset` (class in *dicee*), 214
- `MultiClassClassificationDataset` (class in *dicee.dataset\_classes*), 34
- `MultiLabelDataset` (class in *dicee*), 213
- `MultiLabelDataset` (class in *dicee.dataset\_classes*), 34

## N

- `n` (*dicee.models.FMult2 attribute*), 154
- `n` (*dicee.models.function\_space.FMult2 attribute*), 88
- `n_embd` (*dicee.models.transformers.CausalSelfAttention attribute*), 104
- `n_embd` (*dicee.models.transformers.GPTConfig attribute*), 106
- `n_epochs_eval_model` (*dicee.callbacks.PeriodicEvalCallback attribute*), 27
- `n_epochs_eval_model` (*dicee.config.Namespace attribute*), 32
- `n_head` (*dicee.models.transformers.CausalSelfAttention attribute*), 104
- `n_head` (*dicee.models.transformers.GPTConfig attribute*), 106
- `n_layer` (*dicee.models.transformers.GPTConfig attribute*), 106
- `n_layers` (*dicee.models.FMult2 attribute*), 154
- `n_layers` (*dicee.models.function\_space.FMult2 attribute*), 88
- `name` (*dicee.abstracts.BaseInteractiveKGE property*), 15
- `name` (*dicee.AConEx attribute*), 192
- `name` (*dicee.AConvO attribute*), 192
- `name` (*dicee.AConvQ attribute*), 193
- `name` (*dicee.Byte attribute*), 201

name (*dicee.CKeci attribute*), 183  
 name (*dicee.ComplEx attribute*), 192  
 name (*dicee.ConEx attribute*), 195  
 name (*dicee.ConvO attribute*), 195  
 name (*dicee.ConvQ attribute*), 194  
 name (*dicee.DeCaL attribute*), 187  
 name (*dicee.DistMult attribute*), 183  
 name (*dicee.DualE attribute*), 190  
 name (*dicee.EnsembleKGE attribute*), 205  
 name (*dicee.Keci attribute*), 184  
 name (*dicee.LFMult attribute*), 199  
 name (*dicee.models.AConEx attribute*), 127  
 name (*dicee.models.AConvO attribute*), 140  
 name (*dicee.models.AConvQ attribute*), 134  
 name (*dicee.models.CKeci attribute*), 144  
 name (*dicee.models.clifford.CKeci attribute*), 78  
 name (*dicee.models.clifford.DeCaL attribute*), 79  
 name (*dicee.models.clifford.Keci attribute*), 75  
 name (*dicee.models.ComplEx attribute*), 128  
 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*), 127  
 name (*dicee.models.ConvO attribute*), 140  
 name (*dicee.models.ConvQ attribute*), 134  
 name (*dicee.models.DeCaL attribute*), 144  
 name (*dicee.models.DistMult attribute*), 123  
 name (*dicee.models.DualE attribute*), 156  
 name (*dicee.models.dualE.DualE attribute*), 84  
 name (*dicee.models.ensemble.EnsembleKGE attribute*), 86  
 name (*dicee.models.FMult attribute*), 153  
 name (*dicee.models.FMult2 attribute*), 154  
 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*), 154  
 name (*dicee.models.Keci attribute*), 141  
 name (*dicee.models.LFMult attribute*), 155  
 name (*dicee.models.LFMult1 attribute*), 155  
 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*), 139  
 name (*dicee.models.Pyke attribute*), 124  
 name (*dicee.models.pykeen\_models.PykeenKGE attribute*), 95  
 name (*dicee.models.PykeenKGE attribute*), 150  
 name (*dicee.models.QMult attribute*), 133  
 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.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*), 123  
 name (*dicee.models.TransE attribute*), 123  
 name (*dicee.models.transformers.BytE attribute*), 101  
 name (*dicee.OMult attribute*), 198  
 name (*dicee.Pyke attribute*), 182  
 name (*dicee.PykeenKGE attribute*), 200  
 name (*dicee.QMult attribute*), 197  
 name (*dicee.Shallom attribute*), 198  
 name (*dicee.TransE attribute*), 186  
 named\_children() (*dicee.EnsembleKGE method*), 206  
 named\_children() (*dicee.models.ensemble.EnsembleKGE method*), 86  
 Namespace (*class in dicee.config*), 29

