

---

# DICE Embeddings

*Release 0.1.3.2*

**Caglar Demir**

**Nov 13, 2024**

**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 . . . . .	159
14.3	Classes . . . . .	159
14.4	Functions . . . . .	160
14.5	Package Contents . . . . .	162
	<b>Python Module Index</b>	<b>206</b>

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

## 1 Dicee Manual

**Version:** dicee 0.1.3.2

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

**Publisher and maintainer:** Caglar 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-CPU, 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.executer import Execute
from dicee.config import Namespace
from dicee.knowledge_graph_embeddings import KGE
# (1) Train a KGE model
args = Namespace()
args.model = 'Keci'
args.p=0
args.q=1
args.optim = 'Adam'
args.scoring_technique = "AllvsAll"
args.path_single_kg = "KGs/Family/family-benchmark_rich_background.owl"
args.backend = "rdflib"
args.num_epochs = 200
args.batch_size = 1024
args.lr = 0.1
args.embedding_dim = 512
result = Execute(args).start()
# (2) Load the pre-trained model
```

(continues on next page)

(continued from previous page)

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

For more, please refer to examples/multi\_hop\_query\_answering.

## 8 Predicting Missing Links

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

## 9 Downloading Pretrained Models

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

- For more please look at [dice-research.org/projects/DiceEmbeddings/](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	201	82	59%	104-105, 107-108, 110-111, 113-114, 116-117, 119-120, 122-123, 125-126, 128-129, 131-132, 134-135, 137-138, 140-141, 143-144, 146-147, 149-150, 152-153, 155-156, 158-159, 161-162, 164-165, 167-168, 170-171, 173-174, 176-177, 179-180, 182-183, 185-186, 188-189, 191-192, 194-195, 197-198, 200-201, 203-204, 206-207, 209-210, 212-213, 215-216, 218-219, 221-222, 224-225, 227-228, 230-231, 233-234, 236-237, 239-240, 242-243, 245-246, 248-249, 251-252, 254-255, 257-258, 260-261, 263-264, 266-267, 269-270, 272-273, 275-276, 278-279, 281-282, 284-285, 287-288, 290-291, 293-294, 296-297, 299-300, 302-303, 305-306, 308-309, 311-312, 314-315, 317-318, 320-321, 323-324, 326-327, 329-330, 332-333, 335-336, 338-339, 341-342, 344-345, 347-348, 350-351, 353-354, 356-357, 359-360, 362-363, 365-366, 368-369, 371-372, 374-375, 377-378, 380-381, 383-384, 386-387, 389-390, 392-393, 395-396, 398-399, 401-402, 404-405, 407-408, 410-411, 413-414, 416-417, 419-420, 422-423, 425-426, 428-429, 431-432, 434-435, 437-438, 440-441, 443-444, 446-447, 449-450, 452-453, 455-456, 458-459, 461-462, 464-465, 467-468, 470-471, 473-474, 476-477, 479-480, 482-483, 485-486, 488-489, 491-492, 494-495, 497-498, 500-501, 503-504, 506-507, 509-510, 512-513, 515-516, 518-519, 521-522, 524-525, 527-528, 530-531, 533-534, 536-537, 539-540, 542-543, 545-546, 548-549, 551-552, 554-555, 557-558, 560-561, 563-564, 566-567, 569-570, 572-573, 575-576, 578-579, 581-582, 584-585, 587-588, 590-591, 593-594, 596-597, 599-600, 602-603, 605-606, 608-609, 611-612, 614-615, 617-618, 620-621, 623-624, 626-627, 629-630, 632-633, 635-636, 638-639, 641-642, 644-645, 647-648, 650-651, 653-654, 656-657, 659-660, 662-663, 665-666, 668-669, 671-672, 674-675, 677-678, 680-681, 683-684, 686-687, 689-690, 692-693, 695-696, 698-699, 701-702, 704-705, 707-708, 710-711, 713-714, 716-717, 719-720, 722-723, 725-726, 728-729, 731-732, 734-735, 737-738, 740-741, 743-744, 746-747, 749-750, 752-753, 755-756, 758-759, 761-762, 764-765, 767-768, 770-771, 773-774, 776-777, 779-780, 782-783, 785-786, 788-789, 791-792, 794-795, 797-798, 800-801, 803-804, 806-807, 809-810, 812-813, 815-816, 818-819, 821-822, 824-825, 827-828, 830-831, 833-834, 836-837, 839-840, 842-843, 845-846, 848-849, 851-852, 854-855, 857-858, 860-861, 863-864, 866-867, 869-870, 872-873, 875-876, 878-879, 881-882, 884-885, 887-888, 890-891, 893-894, 896-897, 899-900, 902-903, 905-906, 908-909, 911-912, 914-915, 917-918, 920-921, 923-924, 926-927, 929-930, 932-933, 935-936, 938-939, 941-942, 944-945, 947-948, 950-951, 953-954, 956-957, 959-960, 962-963, 965-966, 968-969, 971-972, 974-975, 977-978, 980-981, 983-984, 986-987, 989-990, 992-993, 995-996, 998-999

(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)

```
→123, 146-147, 152, 165, 197, 240-254, 257-260, 263-266, 301, 314-317, 320-324, 364-  
→375, 390-398, 413, 424-428, 555-575, 581-585, 589-591  
dicee/callbacks.py 245 102 58% 50-55,   
→67-73, 76, 88-93, 98-103, 106-109, 116-133, 138-142, 146-147, 276-280, 286-287, 305-  
→311, 314, 319-320, 332-338, 344-353, 358-360, 405, 416-429, 433-468, 480-486  
dicee/config.py 93 2 98% 141-142  
dicee/dataset_classes.py 299 74 75% 41, 54,   
→87, 93, 99-106, 109, 112, 115-139, 195-201, 204, 207-209, 314, 325-328, 344, 410-  
→411, 429, 528-536, 539, 543-557, 700-707, 710-714  
dicee/eval_static_funcs.py 227 95 58% 101, 106,  
→ 111, 258-353, 360-411  
dicee/evaluator.py 262 51 81% 46, 51,   
→56, 84, 89-90, 93, 109, 126, 137, 141, 146, 177-188, 195-206, 314, 344-367, 455,   
→465, 482-487  
dicee/executer.py 113 4 96% 116, 258-  
→259, 291  
dicee/knowledge_graph.py 65 3 95% 79, 110,   
→114  
dicee/knowledge_graph_embeddings.py 636 443 30% 27, 30-  
→31, 39-52, 57-90, 93-127, 131-139, 170-184, 215-228, 254-274, 324-327, 330-333, 346,  
→ 381-426, 484-486, 502-503, 509-517, 522-525, 528-533, 538, 547, 592-598, 630, 688-  
→1053, 1084-1145, 1149-1177, 1200, 1227-1265  
dicee/models/__init__.py 9 0 100%  
dicee/models/base_model.py 234 31 87% 54, 56,   
→82, 88-103, 157, 190, 230, 236, 245, 248, 252, 259, 263, 265, 280, 288-289, 296-297,  
→ 351, 354, 427, 439  
dicee/models/clifford.py 556 357 36% 31-42,   
→68-117, 122-133, 156-168, 190-220, 235, 237, 241, 248-249, 276-280, 303-311, 325-  
→327, 332-333, 364-384, 406, 413, 417-478, 495-499, 511, 514, 519, 524, 571-607, 625-  
→631, 644, 647, 652, 657, 686-692, 705, 708, 713, 718, 728-737, 753-754, 774-845,   
→856-859, 884-909, 933-966, 1002-1006, 1019, 1029, 1032, 1037, 1042, 1047, 1051,   
→1055, 1064-1065, 1095, 1102, 1107, 1135-1139, 1167-1176, 1186-1194, 1212-1214, 1232-  
→1234, 1250-1252  
dicee/models/complex.py 151 15 90% 86-109  
dicee/models/dualE.py 59 10 83% 93-102,   
→142-156  
dicee/models/function_space.py 262 221 16% 10-24,   
→28-37, 40-49, 53-70, 77-86, 89-98, 101-110, 114-126, 134-156, 159-165, 168-185, 188-  
→194, 197-205, 208, 213-234, 243-246, 250-254, 258-267, 271-292, 301-307, 311-328,   
→332-335, 344-352, 355, 366-372, 392-406, 424-438, 443-453, 461-465, 474-478  
dicee/models/octonion.py 227 83 63% 21-44,   
→320-329, 334-345, 348-370, 374-416, 426-474  
dicee/models/pykeen_models.py 50 5 90% 60-63,   
→118  
dicee/models/quaternion.py 192 69 64% 7-21, 30-  
→55, 68-72, 107, 185, 328-342, 345-364, 368-389, 399-426  
dicee/models/real.py 61 12 80% 36-39,   
→66-69, 87, 103-106  
dicee/models/static_funcs.py 10 0 100%  
dicee/models/transformers.py 236 189 20% 24-43,   
→46, 60-75, 84-102, 105-116, 123-125, 128, 134-151, 155-180, 186-190, 193-197, 203-  
→207, 210-212, 229-256, 265-268, 271-276, 279-304, 310-315, 319-372, 376-398, 404-414
```

(continues on next page)

(continued from previous page)

dicee/query_generator.py	374	346	7%	18-52, ↪
↪56, 62-65, 69-70, 78-92, 100-147, 155-188, 192-206, 212-269, 274-303, 307-443, 453-↪472, 480-501, 508-512, 517, 522-528				
dicee/read_preprocess_save_load_kg/___init___py	3	0	100%	
dicee/read_preprocess_save_load_kg/preprocess.py	256	41	84%	34, 40, ↪
↪78, 102-127, 133, 138-151, 184, 214, 388-389, 444				
dicee/read_preprocess_save_load_kg/read_from_disk.py	36	11	69%	33, 38-↪
↪40, 47, 55, 58-72				
dicee/read_preprocess_save_load_kg/save_load_disk.py	45	18	60%	39-60
dicee/read_preprocess_save_load_kg/util.py	219	126	42%	65-67, ↪
↪72-73, 91-97, 100-102, 107-109, 121, 134, 140-143, 148-156, 161-167, 172-177, 182-↪187, 199-220, 226-282, 286-290, 294-295, 299, 303-304, 334, 351, 356, 363-364				
dicee/sanity_checkers.py	54	23	57%	8-12, 21-↪
↪31, 46, 51, 58, 64-79, 85, 89, 96				
dicee/static_funcs.py	418	163	61%	40, 50, ↪
↪56-61, 83, 105-106, 115, 138, 152, 157-159, 163-165, 167, 194-198, 246, 254, 263-↪268, 290-304, 316-336, 340-357, 362, 386-387, 392-393, 410-411, 413-414, 416-417, ↪				
↪419-420, 428, 446-450, 467-470, 474-479, 483-487, 491-492, 498-500, 526-527, 539-↪542, 547-550, 559-610, 615-627, 644-658, 661-669				
dicee/static_funcs_training.py	123	63	49%	118-215, ↪
↪223-224				
dicee/static_preprocess_funcs.py	100	44	56%	17-25, ↪
↪52, 56, 64, 67, 78, 91-115, 120-123, 128-131, 136-139				
dicee/trainer/___init___py	1	0	100%	
dicee/trainer/dice_trainer.py	126	13	90%	27-32, ↪
↪91, 98, 103-108, 147				
dicee/trainer/torch_trainer.py	79	4	95%	31, 196, ↪
↪207-208				
dicee/trainer/torch_trainer_ddp.py	152	128	16%	13-14, ↪
↪43, 47-72, 83-112, 131-137, 140-149, 164-194, 204-217, 226-246, 251-260, 263-272, ↪				
↪275-299, 302-309				
-----				
TOTAL	6181	2828	54%	

## 13 How to cite

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

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

(continues on next page)

```

@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}}
# Shallow
@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>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

### 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\_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**

**apply\_semantic\_constraint**

**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.**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

## **dicee.analyse\_experiments**

This script should be moved to dicee/scripts

## 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

### 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

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

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

Bases: `dicee.abstracts.AbstractCallback`

A callback for a three-Level Perturbation

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

Parameter Perturbation:

Output Perturbation:

**level**

**ratio**

**method**

**scaler**

**frequency**

**on\_train\_batch\_start** (*trainer, model, batch, batch\_idx*)

Called when the train batch begins.

## `dicee.config`

## Classes

*Namespace*

Simple object for storing attributes.

## Module Contents

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

Bases: argparse.Namespace

Simple object for storing attributes.

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

```
dataset_dir: str = None
```

The path of a folder containing train.txt, and/or valid.txt and/or test.txt

```
save_embeddings_as_csv: bool = False
```

Embeddings of entities and relations are stored into CSV files to facilitate easy usage.

```
storage_path: str = 'Experiments'
```

A directory named with time of execution under `storage_path` that contains related data about embeddings.

```
path_to_store_single_run: str = None
```

A single directory created that contains related data about embeddings.

```
path_single_kg = None
```

Path of a file corresponding to the input knowledge graph

```
sparql_endpoint = None
```

An endpoint of a triple store.

```
model: str = 'Keci'
```

KGE model

```
optim: str = 'Adam'
```

Optimizer

```
embedding_dim: int = 64
```

Size of continuous vector representation of an entity/relation

```
num_epochs: int = 150
```

Number of pass over the training data

```
batch_size: int = 1024
```

Mini-batch size if it is None, an automatic batch finder technique applied

```
lr: float = 0.1
```

Learning rate

```
add_noise_rate: float = None
```

The ratio of added random triples into training dataset

```
gpus = None
```

Number GPUs to be used during training

```
callbacks
```

```
10}}
```

**Type**

Callbacks, e.g., {“PPE”

**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

**\_\_iter\_\_()**

## dicee.dataset\_classes

### Classes

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

## 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`

`num_of_data_points`

```
collate_fn = None
```

```
__len__()
```

```
__getitem__(idx)
```