neg\_ratio (*dicее.BPE\_NegativeSamplingDataset* attribute), 213  
 neg\_ratio (*dicее.config.Namespace* attribute), 30  
 neg\_ratio (*dicее.dataset\_classes.BPE\_NegativeSamplingDataset* attribute), 34  
 neg\_ratio (*dicее.dataset\_classes.KvsSampleDataset* attribute), 39  
 neg\_ratio (*dicее.KvsSampleDataset* attribute), 219  
 neg\_sample\_ratio (*dicее.CVDataModule* attribute), 221  
 neg\_sample\_ratio (*dicее.dataset\_classes.CVDataModule* attribute), 41  
 neg\_sample\_ratio (*dicее.dataset\_classes.NegSampleDataset* attribute), 40  
 neg\_sample\_ratio (*dicее.dataset\_classes.OnevsSample* attribute), 38  
 neg\_sample\_ratio (*dicее.dataset\_classes.TriplePredictionDataset* attribute), 41  
 neg\_sample\_ratio (*dicее.NegSampleDataset* attribute), 219  
 neg\_sample\_ratio (*dicее.OnevsSample* attribute), 217, 218  
 neg\_sample\_ratio (*dicее.TriplePredictionDataset* attribute), 220  
 negnorm () (*dicее.abstracts.InteractiveQueryDecomposition* method), 16  
 NegSampleDataset (class in *dicее*), 219  
 NegSampleDataset (class in *dicее.dataset\_classes*), 39  
 neural\_searcher (in module *dicее.scripts.index\_serve*), 167  
 NeuralSearcher (class in *dicее.scripts.index\_serve*), 167  
 NodeTrainer (class in *dicее.trainer.torch\_trainer\_ddp*), 177  
 norm\_fc1 (*dicее.AConEx* attribute), 192  
 norm\_fc1 (*dicее.AConvO* attribute), 193  
 norm\_fc1 (*dicее.ConEx* attribute), 196  
 norm\_fc1 (*dicее.ConvO* attribute), 195  
 norm\_fc1 (*dicее.models.AConEx* attribute), 127  
 norm\_fc1 (*dicее.models.AConvO* attribute), 140  
 norm\_fc1 (*dicее.models.complex.AConEx* attribute), 83  
 norm\_fc1 (*dicее.models.complex.ConEx* attribute), 82  
 norm\_fc1 (*dicее.models.ConEx* attribute), 127  
 norm\_fc1 (*dicее.models.ConvO* attribute), 140  
 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), 224  
 normalization\_params (*dicее.dataset\_classes.LiteralDataset* attribute), 45, 46  
 normalization\_params (*dicее.LiteralDataset* attribute), 224, 225  
 normalization\_type (*dicее.dataset\_classes.LiteralDataset* attribute), 46  
 normalization\_type (*dicее.LiteralDataset* attribute), 225  
 normalize\_head\_entity\_embeddings (*dicее.BaseKGE* attribute), 204  
 normalize\_head\_entity\_embeddings (*dicее.models.base\_model.BaseKGE* attribute), 73  
 normalize\_head\_entity\_embeddings (*dicее.models.BaseKGE* attribute), 118, 122, 125, 130, 136, 149, 152  
 normalize\_relation\_embeddings (*dicее.BaseKGE* attribute), 204  
 normalize\_relation\_embeddings (*dicее.models.base\_model.BaseKGE* attribute), 73  
 normalize\_relation\_embeddings (*dicее.models.BaseKGE* attribute), 118, 122, 125, 130, 136, 149, 152  
 normalize\_tail\_entity\_embeddings (*dicее.BaseKGE* attribute), 204  
 normalize\_tail\_entity\_embeddings (*dicее.models.base\_model.BaseKGE* attribute), 73  
 normalize\_tail\_entity\_embeddings (*dicее.models.BaseKGE* attribute), 118, 122, 125, 130, 136, 149, 152  
 normalizer\_class (*dicее.BaseKGE* attribute), 204  
 normalizer\_class (*dicее.models.base\_model.BaseKGE* attribute), 73  
 normalizer\_class (*dicее.models.BaseKGE* attribute), 118, 122, 125, 130, 136, 149, 152  
 num\_bpe\_entities (*dicее.BPE\_NegativeSamplingDataset* attribute), 213  
 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), 225  
 num\_datapoints (*dicее.BPE\_NegativeSamplingDataset* attribute), 213  
 num\_datapoints (*dicее.dataset\_classes.BPE\_NegativeSamplingDataset* attribute), 34  
 num\_datapoints (*dicее.dataset\_classes.MultiLabelDataset* attribute), 34  
 num\_datapoints (*dicее.MultiLabelDataset* attribute), 214  
 num\_ent (*dicее.DualE* attribute), 190  
 num\_ent (*dicее.models.DualE* attribute), 157  
 num\_ent (*dicее.models.dualE.DualE* attribute), 85  
 num\_entities (*dicее.BaseKGE* attribute), 204  
 num\_entities (*dicее.CVDataModule* attribute), 221  
 num\_entities (*dicее.dataset\_classes.CVDataModule* attribute), 41  
 num\_entities (*dicее.dataset\_classes.KvsSampleDataset* attribute), 39

num\_entities (*dicdee.dataset\_classes.LiteralDataset* attribute), 45, 46  
 num\_entities (*dicdee.dataset\_classes.NegSampleDataset* attribute), 40  
 num\_entities (*dicdee.dataset\_classes.OnevsSample* attribute), 37, 38  
 num\_entities (*dicdee.dataset\_classes.TriplePredictionDataset* attribute), 41  
 num\_entities (*dicdee.evaluator.Evaluator* attribute), 48  
 num\_entities (*dicdee.knowledge\_graph.KG* attribute), 52  
 num\_entities (*dicdee.KvsSampleDataset* attribute), 219  
 num\_entities (*dicdee.LiteralDataset* attribute), 225  
 num\_entities (*dicdee.models.base\_model.BaseKGE* attribute), 72  
 num\_entities (*dicdee.models.BaseKGE* attribute), 118, 121, 125, 129, 135, 148, 152  
 num\_entities (*dicdee.NegSampleDataset* attribute), 219  
 num\_entities (*dicdee.OnevsSample* attribute), 217, 218  
 num\_entities (*dicdee.TriplePredictionDataset* attribute), 220  
 num\_epochs (*dicdee.abstracts.AbstractPPECallback* attribute), 17  
 num\_epochs (*dicdee.analyse\_experiments.Experiment* attribute), 19  
 num\_epochs (*dicdee.callbacks.ASWA* attribute), 24  
 num\_epochs (*dicdee.config.Namespace* attribute), 30  
 num\_epochs (*dicdee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 178  
 num\_folds\_for\_cv (*dicdee.config.Namespace* attribute), 31  
 num\_of\_data\_points (*dicdee.dataset\_classes.MultiClassClassificationDataset* attribute), 35  
 num\_of\_data\_points (*dicdee.MultiClassClassificationDataset* attribute), 214  
 num\_of\_data\_properties (*dicdee.models.literal.LiteralEmbeddings* attribute), 90, 91  
 num\_of\_epochs (*dicdee.callbacks.PseudoLabellingCallback* attribute), 24  
 num\_of\_output\_channels (*dicdee.BaseKGE* attribute), 204  
 num\_of\_output\_channels (*dicdee.config.Namespace* attribute), 31  
 num\_of\_output\_channels (*dicdee.models.base\_model.BaseKGE* attribute), 72  
 num\_of\_output\_channels (*dicdee.models.BaseKGE* attribute), 118, 121, 125, 130, 136, 149, 152  
 num\_params (*dicdee.analyse\_experiments.Experiment* attribute), 19  
 num\_relations (*dicdee.BaseKGE* attribute), 204  
 num\_relations (*dicdee.CVDDataModule* attribute), 221  
 num\_relations (*dicdee.dataset\_classes.CVDDataModule* attribute), 41  
 num\_relations (*dicdee.dataset\_classes.NegSampleDataset* attribute), 40  
 num\_relations (*dicdee.dataset\_classes.OnevsSample* attribute), 38  
 num\_relations (*dicdee.dataset\_classes.TriplePredictionDataset* attribute), 41  
 num\_relations (*dicdee.evaluator.Evaluator* attribute), 48  
 num\_relations (*dicdee.knowledge\_graph.KG* attribute), 52  
 num\_relations (*dicdee.models.base\_model.BaseKGE* attribute), 72  
 num\_relations (*dicdee.models.BaseKGE* attribute), 118, 121, 125, 129, 135, 148, 152  
 num\_relations (*dicdee.NegSampleDataset* attribute), 219  
 num\_relations (*dicdee.OnevsSample* attribute), 217, 218  
 num\_relations (*dicdee.TriplePredictionDataset* attribute), 220  
 num\_sample (*dicdee.models.FMult* attribute), 153  
 num\_sample (*dicdee.models.function\_space.FMult* attribute), 87  
 num\_sample (*dicdee.models.function\_space.GFMult* attribute), 87  
 num\_sample (*dicdee.models.GFMult* attribute), 154  
 num\_tokens (*dicdee.BaseKGE* attribute), 204  
 num\_tokens (*dicdee.knowledge\_graph.KG* attribute), 52  
 num\_tokens (*dicdee.models.base\_model.BaseKGE* attribute), 72  
 num\_tokens (*dicdee.models.BaseKGE* attribute), 118, 121, 125, 129, 135, 148, 152  
 num\_workers (*dicdee.CVDDataModule* attribute), 221  
 num\_workers (*dicdee.dataset\_classes.CVDDataModule* attribute), 41  
 numpy\_data\_type\_changer () (in module *dicdee*), 207  
 numpy\_data\_type\_changer () (in module *dicdee.static\_funcs*), 170