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

Bases: torch.utils.data.Dataset

Dataset for the 1vsALL training strategy

#### Parameters

- **train\_set\_idx** – Indexed triples for the training.
- **entity\_idx** – mapping.
- **relation\_idx** – mapping.
- **form** – ?
- **num\_workers** – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

#### Return type

torch.utils.data.Dataset

```
train_data
```

```
target_dim
```

```
collate_fn = None
```

```
__len__()
```

```
__getitem__(idx)
```

```
class dicee.dataset_classes.KvsAll (train_set_idx: numpy.ndarray, entity_idx, relation_idx, form,  
store=None, label_smoothing_rate: float = 0.0)
```

Bases: torch.utils.data.Dataset

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

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

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

#### Note

TODO

**train\_set\_idx**

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

**entity\_idx**

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

**relation\_idx**

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

```
self : torch.utils.data.Dataset
```

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

```
train_data = None
```

```
train_target = None
```

```
label_smoothing_rate
```

```
collate_fn = None
```

```
__len__()
```

```
__getitem__(idx)
```

```
class dicee.dataset_classes.AllvsAll (train_set_idx: numpy.ndarray, entity_idxes, relation_idxes,
    label_smoothing_rate=0.0)
```

```
Bases: torch.utils.data.Dataset
```

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

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

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

**relation\_idxes**

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

```
self : torch.utils.data.Dataset
```

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

```
train_data = None
```

```
train_target = None
```

```
label_smoothing_rate
```



`collate_fn = None`

`target_dim`

`__len__()`

`__getitem__(idx)`

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

Bases: `torch.utils.data.Dataset`

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

#### Parameters

- **train\_set** (*np.ndarray*) – A numpy array containing triples of knowledge graph data. Each triple consists of (head\_entity, relation, tail\_entity).
- **num\_entities** (*int*) – The number of unique entities in the knowledge graph.
- **num\_relations** (*int*) – The number of unique relations in the knowledge graph.
- **neg\_sample\_ratio** (*int, optional*) – The number of negative samples to be generated per positive sample. Must be a positive integer and less than num\_entities.
- **label\_smoothing\_rate** (*float, optional*) – A label smoothing rate to apply to the positive and negative labels. Defaults to 0.0.

**train\_data**

The input data converted into a PyTorch tensor.

#### Type

`torch.Tensor`

**num\_entities**

Number of entities in the dataset.

#### Type

`int`

**num\_relations**

Number of relations in the dataset.

#### Type

`int`

**neg\_sample\_ratio**

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

#### Type

`int`

**label\_smoothing\_rate**

The smoothing factor applied to the labels.

#### Type

`torch.Tensor`

**collate\_fn**

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

**Type**

function, optional

**train\_data**

**num\_entities**

**num\_relations**

**neg\_sample\_ratio**

**label\_smoothing\_rate**

**collate\_fn = None**

**\_\_len\_\_()**

Returns the number of samples in the dataset.

**\_\_getitem\_\_(idx)**

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

**Parameters**

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

**Returns**

**A tuple consisting of:**

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

**Return type**

tuple

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

Bases: torch.utils.data.Dataset

**KvsSample a Dataset:**

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

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

**forall y\_i =1 s.t. (h r E\_i) in KG**

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

**|new\_y| << |E|** new\_y contains all 1's if sum(y)< neg\_sample ratio new\_y contains

**train\_set\_idx**

Indexed triples for the training.

**entity\_idxes**

mapping.

```

    relation_idx
    mapping.

    form
    ?

    store
    ?

    label_smoothing_rate
    ?

    torch.utils.data.Dataset

```

```

train_data = None

train_target = None

neg_ratio

num_entities

label_smoothing_rate

collate_fn = None

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_set

length

num_entities

num_relations

```

```

__len__()

__getitem__(idx)

class dicee.dataset_classes.TriplePredictionDataset (train_set: numpy.ndarray,
    num_entities: int, num_relations: int, neg_sample_ratio: int = 1, label_smoothing_rate: float = 0.0)
Bases: torch.utils.data.Dataset
    Triple Dataset
        D:= {(x)_i}_i ^N, where
            . x:(h,r, t) in KG is a unique h in E and a relation r in R and . collect_fn => Generates
            negative triples
        collect_fn:
orall (h,r,t) in G obtain, create negative triples {(h,r,x),(r,t),(h,m,t)}
        y:labels are represented in torch.float16

    train_set_idx
        Indexed triples for the training.

    entity_idxxs
        mapping.

    relation_idxxs
        mapping.

    form
        ?

    store
        ?

    label_smoothing_rate

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

label_smoothing_rate

neg_sample_ratio

train_set

length

num_entities

num_relations

__len__()

__getitem__(idx)

collate_fn (batch: List[torch.Tensor])

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

```

## Parameters

- **train\_set\_idx** – Indexed triples for the training.
- **num\_entities** – entity to index mapping.
- **num\_relations** – relation to index mapping.
- **batch\_size** – int
- **form** – ?
- **num\_workers** – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

## Return type

?

**train\_set\_idx**

**num\_entities**

**num\_relations**

**neg\_sample\_ratio**

**batch\_size**

**num\_workers**

**train\_dataloader()** → torch.utils.data.DataLoader

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:param-ref:~pytorch\_lightning.trainer.trainer.Trainer.reload\_dataloaders\_every\_n\_epochs`** to a positive integer.

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

### Warning

do not assign state in `prepare_data`

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

### Note

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

**setup**(\*args, \*\*kwargs)

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

#### Parameters

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

Example:

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

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

        # don't do this
        self.something = else

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

**transfer\_batch\_to\_device**(\*args, \*\*kwargs)

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

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

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

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

### Note

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

#### Parameters

- **batch** – A batch of data that needs to be transferred to a new device.

- **device** – The target device as defined in PyTorch.
- **dataloader\_idx** – The index of the dataloader to which the batch belongs.

#### Returns

A reference to the data on the new device.

Example:

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

#### See also

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

#### `prepare_data(*args, **kwargs)`

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

#### Warning

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

Example:

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

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

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

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

Example:

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

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

This is called before requesting the dataloaders:

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

## dicee.eval\_static\_funcs

### Functions

```
evaluate_link_prediction_performance(→
Dict)
evaluate_link_prediction_performance_with_
evaluate_link_prediction_performance_with_
evaluate_link_prediction_performance_with_
...)
evaluate_lp_bpe_k_vs_all(model,      triples[,
er_vocab, ...])
```

### 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.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
```

```
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

**args**

**is\_continual\_training**

**trainer = None**

**trained\_model = None**

**knowledge\_graph = None**

**report**

**evaluator = None**

**start\_time = None**

**setup\_executor()** → None

**dept\_read\_preprocess\_index\_serialize\_data()** → None

Read & Preprocess & Index & Serialize Input Data

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

### Parameter

**rtype**

None

**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

<i>KG</i>	Knowledge Graph
-----------	-----------------

### Module Contents

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

Knowledge Graph

`dataset_dir`

`sparql_endpoint`

`path_single_kg`

`byte_pair_encoding`

`ordered_shaped_bpe_tokens = None`

`add_noise_rate`

`num_entities = None`

`num_relations = None`

`path_for_deserialization`

`add_reciprocal`

`eval_model`

`read_only_few`

`sample_triples_ratio`

`path_for_serialization`

`entity_to_idx`

`relation_to_idx`

`backend`

`training_technique`

`idx_entity_to_bpe_shaped`

`enc`

`num_tokens`

```

num_bpe_entities = None

padding

dummy_id

max_length_subword_tokens = None

train_set_target = None

target_dim = None

train_target_indices = None

ordered_bpe_entities = None

separator

description_of_input = None

describe() → 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
    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*)  
→ 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*)  
→ 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*) → 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)

Predict missing item in a given triple.

## Parameter

head\_entity: Union[str, List[str]]

String representation of selected entities.

relation: Union[str, List[str]]

String representation of selected relations.

tail\_entity: Union[str, List[str]]

String representation of selected entities.

k: int

Highest ranked k item.

## Returns: Tuple

Highest K scores and items

**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

**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

**return\_multi\_hop\_query\_results** (*aggregated\_query\_for\_all\_entities, k: int, only\_scores*)

**single\_hop\_query\_answering** (*query: tuple, only\_scores: bool = True, k: int = None*)

**answer\_multi\_hop\_query** (*query\_type: str = None, query: Tuple[str | Tuple[str, str], Ellipsis] = None, queries: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, tnorm: str = 'prod', neg\_norm: str = 'standard', lambda\_: float = 0.0, k: int = 10, only\_scores=False*) → List[Tuple[str, torch.Tensor]]

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

Find an answer set for EPFO queries including negation and disjunction

## Parameter

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

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

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

tnorm: str The t-norm operator.

neg\_norm: str The negation norm.

**lambda\_**: float lambda parameter for sugeno and yager negation norms

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

## returns

- List[Tuple[str, torch.Tensor]]

- Entities and corresponding scores sorted in the descening order of scores

**find\_missing\_triples** (*confidence: float, entities: List[str] = None, relations: List[str] = None, topk: int = 10, at\_most: int = sys.maxsize*) → Set

Find missing triples

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

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

Return (e,r,x)

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

confidence: float

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

topk: int

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

at\_most: int

Stop after finding at\_most missing triples

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

otin G

**deploy** (*share: bool = False, top\_k: int = 10*)

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

**train\_k\_vs\_all** (*h, r, iteration=1, lr=0.001*)

Train k vs all :param head\_entity: :param relation: :param iteration: :param lr: :return:

**train** (*kg, lr=0.1, epoch=10, batch\_size=32, neg\_sample\_ratio=10, num\_workers=1*) → None

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

## dicee.models

### Submodules

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

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

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

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

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

#### 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):
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

#### Note

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



## Variables

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

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

`loss`

`selected_optimizer = None`

`normalizer_class = None`

`normalize_head_entity_embeddings`

`normalize_relation_embeddings`

`normalize_tail_entity_embeddings`

`hidden_normalizer`

`param_init`

`input_dp_ent_real`

`input_dp_rel_real`

`hidden_dropout`

`loss_history = []`

`byte_pair_encoding`

`max_length_subword_tokens`

`block_size`

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

**Parameters**

**x** (*B* × 2 × *T*)

**forward\_byte\_pair\_encoded\_triple** (*x*: *Tuple*[*torch.LongTensor*, *torch.LongTensor*])

byte pair encoded neural link predictors

**Parameters**

-----

**init\_params\_with\_sanity\_checking** ()

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

**Parameters**

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

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

**Parameters**

**x**

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

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

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

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

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

**Parameters**

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

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

**Parameters**

**x** (*B* × 2 × *T*)

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

**class** *dicee.models.base\_model.IdentityClass* (*args*=*None*)

Bases: *torch.nn.Module*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### Note

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

#### Variables

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

**args**

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

**static forward** (*x*)

## dicee.models.clifford

### Classes

<i>Keci</i>	Base class for all neural network modules.
<i>KeciBase</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 to nest them in a tree structure. You can assign the submodules as regular attributes:

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

(continues on next page)

(continued from previous page)

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

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

Submodules assigned in this way will be registered, and will have their parameters converted too 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  $p = e_1, e_2, e_3$ , we compute interactions between  $e_1 e_2, e_1 e_3$ , and  $e_2 e_3$  This can be implemented with a nested two for loops

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

    for k in range(i + 1, p):

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

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

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

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

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

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

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

```

results = [] for j in range(q - 1):
    for k in range(j + 1, q):
        results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])
sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))

```

Yet, this computation would be quite inefficient. Instead, we compute interactions along all  $p$ , e.g.,  $e_1e_1$ ,  $e_1e_2$ ,  $e_1e_3$ ,

```

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

```

Then select the triangular matrix without diagonals:  $e_1e_2$ ,  $e_1e_3$ ,  $e_2e_3$ .

```

compute_sigma_pq(*, hp, hq, rp, rq)
sum_{i=1}^p sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j
results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):
    for j in range(q):
        sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
print(sigma_pq.shape)

```

**apply\_coefficients** ( $hp, hq, rp, rq$ )  
 Multiplying a base vector with its scalar coefficient

**clifford\_multiplication** ( $h0, hp, hq, r0, rp, rq$ )  
 Compute our CL multiplication

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

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

$$eq_j$$

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

- (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 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** `dicce.models.clifford.KeciBase` (*args*)

Bases: *Keci*

Without learning dimension scaling

**name** = 'KeciBase'

```
requires_grad_for_interactions = False
```

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### Note

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

#### Variables

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

**name** = 'DeCaL'

**entity\_embeddings**

**relation\_embeddings**

**p**

**q**

**r**

**re**

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

## Parameter

x: torch.LongTensor with (n, ) shape

**rtype**

torch.FloatTensor with (n) shape

**cl\_pqr** (a: torch.tensor) → torch.tensor

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

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

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

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

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

and return:

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

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

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

For same bases vectors interaction we have

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

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, |E|) shape

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

Multiplying a base vector with its scalar coefficient

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

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



## Parameter

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

### returns

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

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

Compute .. math:

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

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

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

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)

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

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

Compute

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

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

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

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

```
    print(sigma_pq.shape)
```

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

Compute

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

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

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

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

```
    print(sigma_pq.shape)
```

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

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

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

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

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

```
    print(sigma_pq.shape)
```

## dicee.models.complex

### Classes

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

### Module Contents

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

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

Convolutional ComplEx Knowledge Graph Embeddings

```
name = 'ConEx'
```

```

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

residual_convolution (C_1: Tuple[torch.Tensor, torch.Tensor],
                      C_2: Tuple[torch.Tensor, torch.Tensor]) → torch.FloatTensor
    Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors
    that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds
    complex-valued embeddings :return:

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

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

    Parameters
    x

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

class dicee.models.complex.AConEx (args)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional ComplEx Knowledge Graph Embeddings
    name = 'AConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

residual_convolution (C_1: Tuple[torch.Tensor, torch.Tensor],
                      C_2: Tuple[torch.Tensor, torch.Tensor]) → torch.FloatTensor
    Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors
    that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds
    complex-valued embeddings :return:

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

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

    Parameters
    x

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

```

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### 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

<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> )
--------------	--

### 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**

**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

$\mathbf{T}(x: \text{torch.tensor}) \rightarrow \text{torch.tensor}$

Transpose function

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

## dicее.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 dicее.models.function_space.FMult (args)
    Bases: dicее.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    name = 'FMult'
    entity_embeddings
    relation_embeddings
    k
    num_sample = 50
    gamma
    roots
    weights
    compute_func (weights: torch.FloatTensor, x)  $\rightarrow$  torch.FloatTensor
    chain_func (weights, x: torch.FloatTensor)
    forward_triples (idx_triple: torch.Tensor)  $\rightarrow$  torch.Tensor

    Parameters
    x
class dicее.models.function_space.GFMult (args)
    Bases: dicее.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:  $f(x) = \sum_{i=0}^{d-1} a_i x^{i\%d}$  and use the three differents scoring function as in the paper to evaluate the score.
    We also consider combining with Neural Networks.

    name = 'LFMult'

    entity_embeddings

    relation_embeddings

    degree

    m

    x_values

    forward_triples(idx_triple)

    Parameters
    x

    construct_multi_coeff(x)

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

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

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

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

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

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

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

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

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

## 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

<i>octonion_mul</i> (*, O_1, O_2)
<i>octonion_mul_norm</i> (*, 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 to nest them in a tree structure. You can assign the submodules as regular attributes:

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

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

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

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

### 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.,  $[\text{score}(h,r,x) | x \in \text{Entities}] \Rightarrow [0.0, 0.1, \dots, 0.8]$ , shape  $\Rightarrow (1, \text{Entities!})$  Given a batch of head entities and relations  $\Rightarrow \text{shape}(\text{size of batch}, |\text{Entities}|)$

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### 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:

**model\_kwargs**

```

name

model

loss_history = []

args

entity_embeddings = None

relation_embeddings = None

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

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

    # (3) Reshape all entities. if self.last_dim > 0:
        t = self.entity_embeddings.weight.reshape(self.num_entities, self.embedding_dim, self.last_dim)

    else:
        t = self.entity_embeddings.weight

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

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

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

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

abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)

```

## dicee.models.quaternion

### Classes

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

## Functions

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

## Module Contents

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

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### 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*)  $\rightarrow$  *torch.FloatTensor*

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

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

### 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, 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

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

## Module Contents

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

```

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

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

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

```

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

#### **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**

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

```

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

### 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>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.
<i>KeciBase</i>	Without learning dimension scaling

continues on next page

Table 1 – continued from previous page

<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>GMult</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.**BaseKGELightning**(\*args, \*\*kwargs)

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

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

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

```

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

#### Note

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

### Variables

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

**training\_step\_outputs** = []

**mem\_of\_model**() → Dict

Size of model in MB and number of params

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

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

### Parameters

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

### Returns

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

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

Example:

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

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

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

# Multiple optimizers (e.g.: GANs)
```

(continues on next page)

(continued from previous page)

```
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

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

### Note

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

**loss\_function** (*yhat\_batch*: *torch.FloatTensor*, *y\_batch*: *torch.FloatTensor*)

#### Parameters

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

**on\_train\_epoch\_end** (\*args, \*\*kwargs)

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

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

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

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

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

**test\_epoch\_end** (outputs: *List[Any]*)

**test\_dataloader**() → *None*

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

#### **Warning**

do not assign state in `prepare_data`

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

#### **Note**

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

#### **Note**

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

`val_dataloader()` → None

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:param-ref:~lightning.pytorch.trainer.trainer.Trainer.reload\_dataloaders\_every\_n\_epochs** to a positive integer.

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

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

#### **Note**

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

#### **Note**

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

`predict_dataloader()` → None

An iterable or collection of iterables specifying prediction samples.

For more information about multiple dataloaders, see this section.

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

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

#### Note

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

#### Returns

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

`train_dataloader()` → None

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:param-ref:~lightning.pytorch.trainer.trainer.Trainer.reload\_dataloaders\_every\_n\_epochs** to a positive integer.

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

#### Warning

do not assign state in `prepare_data`

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

#### Note

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

**configure\_optimizers** (*parameters=None*)

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

### Returns

Any of these 6 options.

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

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

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

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

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

### Note

Some things to know:

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

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

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

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

(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 have their parameters converted too 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**

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

**static forward**(*x*)

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### **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
         $\mathbf{x}$ 

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

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

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

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

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

### Note

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

### Variables

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

**args**

**embedding\_dim** = None

**num\_entities** = None

**num\_relations** = None

**num\_tokens** = None

**learning\_rate** = None

**apply\_unit\_norm** = None

**input\_dropout\_rate** = None

**hidden\_dropout\_rate** = None

```

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

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

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

    Parameters
    -----

init_params_with_sanity_checking()

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

    Parameters

    •  $\mathbf{x}$ 

    •  $\mathbf{y\_idx}$ 

    • ordered_bpe_entities

```



```

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

    Parameters
    x

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

get_sentence_representation (x: torch.LongTensor)

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

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

    Parameters
    x (B × 2 × 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 to nest them in a tree structure. You can assign the submodules as regular attributes:

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

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

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

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

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

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

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

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

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

#### 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.IdentityClass (args=None)`

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### Note

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

### Variables

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

`args`

`__call__(x)`

`static forward(x)`

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

`class dicee.models.QMult (args)`

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

(continues on next page)

```

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

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

```

Submodules assigned in this way will be registered, and will have their parameters converted too 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 to nest them in a tree structure. You can assign the submodules as regular attributes:

```

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

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

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

```

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

### 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.IdentityClass (args=None)
```

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### Note

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

#### Variables

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

**args**

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

**static forward** (*x*)

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

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

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

(continues on next page)

(continued from previous page)

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

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

Submodules assigned in this way will be registered, and will have their parameters converted too 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.,  $[\text{score}(h,r,x) | x \text{ in Entities}] \Rightarrow [0.0, 0.1, \dots, 0.8]$ , shape  $\Rightarrow (1, \text{Entities!})$  Given a batch of head entities and relations  $\Rightarrow$  shape (size of batch, l Entities!)

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)

(continued from previous page)

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

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

Submodules assigned in this way will be registered, and will have their parameters converted too 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 to nest them in a tree structure. You can assign the submodules as regular attributes:

```

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

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

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

```

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

### Note

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

### Variables

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