## O

octonion\_mul () (in module *dicdee.models*), 138  
 octonion\_mul () (in module *dicdee.models.octonion*), 92  
 octonion\_mul\_norm () (in module *dicdee.models*), 138  
 octonion\_mul\_norm () (in module *dicdee.models.octonion*), 92  
 octonion\_normalizer () (*dicdee.AConvO* static method), 193  
 octonion\_normalizer () (*dicdee.ConvO* static method), 195  
 octonion\_normalizer () (*dicdee.models.AConvO* static method), 140  
 octonion\_normalizer () (*dicdee.models.ConvO* static method), 140  
 octonion\_normalizer () (*dicdee.models.octonion.AConvO* static method), 94  
 octonion\_normalizer () (*dicdee.models.octonion.ConvO* static method), 94  
 octonion\_normalizer () (*dicdee.models.octonion.OMult* static method), 92  
 octonion\_normalizer () (*dicdee.models.OMult* static method), 139

octonion\_normalizer() (*dicee.OMult* static method), 198  
 OMult (*class in dicee*), 197  
 OMult (*class in dicee.models*), 138  
 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), 113  
 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*), 215  
 OnevsAllDataset (*class in dicee.dataset\_classes*), 35  
 OnevsSample (*class in dicee*), 217  
 OnevsSample (*class in dicee.dataset\_classes*), 37  
 optim (*dicee.config.Namespace* attribute), 30  
 optimizer (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 177  
 optimizer (*dicee.trainer.torch\_trainer.TorchTrainer* attribute), 176  
 optimizer\_name (*dicee.BaseKGE* attribute), 204  
 optimizer\_name (*dicee.models.base\_model.BaseKGE* attribute), 72  
 optimizer\_name (*dicee.models.BaseKGE* attribute), 118, 121, 125, 130, 136, 148, 152  
 ordered\_bpe\_entities (*dicee.BPE\_NegativeSamplingDataset* attribute), 213  
 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), 187  
 p (*dicee.Keci* attribute), 184  
 p (*dicee.models.clifford.DeCaL* attribute), 79  
 p (*dicee.models.clifford.Keci* attribute), 75