```

name = 'Keci'

p

q

r

requires_grad_for_interactions = True

compute_sigma_pp(hp, rp)
    Compute  $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k$ 
     $\sigma_{pp}$  captures the interactions between along p bases For instance, let p e_1, e_2, e_3, we compute
    interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops

    results = [] for i in range(p - 1):
        for k in range(i + 1, p):
            results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

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

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

    e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

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

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

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

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

    e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

compute_sigma_pq(*, hp, hq, rp, rq)
     $\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$ 
    results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):
        for j in range(q):
            sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

    print(sigma_pq.shape)

apply_coefficients(hp, hq, rp, rq)
    Multiplying a base vector with its scalar coefficient

```



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

Compute our CL multiplication

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

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

$e_j$

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

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

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

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

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

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

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

**construct\_cl\_multivector** (*x: torch.FloatTensor, r: int, p: int, q: int*)

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

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

## Parameter

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

**returns**

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

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

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

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

Kvsall training

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

**forward\_k\_vs\_with\_explicit** and this functions are identical Parameter ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, **IEI**) shape

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

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

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

## Parameter

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

### returns

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

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

## Parameter

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

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

### rtype

torch.FloatTensor with (n, k) shape

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

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

## Parameter

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

### rtype

torch.FloatTensor with (n) shape

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

Bases: *Keci*

Without learning dimension scaling

**name** = 'KeciBase'

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

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(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 have their parameters converted too when you call `to()`, etc.

### **Note**

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

### **Variables**

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

**name** = 'DeCaL'

**entity\_embeddings**

**relation\_embeddings**

**p**

**q**

**r**

**re**

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

### **Parameter**

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

**rtype**

torch.FloatTensor with (n) shape

**cl\_pqr** (*a: torch.tensor*) → torch.tensor

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

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

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

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

$$s_0 = h_0 r_0 t_0 s_1 = \sum_{i=1}^p h_i r_i t_0 s_2 = \sum_{j=p+1}^{p+q} h_j r_j t_0 s_3 = \sum_{i=1}^q (h_0 r_i t_i + h_i r_0 t_i) s_4 = \sum_{i=p+1}^{p+q} (h_0 r_i t_i + h_i r_0 t_i) s_5 = \sum_{i=p+q+1}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i)$$

and return:

$$\sigma_0 t = \sigma_0 \cdot t_0 = s_0 + s_1 - s_2 s_3, s_4 \text{ and } s_5$$

**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 e\_1, e\_2, e\_3, we compute interactions between e\_1 e\_2, e\_1 e\_3, and e\_2 e\_3 This can be implemented with a nested two for loops

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

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

    Compute

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

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

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

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

    Compute

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

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

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

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

print(sigma\_pq.shape)

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

    Compute

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

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

```

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

```

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

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

```

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

```

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

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

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

```

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

### 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.PykeenKGE(args: dict)
```

Bases: `dicee.models.base_model.BaseKGE`

A class for using knowledge graph embedding models implemented in Pykeen

Notes: Pykeen\_DistMult: C Pykeen\_ComplEx: Pykeen\_QuatE: Pykeen\_MuRE: Pykeen\_CP: Pykeen\_HolE: Pykeen\_HolE:

**model\_kwargs**

**name**

**model**

**loss\_history** = []

**args**

**entity\_embeddings** = None



```
relation_embeddings = None
```

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

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

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

```
    h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim,
    self.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 to nest them in a tree structure. You can assign the submodules as regular attributes:

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

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

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

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

### 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.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 differents scoring function as in the paper to evaluate
the score

name = 'LFMult1'

entity_embeddings

relation_embeddings

forward_triples (idx_triple)

Parameters
x
tri_score (h, r, t)

vtp_score (h, r, t)

class dicee.models.LFMult (args)
Bases: dicee.models.base_model.BaseKGE

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

name = 'LFMult'

```

**entity\_embeddings**

**relation\_embeddings**

**degree**

**m**

**x\_values**

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

### Parameters

**x**

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

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

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

**linear** (*x, w, b*)

**scalar\_batch\_NN** (*a, b, c*)

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

**tri\_score** (*coeff\_h, coeff\_r, coeff\_t*)

this part implement the trilinear scoring techniques:

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

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

**vtp\_score** (*h, r, t*)

this part implement the vector triple product scoring techniques:

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

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

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

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

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

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

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

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

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

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

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

Bases: `dicee.models.base_model.BaseKGE`

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

**name** = 'DualE'

**entity\_embeddings**

**relation\_embeddings**

**num\_ent**

**kvsall\_score** (*e\_1\_h*, *e\_2\_h*, *e\_3\_h*, *e\_4\_h*, *e\_5\_h*, *e\_6\_h*, *e\_7\_h*, *e\_8\_h*, *e\_1\_t*, *e\_2\_t*, *e\_3\_t*, *e\_4\_t*,  
*e\_5\_t*, *e\_6\_t*, *e\_7\_t*, *e\_8\_t*, *r\_1*, *r\_2*, *r\_3*, *r\_4*, *r\_5*, *r\_6*, *r\_7*, *r\_8*) → torch.tensor

KvsAll scoring function

### Input

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

### Output

torch.FloatTensor with (n) shape

**forward\_triples** (*idx\_triple*: torch.tensor) → torch.tensor

Negative Sampling forward pass:

### Input

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

### Output

torch.FloatTensor with (n) shape

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

KvsAll forward pass

### Input

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

### Output

torch.FloatTensor with (n) shape

**T** (*x: torch.tensor*) → torch.tensor

Transpose function

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

## dicee.query\_generator

### Classes

---

*QueryGenerator*

---

### Module Contents

```
class dicee.query_generator.QueryGenerator (train_path, val_path: str, test_path: str,  
      ent2id: Dict = None, rel2id: Dict = None, seed: int = 1, gen_valid: bool = False,  
      gen_test: bool = True)  
  
    train_path  
  
    val_path  
  
    test_path  
  
    gen_valid  
  
    gen_test  
  
    seed  
  
    max_ans_num = 1000000.0  
  
    mode  
  
    ent2id  
  
    rel2id: Dict  
  
    ent_in: Dict  
  
    ent_out: Dict  
  
    query_name_to_struct  
  
    list2tuple (list_data)  
  
    tuple2list (x: List | Tuple) → List | Tuple  
        Convert a nested tuple to a nested list.  
  
    set_global_seed (seed: int)  
        Set seed  
  
    construct_graph (paths: List[str]) → Tuple[Dict, Dict]  
        Construct graph from triples Returns dicts with incoming and outgoing edges  
  
    fill_query (query_structure: List[str | List], ent_in: Dict, ent_out: Dict, answer: int) → bool  
        Private method for fill_query logic.
```



```

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

write_links (ent_out, small_ent_out)

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

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

unmap_query (query_structure, query, id2ent, id2rel)

generate_queries (query_struct: List, gen_num: int, query_type: str)
    Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting
    queries and answers in return @ TODO: create a class for each single query struct

save_queries (query_type: str, gen_num: int, save_path: str)

abstract load_queries (path)

get_queries (query_type: str, gen_num: int)

static save_queries_and_answers (path: str, data: List[Tuple[str, Tuple[collections.defaultdict]]])
    → None
    Save Queries into Disk

static load_queries_and_answers (path: str) → List[Tuple[str, Tuple[collections.defaultdict]]]
    Load Queries from Disk to Memory

```

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

### **Submodules**

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

### **Classes**

<i>PreprocessKG</i>	Preprocess the data in memory
---------------------	-------------------------------

### **Module Contents**

```

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

    kg

    start () → None
        Preprocess train, valid and test datasets stored in knowledge graph instance

```

### **Parameter**

```

    rtype
        None

```

`preprocess_with_byte_pair_encoding()`

`preprocess_with_byte_pair_encoding_with_padding()` → None

`preprocess_with_pandas()` → None

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

- (1) Add recipriocal or noisy triples
- (2) Construct vocabulary
- (3) Index datasets

### Parameter

**rtype**

None

`preprocess_with_polars()` → None

`sequential_vocabulary_construction()` → None

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

**`dicee.read_preprocess_save_load_kg.read_from_disk`**

### Classes

---

*ReadFromDisk*

Read the data from disk into memory

---

### Module Contents

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

Read the data from disk into memory

**kg**

**start()** → None

Read a knowledge graph from disk into memory

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

### Parameter

None

**rtype**

None

**`add_noisy_triples_into_training()`**

`dicee.read_preprocess_save_load_kg.save_load_disk`

## Classes

---

*LoadSaveToDisk*

---

## Module Contents

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

**kg**

**save** ()

**load** ()

`dicee.read_preprocess_save_load_kg.util`

## Functions

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

## Module Contents

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

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

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

## Parameters:

### **df\_polars**

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

### **idx\_entity**

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

### **idx\_relation**

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

## Returns:

### **polars.DataFrame**

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

## Example Usage:

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

## Steps:

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

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

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

## Parameters:

### **df\_pandas**

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

### **idx\_entity**

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

### **idx\_relation**

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

## Returns:

### **pd.DataFrame**

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

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

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

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

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

Load and Preprocess via Polars

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

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

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

Read triples from triple store into pandas dataframe

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

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

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

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

(1) Extract domains and ranges of relations

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

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

Deserialize data

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

```

dicee.read_preprocess_save_load_kg.util.load_numpy_ndarray(* , file_path: str)
dicee.read_preprocess_save_load_kg.util.save_pickle(* , data: object, file_path=str)
dicee.read_preprocess_save_load_kg.util.load_pickle(* , file_path=str)
dicee.read_preprocess_save_load_kg.util.create_recipriocal_triples(x)
    Add inverse triples into dask dataframe :param x: :return:
dicee.read_preprocess_save_load_kg.util.dataset_sanity_checking(
    train_set: numpy.ndarray, num_entities: int, num_relations: int) → None

```

#### Parameters

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

#### Returns

### Classes

<i>PreprocessKG</i>	Preprocess the data in memory
<i>LoadSaveToDisk</i>	
<i>ReadFromDisk</i>	Read the data from disk into memory

### Package Contents

```

class dicee.read_preprocess_save_load_kg.PreprocessKG(kg)
    Preprocess the data in memory
    kg
    start() → None
        Preprocess train, valid and test datasets stored in knowledge graph instance

```

#### Parameter

```

    rtype
        None

    preprocess_with_byte_pair_encoding()

    preprocess_with_byte_pair_encoding_with_padding() → None

    preprocess_with_pandas() → None
        Preprocess train, valid and test datasets stored in knowledge graph instance with pandas
        (1) Add recipriocal or noisy triples
        (2) Construct vocabulary
        (3) Index datasets

```

## Parameter

**rtype**  
None

**preprocess\_with\_polars()** → None

**sequential\_vocabulary\_construction()** → None

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

**class** dicee.read\_preprocess\_save\_load\_kg.**LoadSaveToDisk**(kg)

**kg**

**save()**

**load()**

**class** dicee.read\_preprocess\_save\_load\_kg.**ReadFromDisk**(kg)

Read the data from disk into memory

**kg**

**start()** → None

Read a knowledge graph from disk into memory

Data will be available at the train\_set, test\_set, valid\_set attributes.

## Parameter

None

**rtype**  
None

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

## **dicee.sanity\_checkers**

### Functions

```
is_sparql_endpoint_alive([sparql_endpoint])
```

```
validate_knowledge_graph(args)
```

Validating the source of knowledge graph

```
sanity_checking_with_arguments(args)
```

## 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.scripts**

### **Submodules**

#### **dicee.scripts.index**

#### **Functions**

---

`get_default_arguments()`

`main()`

---

### **Module Contents**

`dicee.scripts.index.get_default_arguments ()`

`dicee.scripts.index.main ()`

## **dicee.scripts.run**

### **Functions**

---

`get_default_arguments([description])`

Extends pytorch\_lightning Trainer's arguments with ours

`main()`

---

### **Module Contents**

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

Extends pytorch\_lightning Trainer's arguments with ours

`dicee.scripts.run.main ()`

## **dicee.scripts.serve**

### **Attributes**

---

`app`

`neural_searcher`

---

## Classes

---