*p* (*dicee.models.DeCaL* attribute), 145  
*p* (*dicee.models.Keci* attribute), 141  
padding (*dicee.knowledge\_graph.KG* attribute), 52  
pandas\_dataframe\_indexer() (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 162  
param\_init (*dicee.BaseKGE* attribute), 204  
param\_init (*dicee.models.base\_model.BaseKGE* attribute), 73  
param\_init (*dicee.models.BaseKGE* attribute), 118, 122, 125, 130, 136, 149, 152  
parameters() (*dicee.abstracts.BaseInteractiveKGE* method), 16  
parameters() (*dicee.EnsembleKGE* method), 206  
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*), 161  
poly\_NN() (*dicee.LFMMult* method), 199  
poly\_NN() (*dicee.models.function\_space.LFMMult* method), 89  
poly\_NN() (*dicee.models.LFMMult* method), 156  
polynomial() (*dicee.LFMMult* method), 200  
polynomial() (*dicee.models.function\_space.LFMMult* method), 89  
polynomial() (*dicee.models.LFMMult* method), 156  
pop() (*dicee.LFMMult* method), 200  
pop() (*dicee.models.function\_space.LFMMult* method), 90  
pop() (*dicee.models.LFMMult* method), 156  
pq (*dicee.analyse\_experiments.Experiment* attribute), 20  
predict() (*dicee.KGE* method), 211  
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), 115  
predict\_literals() (*dicee.KGE* method), 212  
predict\_literals() (*dicee.knowledge\_graph\_embeddings.KGE* method), 56  
predict\_missing\_head\_entity() (*dicee.KGE* method), 209  
predict\_missing\_head\_entity() (*dicee.knowledge\_graph\_embeddings.KGE* method), 53  
predict\_missing\_relations() (*dicee.KGE* method), 210  
predict\_missing\_relations() (*dicee.knowledge\_graph\_embeddings.KGE* method), 54  
predict\_missing\_tail\_entity() (*dicee.KGE* method), 210  
predict\_missing\_tail\_entity() (*dicee.knowledge\_graph\_embeddings.KGE* method), 54  
predict\_topk() (*dicee.KGE* method), 211  
predict\_topk() (*dicee.knowledge\_graph\_embeddings.KGE* method), 55  
prepare\_data() (*dicee.CVDataModule* method), 223  
prepare\_data() (*dicee.dataset\_classes.CVDataModule* method), 43  
preprocess\_with\_byte\_pair\_encoding() (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG* method), 164  
preprocess\_with\_byte\_pair\_encoding() (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG* method), 159  
preprocess\_with\_byte\_pair\_encoding\_with\_padding() (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG* method), 165  
preprocess\_with\_byte\_pair\_encoding\_with\_padding() (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG* method), 159  
preprocess\_with\_pandas() (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG* method), 165  
preprocess\_with\_pandas() (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG* method), 159  
preprocess\_with\_polars() (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG* method), 165  
preprocess\_with\_polars() (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG* method), 159  
preprocesses\_input\_args() (in module *dicee.static\_preprocess\_funcs*), 172  
PreprocessKG (class in *dicee.read\_preprocess\_save\_load\_kg*), 164  
PreprocessKG (class in *dicee.read\_preprocess\_save\_load\_kg.preprocess*), 159  
PrintCallback (class in *dicee.callbacks*), 21  
process (*dicee.trainer.torch\_trainer.TorchTrainer* attribute), 176  
PseudoLabellingCallback (class in *dicee.callbacks*), 24  
Pyke (class in *dicee*), 182  
Pyke (class in *dicee.models*), 124  
Pyke (class in *dicee.models.real*), 100  
pykeen\_model\_kwargs (*dicee.config.Namespace* attribute), 31

PykeenKGE (class in dicee), 200  
PykeenKGE (class in dicee.models), 150  
PykeenKGE (class in dicee.models.pykeen\_models), 95

## Q

q (dicee.config.Namespace attribute), 31  
q (dicee.DeCaL attribute), 187  
q (dicee.Keci attribute), 184  
q (dicee.models.clifford.DeCaL attribute), 79  
q (dicee.models.clifford.Keci attribute), 76  
q (dicee.models.DeCaL attribute), 145  
q (dicee.models.Keci attribute), 141  
qdrant\_client (dicee.scripts.index\_serve.NeuralSearcher attribute), 167  
QMult (class in dicee), 196  
QMult (class in dicee.models), 132  
QMult (class in dicee.models.quaternion), 96  
quaternion\_mul() (in module dicee.models), 129  
quaternion\_mul() (in module dicee.models.static\_funcs), 100  
quaternion\_mul\_with\_unit\_norm() (in module dicee.models), 132  
quaternion\_mul\_with\_unit\_norm() (in module dicee.models.quaternion), 96  
quaternion\_multiplication\_followed\_by\_inner\_product() (dicee.models.QMult method), 133  
quaternion\_multiplication\_followed\_by\_inner\_product() (dicee.models.quaternion.QMult method), 97  
quaternion\_multiplication\_followed\_by\_inner\_product() (dicee.QMult method), 197  
quaternion\_normalizer() (dicee.models.QMult static method), 133  
quaternion\_normalizer() (dicee.models.quaternion.QMult static method), 97  
quaternion\_normalizer() (dicee.QMult static method), 197  
queries (dicee.scripts.index\_serve.StringListRequest attribute), 167  
query\_name\_to\_struct (dicee.query\_generator.QueryGenerator attribute), 158  
query\_name\_to\_struct (dicee.QueryGenerator attribute), 226  
QueryGenerator (class in dicee), 226  
QueryGenerator (class in dicee.query\_generator), 158

## R

r (dicee.DeCaL attribute), 187  
r (dicee.Keci attribute), 184  
r (dicee.models.clifford.DeCaL attribute), 79  
r (dicee.models.clifford.Keci attribute), 76  
r (dicee.models.DeCaL attribute), 145  
r (dicee.models.Keci attribute), 141  
random\_prediction() (in module dicee), 207  
random\_prediction() (in module dicee.static\_funcs), 170  
random\_seed (dicee.config.Namespace attribute), 31  
ratio (dicee.callbacks.Perturb attribute), 27  
re (dicee.DeCaL attribute), 187  
re (dicee.models.clifford.DeCaL attribute), 79  
re (dicee.models.DeCaL attribute), 145  
re\_vocab (dicee.evaluator.Evaluator attribute), 48  
read\_from\_disk() (in module dicee.read\_preprocess\_save\_load\_kg.util), 163  
read\_from\_triple\_store\_with\_pandas() (in module dicee.read\_preprocess\_save\_load\_kg.util), 163  
read\_from\_triple\_store\_with\_polars() (in module dicee.read\_preprocess\_save\_load\_kg.util), 163  
read\_only\_few (dicee.config.Namespace attribute), 31  
read\_only\_few (dicee.knowledge\_graph.KG attribute), 52  
read\_or\_load\_kg() (in module dicee), 207  
read\_or\_load\_kg() (in module dicee.static\_funcs), 170  
read\_with\_pandas() (in module dicee.read\_preprocess\_save\_load\_kg.util), 163  
read\_with\_polars() (in module dicee.read\_preprocess\_save\_load\_kg.util), 163  
ReadFromDisk (class in dicee.read\_preprocess\_save\_load\_kg), 165  
ReadFromDisk (class in dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk), 160  
reducer (dicee.scripts.index\_serve.StringListRequest attribute), 167  
rel2id (dicee.query\_generator.QueryGenerator attribute), 158  
rel2id (dicee.QueryGenerator attribute), 226  
relation\_embeddings (dicee.AConvQ attribute), 193  
relation\_embeddings (dicee.ConvQ attribute), 194  
relation\_embeddings (dicee.DeCaL attribute), 187  
relation\_embeddings (dicee.DualE attribute), 190  
relation\_embeddings (dicee.LFMult attribute), 199  
relation\_embeddings (dicee.models.AConvQ attribute), 134

- `relation_embeddings` (*dicee.models.clifford.DeCaL* attribute), 79
- `relation_embeddings` (*dicee.models.ConvQ* attribute), 134
- `relation_embeddings` (*dicee.models.DeCaL* attribute), 145
- `relation_embeddings` (*dicee.models.DualE* attribute), 157
- `relation_embeddings` (*dicee.models.dualE.DualE* attribute), 85
- `relation_embeddings` (*dicee.models.FMult* attribute), 153
- `relation_embeddings` (*dicee.models.FMult2* attribute), 155
- `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.LFMult1* attribute), 88
- `relation_embeddings` (*dicee.models.GFMult* attribute), 154
- `relation_embeddings` (*dicee.models.LFMult* attribute), 155
- `relation_embeddings` (*dicee.models.LFMult1* attribute), 155
- `relation_embeddings` (*dicee.models.pykeen\_models.PykeenKGE* attribute), 95
- `relation_embeddings` (*dicee.models.PykeenKGE* attribute), 150
- `relation_embeddings` (*dicee.models.quaternion.AConvQ* attribute), 98
- `relation_embeddings` (*dicee.models.quaternion.ConvQ* attribute), 98
- `relation_embeddings` (*dicee.PykeenKGE* attribute), 200
- `relation_to_idx` (*dicee.knowledge\_graph.KG* attribute), 52
- `relations_str` (*dicee.knowledge\_graph.KG* property), 53
- `reload_dataset()` (in module *dicee*), 213
- `reload_dataset()` (in module *dicee.dataset\_classes*), 33
- `report` (*dicee.DICE\_Trainer* attribute), 208
- `report` (*dicee.evaluator.Evaluator* attribute), 48
- `report` (*dicee.executer.Execute* attribute), 50
- `report` (*dicee.trainer.DICE\_Trainer* attribute), 178
- `report` (*dicee.trainer.dice\_trainer.DICE\_Trainer* attribute), 173
- `reports` (*dicee.callbacks.Eval* attribute), 25
- `reports` (*dicee.callbacks.PeriodicEvalCallback* attribute), 27
- `requires_grad_for_interactions` (*dicee.CKeci* attribute), 183
- `requires_grad_for_interactions` (*dicee.Keci* attribute), 184
- `requires_grad_for_interactions` (*dicee.models.CKeci* attribute), 144
- `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), 141
- `resid_dropout` (*dicee.models.transformers.CausalSelfAttention* attribute), 104
- `residual_convolution()` (*dicee.AConEx* method), 192
- `residual_convolution()` (*dicee.AConvO* method), 193
- `residual_convolution()` (*dicee.AConvQ* method), 193
- `residual_convolution()` (*dicee.ConEx* method), 196
- `residual_convolution()` (*dicee.ConvO* method), 195
- `residual_convolution()` (*dicee.ConvQ* method), 194
- `residual_convolution()` (*dicee.models.AConEx* method), 127
- `residual_convolution()` (*dicee.models.AConvO* method), 140
- `residual_convolution()` (*dicee.models.AConvQ* method), 134
- `residual_convolution()` (*dicee.models.complex.AConEx* method), 83
- `residual_convolution()` (*dicee.models.complex.ConEx* method), 82
- `residual_convolution()` (*dicee.models.ConEx* method), 127
- `residual_convolution()` (*dicee.models.ConvO* method), 140
- `residual_convolution()` (*dicee.models.ConvQ* method), 134
- `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), 167
- `retrieve_embeddings()` (in module *dicee.scripts.index\_serve*), 167
- `return_multi_hop_query_results()` (*dicee.KGE* method), 211
- `return_multi_hop_query_results()` (*dicee.knowledge\_graph\_embeddings.KGE* method), 55
- `root()` (in module *dicee.scripts.index\_serve*), 167
- `roots` (*dicee.models.FMult* attribute), 154
- `roots` (*dicee.models.function\_space.FMult* attribute), 87
- `roots` (*dicee.models.function\_space.GFMult* attribute), 87
- `roots` (*dicee.models.GFMult* attribute), 154
- `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), 224, 225  
`sanity_check_callback_args()` (in module *dicee.sanity\_checkers*), 166  
`sanity_checking_with_arguments()` (in module *dicee.sanity\_checkers*), 166  
`save()` (*dicee.abstracts.BaseInteractiveKGE* method), 15  
`save()` (*dicee.read\_preprocess\_save\_load\_kg.LoadSaveToDisk* method), 165  
`save()` (*dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk.LoadSaveToDisk* method), 160  
`save_checkpoint()` (*dicee.abstracts.AbstractTrainer* static method), 14  
`save_checkpoint_model()` (in module *dicee*), 207  
`save_checkpoint_model()` (in module *dicee.static\_funcs*), 170  
`save_embeddings()` (in module *dicee*), 207  
`save_embeddings()` (in module *dicee.static\_funcs*), 170  
`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*), 207  
`save_numpy_ndarray()` (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 164  
`save_numpy_ndarray()` (in module *dicee.static\_funcs*), 170  
`save_pickle()` (in module *dicee*), 206  
`save_pickle()` (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 164  
`save_pickle()` (in module *dicee.static\_funcs*), 170  
`save_queries()` (*dicee.query\_generator.QueryGenerator* method), 159  
`save_queries()` (*dicee.QueryGenerator* method), 227  
`save_queries_and_answers()` (*dicee.query\_generator.QueryGenerator* static method), 159  
`save_queries_and_answers()` (*dicee.QueryGenerator* static method), 227  
`save_trained_model()` (*dicee.executer.Execute* method), 50  
`scalar_batch_NN()` (*dicee.LFMMult* method), 199  
`scalar_batch_NN()` (*dicee.models.function\_space.LFMMult* method), 89  
`scalar_batch_NN()` (*dicee.models.LFMMult* method), 156  
`scaler` (*dicee.callbacks.Perturb* attribute), 27  
`scaler` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 178  
`scheduler` (*dicee.callbacks.LRScheduler* attribute), 28  
`score()` (*dicee.ComplEx* static method), 192  
`score()` (*dicee.DistMult* method), 183  
`score()` (*dicee.Keci* method), 186  
`score()` (*dicee.models.clifford.Keci* method), 78  
`score()` (*dicee.models.ComplEx* static method), 128  
`score()` (*dicee.models.complex.ComplEx* static method), 84  
`score()` (*dicee.models.DistMult* method), 123  
`score()` (*dicee.models.Keci* method), 143  
`score()` (*dicee.models.octonion.OMult* method), 92  
`score()` (*dicee.models.OMult* method), 139  
`score()` (*dicee.models.QMult* method), 133  
`score()` (*dicee.models.quaternion.QMult* method), 97  
`score()` (*dicee.models.real.DistMult* method), 99  
`score()` (*dicee.models.real.TransE* method), 99  
`score()` (*dicee.models.TransE* method), 123  
`score()` (*dicee.OMult* method), 198  
`score()` (*dicee.QMult* method), 197  
`score()` (*dicee.TransE* method), 186  
`score_func` (*dicee.models.FMMult2* attribute), 154  
`score_func` (*dicee.models.function\_space.FMMult2* attribute), 88  
`scoring_technique` (*dicee.analyse\_experiments.Experiment* attribute), 20  
`scoring_technique` (*dicee.config.Namespace* attribute), 30  
`search()` (*dicee.scripts.index\_serve.NeuralSearcher* method), 167  
`search_embeddings()` (in module *dicee.scripts.index\_serve*), 167  
`search_embeddings_batch()` (in module *dicee.scripts.index\_serve*), 167  
`seed` (*dicee.query\_generator.QueryGenerator* attribute), 158  
`seed` (*dicee.QueryGenerator* attribute), 226  
`select_model()` (in module *dicee*), 206

`select_model()` (in module *dicее.static\_funcs*), 170  
`selected_optimizer` (*dicее.BaseKGE* attribute), 204  
`selected_optimizer` (*dicее.models.base\_model.BaseKGE* attribute), 73  
`selected_optimizer` (*dicее.models.BaseKGE* attribute), 118, 121, 125, 130, 136, 149, 152  
`separator` (*dicее.config.Namespace* attribute), 30  
`separator` (*dicее.knowledge\_graph.KG* attribute), 53  
`sequential_vocabulary_construction()` (*dicее.read\_preprocess\_save\_load\_kg.PreprocessKG* method), 165  
`sequential_vocabulary_construction()` (*dicее.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG* method), 159  
`serve()` (in module *dicее.scripts.index\_serve*), 168  
`set_global_seed()` (*dicее.query\_generator.QueryGenerator* method), 158  
`set_global_seed()` (*dicее.QueryGenerator* method), 226  
`set_model_eval_mode()` (*dicее.abstracts.BaseInteractiveKGE* method), 15  
`set_model_train_mode()` (*dicее.abstracts.BaseInteractiveKGE* method), 14  
`setup()` (*dicее.CVDDataModule* method), 222  
`setup()` (*dicее.dataset\_classes.CVDDataModule* method), 42  
`setup_executor()` (*dicее.executor.Execute* method), 50  
`Shallom` (class in *dicее*), 198  
`Shallom` (class in *dicее.models*), 123  
`Shallom` (class in *dicее.models.real*), 100  
`shallom` (*dicее.models.real.Shallom* attribute), 100  
`shallom` (*dicее.models.Shallom* attribute), 124  
`shallom` (*dicее.Shallom* attribute), 198  
`single_hop_query_answering()` (*dicее.KGE* method), 211  
`single_hop_query_answering()` (*dicее.knowledge\_graph\_embeddings.KGE* method), 55  
`snapshot_dir` (*dicее.callbacks.LRScheduler* attribute), 28  
`snapshot_loss` (*dicее.callbacks.LRScheduler* attribute), 28  
`sparql_endpoint` (*dicее.config.Namespace* attribute), 30  
`sparql_endpoint` (*dicее.knowledge\_graph.KG* attribute), 52  
`start()` (*dicее.DICE\_Trainer* method), 209  
`start()` (*dicее.executor.Execute* method), 51  
`start()` (*dicее.read\_preprocess\_save\_load\_kg.PreprocessKG* method), 164  
`start()` (*dicее.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG* method), 159  
`start()` (*dicее.read\_preprocess\_save\_load\_kg.read\_from\_disk.ReadFromDisk* method), 160  
`start()` (*dicее.read\_preprocess\_save\_load\_kg.ReadFromDisk* method), 165  
`start()` (*dicее.trainer.DICE\_Trainer* method), 179  
`start()` (*dicее.trainer.dice\_trainer.DICE\_Trainer* method), 174  
`start_time` (*dicее.callbacks.PrintCallback* attribute), 22  
`start_time` (*dicее.executor.Execute* attribute), 50  
`state_dict()` (*dicее.EnsembleKGE* method), 206  
`state_dict()` (*dicее.models.ensemble.EnsembleKGE* method), 86  
`step()` (*dicее.EnsembleKGE* method), 206  
`step()` (*dicее.models.ADOPT* method), 110  
`step()` (*dicее.models.adopt.ADOPT* method), 60  
`step()` (*dicее.models.ensemble.EnsembleKGE* method), 86  
`step_count` (*dicее.callbacks.LRScheduler* attribute), 28  
`storage_path` (*dicее.config.Namespace* attribute), 29  
`storage_path` (*dicее.DICE\_Trainer* attribute), 208  
`storage_path` (*dicее.trainer.DICE\_Trainer* attribute), 178  
`storage_path` (*dicее.trainer.dice\_trainer.DICE\_Trainer* attribute), 174  
`store()` (in module *dicее*), 207  
`store()` (in module *dicее.static\_funcs*), 170  
`store_ensemble()` (*dicее.abstracts.AbstractPPECallback* method), 18  
`strategy` (*dicее.abstracts.AbstractTrainer* attribute), 13  
`StringListRequest` (class in *dicее.scripts.index\_serve*), 167  
`SWA` (class in *dicее.callbacks*), 28  
`swa` (*dicее.config.Namespace* attribute), 32  
`swa_c_epochs` (*dicее.callbacks.SWA* attribute), 29  
`swa_lr` (*dicее.callbacks.SWA* attribute), 29  
`swa_model` (*dicее.callbacks.SWA* attribute), 29  
`swa_n` (*dicее.callbacks.SWA* attribute), 29  
`swa_start_epoch` (*dicее.callbacks.SWA* attribute), 29  
`swa_start_epoch` (*dicее.config.Namespace* attribute), 32

## T

`T()` (*dicее.DualE* method), 191  
`T()` (*dicее.models.DualE* method), 157  
`T()` (*dicее.models.dualE.DualE* method), 85

`t_conorm()` (*dicee.abstracts.InteractiveQueryDecomposition method*), 16  
`t_norm()` (*dicee.abstracts.InteractiveQueryDecomposition method*), 16  
`target_dim` (*dicee.AllvsAll attribute*), 216  
`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*), 214  
`target_dim` (*dicee.OnevsAllDataset attribute*), 215  
`temperature` (*dicee.BytE attribute*), 202  
`temperature` (*dicee.models.transformers.BytE attribute*), 101  
`tensor_t_norm()` (*dicee.abstracts.InteractiveQueryDecomposition method*), 16  
`TensorParallel` (*class in dicee.trainer.model\_parallelism*), 175  
`test_dataloader()` (*dicee.models.base\_model.BaseKGELightning method*), 68  
`test_dataloader()` (*dicee.models.BaseKGELightning method*), 114  
`test_epoch_end()` (*dicee.models.base\_model.BaseKGELightning method*), 68  
`test_epoch_end()` (*dicee.models.BaseKGELightning method*), 114  
`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*), 158  
`test_path` (*dicee.QueryGenerator attribute*), 226  
`timeit()` (*in module dicee*), 206, 213  
`timeit()` (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 163  
`timeit()` (*in module dicee.static\_funcs*), 169  
`timeit()` (*in module dicee.static\_preprocess\_funcs*), 172  
`to()` (*dicee.EnsembleKGE method*), 206  
`to()` (*dicee.KGE method*), 209  
`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*), 202  
`topk` (*dicee.models.transformers.BytE attribute*), 102  
`topk` (*dicee.scripts.index\_serve.NeuralSearcher attribute*), 167  
`torch_ordered_shaped_bpe_entities` (*dicee.dataset\_classes.MultiLabelDataset attribute*), 34  
`torch_ordered_shaped_bpe_entities` (*dicee.MultiLabelDataset attribute*), 214  
`TorchDDPTrainer` (*class in dicee.trainer.torch\_trainer\_ddp*), 177  
`TorchTrainer` (*class in dicee.trainer.torch\_trainer*), 176  
`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*), 178  
`train_data` (*dicee.AllvsAll attribute*), 216  
`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*), 216  
`train_data` (*dicee.KvsSampleDataset attribute*), 219  
`train_data` (*dicee.MultiClassClassificationDataset attribute*), 214  
`train_data` (*dicee.OnevsAllDataset attribute*), 215  
`train_data` (*dicee.OnevsSample attribute*), 217, 218  
`train_dataloader()` (*dicee.CVDDataModule method*), 221  
`train_dataloader()` (*dicee.dataset\_classes.CVDDataModule method*), 41  
`train_dataloader()` (*dicee.models.base\_model.BaseKGELightning method*), 69  
`train_dataloader()` (*dicee.models.BaseKGELightning method*), 115  
`train_dataloaders` (*dicee.trainer.torch\_trainer.TorchTrainer attribute*), 176  
`train_dataset_loader` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 178  
`train_file_path` (*dicee.dataset\_classes.LiteralDataset attribute*), 45  
`train_file_path` (*dicee.LiteralDataset attribute*), 224, 225  
`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*), 214

`train_k_vs_all()` (*dicee.abstracts.BaseInteractiveTrainKGE method*), 18  
`train_literals()` (*dicee.abstracts.BaseInteractiveTrainKGE method*), 18  
`train_mode` (*dicee.EnsembleKGE attribute*), 205  
`train_mode` (*dicee.models.ensemble.EnsembleKGE attribute*), 86  
`train_mrr` (*dicee.analyse\_experiments.Experiment attribute*), 20  
`train_path` (*dicee.query\_generator.QueryGenerator attribute*), 158  
`train_path` (*dicee.QueryGenerator attribute*), 226  
`train_set` (*dicee.BPE\_NegativeSamplingDataset attribute*), 213  
`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*), 214  
`train_set` (*dicee.NegSampleDataset attribute*), 219  
`train_set` (*dicee.TriplePredictionDataset attribute*), 220  
`train_set_idx` (*dicee.CVDDataModule attribute*), 221  
`train_set_idx` (*dicee.dataset\_classes.CVDDataModule attribute*), 41  
`train_set_target` (*dicee.knowledge\_graph.KG attribute*), 52  
`train_target` (*dicee.AllvsAll attribute*), 216  
`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*), 216  
`train_target` (*dicee.KvsSampleDataset attribute*), 219  
`train_target_indices` (*dicee.knowledge\_graph.KG attribute*), 53  
`train_triples` (*dicee.dataset\_classes.NegSampleDataset attribute*), 40  
`train_triples` (*dicee.NegSampleDataset attribute*), 219  
`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*), 208  
`trainer` (*dicee.executer.Execute attribute*), 50  
`trainer` (*dicee.trainer.DICE\_Trainer attribute*), 178  
`trainer` (*dicee.trainer.dice\_trainer.DICE\_Trainer attribute*), 173  
`trainer` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 177  
`training_step` (*dicee.trainer.torch\_trainer.TorchTrainer attribute*), 176  
`training_step()` (*dicee.BytE method*), 202  
`training_step()` (*dicee.models.base\_model.BaseKGELightning method*), 66  
`training_step()` (*dicee.models.BaseKGELightning method*), 112  
`training_step()` (*dicee.models.transformers.BytE method*), 102  
`training_step_outputs` (*dicee.models.base\_model.BaseKGELightning attribute*), 66  
`training_step_outputs` (*dicee.models.BaseKGELightning attribute*), 112  
`training_technique` (*dicee.knowledge\_graph.KG attribute*), 52  
`TransE` (*class in dicee*), 186  
`TransE` (*class in dicee.models*), 123  
`TransE` (*class in dicee.models.real*), 99  
`transfer_batch_to_device()` (*dicee.CVDDataModule method*), 222  
`transfer_batch_to_device()` (*dicee.dataset\_classes.CVDDataModule method*), 42  
`transformer` (*dicee.BytE attribute*), 202  
`transformer` (*dicee.models.transformers.BytE attribute*), 102  
`transformer` (*dicee.models.transformers.GPT attribute*), 107  
`trapezoid()` (*dicee.models.FMult2 method*), 155  
`trapezoid()` (*dicee.models.function\_space.FMult2 method*), 88  
`tri_score()` (*dicee.LFMult method*), 199  
`tri_score()` (*dicee.models.function\_space.LFMult method*), 89  
`tri_score()` (*dicee.models.function\_space.LFMult1 method*), 88  
`tri_score()` (*dicee.models.LFMult method*), 156  
`tri_score()` (*dicee.models.LFMult1 method*), 155  
`triple_score()` (*dicee.KGE method*), 211  
`triple_score()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 55  
`TriplePredictionDataset` (*class in dicee*), 220  
`TriplePredictionDataset` (*class in dicee.dataset\_classes*), 40  
`tuple2list()` (*dicee.query\_generator.QueryGenerator method*), 158  
`tuple2list()` (*dicee.QueryGenerator method*), 226

## U

`unlabelled_size` (*dicee.callbacks.PseudoLabellingCallback attribute*), 24

`unmap()` (*dicee.query\_generator.QueryGenerator method*), 158  
`unmap()` (*dicee.QueryGenerator method*), 227  
`unmap_query()` (*dicee.query\_generator.QueryGenerator method*), 158  
`unmap_query()` (*dicee.QueryGenerator method*), 227

## V

`val_aswa` (*dicee.callbacks.ASWA attribute*), 24  
`val_data_loader()` (*dicee.models.base\_model.BaseKGELighting method*), 69  
`val_data_loader()` (*dicee.models.BaseKGELighting method*), 114  
`val_h1` (*dicee.analyse\_experiments.Experiment attribute*), 20  
`val_h3` (*dicee.analyse\_experiments.Experiment attribute*), 20  
`val_h10` (*dicee.analyse\_experiments.Experiment attribute*), 20  
`val_mrr` (*dicee.analyse\_experiments.Experiment attribute*), 20  
`val_path` (*dicee.query\_generator.QueryGenerator attribute*), 158  
`val_path` (*dicee.QueryGenerator attribute*), 226  
`validate_knowledge_graph()` (*in module dicee.sanity\_checkers*), 166  
`vocab_preparation()` (*dicee.evaluator.Evaluator method*), 49  
`vocab_size` (*dicee.models.transformers.GPTConfig attribute*), 106  
`vocab_to_parquet()` (*in module dicee*), 207  
`vocab_to_parquet()` (*in module dicee.static\_funcs*), 171  
`vtp_score()` (*dicee.LFMult method*), 199  
`vtp_score()` (*dicee.models.function\_space.LFMult method*), 89  
`vtp_score()` (*dicee.models.function\_space.LFMult1 method*), 88  
`vtp_score()` (*dicee.models.LFMult method*), 156  
`vtp_score()` (*dicee.models.LFMult1 method*), 155

## W

`warmup_steps` (*dicee.callbacks.LRScheduler attribute*), 28  
`weight` (*dicee.models.transformers.LayerNorm attribute*), 103  
`weight_decay` (*dicee.BaseKGE attribute*), 204  
`weight_decay` (*dicee.config.Namespace attribute*), 30  
`weight_decay` (*dicee.models.base\_model.BaseKGE attribute*), 72  
`weight_decay` (*dicee.models.BaseKGE attribute*), 118, 121, 125, 130, 136, 149, 152  
`weights` (*dicee.models.FMult attribute*), 154  
`weights` (*dicee.models.function\_space.FMult attribute*), 87  
`weights` (*dicee.models.function\_space.GFMult attribute*), 87  
`weights` (*dicee.models.GFMult attribute*), 154  
`write_csv_from_model_parallel()` (*in module dicee*), 208  
`write_csv_from_model_parallel()` (*in module dicee.static\_funcs*), 171  
`write_links()` (*dicee.query\_generator.QueryGenerator method*), 158  
`write_links()` (*dicee.QueryGenerator method*), 226  
`write_report()` (*dicee.executer.Execute method*), 51

## X

`x_values` (*dicee.LFMult attribute*), 199  
`x_values` (*dicee.models.function\_space.LFMult attribute*), 89  
`x_values` (*dicee.models.LFMult attribute*), 155