```
NeuralSearcher
```

---

## Functions

---

```
get_default_arguments()
```

```
root()
```

```
search_embeddings(q)
```

```
retrieve_embeddings(q)
```

```
main()
```

---

## Module Contents

```
dicee.scripts.serve.app
```

```
dicee.scripts.serve.neural_searcher = None
```

```
dicee.scripts.serve.get_default_arguments()
```

```
async dicee.scripts.serve.root()
```

```
async dicee.scripts.serve.search_embeddings(q: str)
```

```
async dicee.scripts.serve.retrieve_embeddings(q: str)
```

```
class dicee.scripts.serve.NeuralSearcher(args)
```

```
    collection_name
```

```
    model
```

```
    qdrant_client
```

```
    get(entity: str)
```

```
    search(entity: str)
```

```
dicee.scripts.serve.main()
```

## `dicee.static_funcs`

## Functions

---

```
create_recipriocal_triples(x)
```

```
Add inverse triples into dask dataframe
```

continues on next page

Table 2 – continued from previous page

<code>get_er_vocab(data[, file_path])</code>	
<code>get_re_vocab(data[, file_path])</code>	
<code>get_ee_vocab(data[, file_path])</code>	
<code>timeit(func)</code>	
<code>save_pickle(*[, data, file_path])</code>	
<code>load_pickle([file_path])</code>	
<code>load_term_mapping([file_path])</code>	
<code>select_model(args[, is_continual_training, storage_path])</code>	
<code>load_model(→ Tuple[object, Tuple[dict, dict]])</code>	Load weights and initialize pytorch module from namespace arguments
<code>load_model_ensemble(...)</code>	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
<code>save_numpy_ndarray(*, data, file_path)</code>	
<code>numpy_data_type_changer(→ numpy.ndarray)</code>	Detect most efficient data type for a given triples
<code>save_checkpoint_model(→ None)</code>	Store Pytorch model into disk
<code>store(→ None)</code>	Store trained_model model and save embeddings into csv file.
<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>	

continues on next page

Table 2 – continued from previous page

---

`load_numpy(→ numpy.ndarray)`
`evaluate(entity_to_idx, scores, easy_answers, # @TODO: CD: Renamed this function  
hard_answers)`
`download_file(url[, destination_folder])`
`download_files_from_url(→ None)`
`download_pretrained_model(→ str)`


---

## Module Contents

`dicee.static_funcs.create_recipriocal_triples(x)`

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

`dicee.static_funcs.get_er_vocab(data, file_path: str = None)`
`dicee.static_funcs.get_re_vocab(data, file_path: str = None)`
`dicee.static_funcs.get_ee_vocab(data, file_path: str = None)`
`dicee.static_funcs.timeit(func)`
`dicee.static_funcs.save_pickle(*, data: object = None, file_path=str)`
`dicee.static_funcs.load_pickle(file_path=str)`
`dicee.static_funcs.load_term_mapping(file_path=str)`
`dicee.static_funcs.select_model(args: dict, is_continual_training: bool = None,  
storage_path: str = None)`
`dicee.static_funcs.load_model(path_of_experiment_folder: str, model_name='model.pt', verbose=0)  
→ Tuple[object, Tuple[dict, dict]]`

Load weights and initialize pytorch module from namespace arguments

`dicee.static_funcs.load_model_ensemble(path_of_experiment_folder: str)  
→ Tuple[dicee.models.base_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]`

Construct Ensemble Of weights and initialize pytorch module from namespace arguments

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

`dicee.static_funcs.save_numpy_ndarray(*, data: numpy.ndarray, file_path: str)`
`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(trainer, trained_model, model_name: str = 'model',
    full_storage_path: str = None, save_embeddings_as_csv=False) → None
    Store trained_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param
    full_storage_path: path to save parameters. :param model_name: string representation of the name of the model.
    :param trained_model: an instance of BaseKGE see core.models.base_model . :param save_embeddings_as_csv:
    for easy access of embeddings. :return:

dicee.static_funcs.add_noisy_triples(train_set: pandas.DataFrame, add_noise_rate: float)
    → pandas.DataFrame
    Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

dicee.static_funcs.read_or_load_kg(args, cls)

dicee.static_funcs.intialize_model(args: dict, verbose=0) → Tuple[object, str]

dicee.static_funcs.load_json(p: str) → dict

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

dicee.static_funcs.random_prediction(pre_trained_kge)

dicee.static_funcs.deploy_triple_prediction(pre_trained_kge, str_subject, str_predicate,
    str_object)

dicee.static_funcs.deploy_tail_entity_prediction(pre_trained_kge, str_subject, str_predicate,
    top_k)

dicee.static_funcs.deploy_head_entity_prediction(pre_trained_kge, str_object, str_predicate,
    top_k)

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

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

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

dicee.static_funcs.continual_training_setup_executor(executor) → None

dicee.static_funcs.exponential_function(x: numpy.ndarray, lam: float, ascending_order=True)
    → torch.FloatTensor

dicee.static_funcs.load_numpy(path) → numpy.ndarray

dicee.static_funcs.evaluate(entity_to_idx, scores, easy_answers, hard_answers)
    # @TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

dicee.static_funcs.download_file(url, destination_folder='.')

dicee.static_funcs.download_files_from_url(base_url: str, destination_folder='.') → None

```

### Parameters

- **base\_url** (e.g. `"https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll"`)
- **destination\_folder** (e.g. `"KINSHIP-Keci-dim128-epoch256-KvsAll"`)

```

dicee.static_funcs.download_pretrained_model(url: str) → str

```

## dicee.static\_funcs\_training

### Functions

---

```
make_iterable_verbose(→ Iterable)

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

efficient_zero_grad(model)
```

---

### Module Contents

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

```
dicee.static_funcs_training.evaluate_lp(model, triple_idx, num_entities,
er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List], info='Eval Starts')
```

Evaluate model in a standard link prediction task

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

```
dicee.static_funcs_training.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

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

## 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

<code>EnsembleKGE</code>	
<code>DICE_Trainer</code>	DICE_Trainer implement

## Functions

```
load_term_mapping([file_path])
```

```
initialize_trainer(...)
```

```
get_callbacks(args)
```

## Module Contents

```
class dicee.trainer.dice_trainer.EnsembleKGE(model)
```

```
models = []
```

```
optimizers = []
```

```
__iter__()
```

```
__len__()
```

```
__call__(*args, **kwargs)
```

```
__getattr__(name)
```

```
__str__()
```

```
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.MP | dicee.trainer.torch_trainer_ddp.TorchDDP
```

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

```
class dicee.trainer.dice_trainer.DICE_Trainer(args, is_continual_training, storage_path, evaluator=None)
```

### DICE\_Trainer implement

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

```
args
```

```
is_continual_training:bool
```

```
storage_path:str
```

```
evaluator:
```

```
report:dict
```

```
report
```

```
args
```

```
trainer = None
```

```
is_continual_training
```



**storage\_path**

**evaluator**

**form\_of\_labelling** = None

**continual\_start** (*knowledge\_graph*)

- (1) Initialize training.
- (2) Load model
- (3) Load trainer (3) Fit model

## Parameter

### returns

- *model*
- **form\_of\_labelling** (*str*)

**initialize\_trainer** (*callbacks: List*)

→ `lightning.Trainer` | `dicke.trainer.model_parallelism.MP` | `dicke.trainer.torch_trainer.TorchTrainer` | `dicke.trainer.torch`

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>MP</i>	Abstract class for Trainer class for knowledge graph embedding models
-----------	---

### Module Contents

**class** `dicee.trainer.model_parallelism.MP` (*args, callbacks*)  
Bases: `dicee.abstracts.AbstractTrainer`  
Abstract class for Trainer class for knowledge graph embedding models

#### Parameter

**args**  
[str] ?  
**callbacks: list**  
?  
**get\_ensemble()**  
**fit** (\*args, \*\*kwargs)  
Train model  
**extract\_input\_outputs** (z: list)

## dicee.trainer.torch\_trainer

### Classes

<i>xMP</i>	Abstract class for Trainer class for knowledge graph embedding models
<i>TorchTrainer</i>	TorchTrainer for using single GPU or multi CPUs on a single node

### Module Contents

**class** `dicee.trainer.torch_trainer.xMP` (*args, callbacks*)  
Bases: `dicee.abstracts.AbstractTrainer`  
Abstract class for Trainer class for knowledge graph embedding models

#### Parameter

**args**  
[str] ?  
**callbacks: list**  
?

```

loss_function = None

optimizer = None

model = None

train_dataloaders = None

training_step = None

available_gpus

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

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\_data loaders, \*\*kwargs) → None

Training starts

Arguments

**kwargs: Tuple**

empty dictionary

**Return type**

batch loss (float)

**forward\_backward\_update** (x\_batch: torch.Tensor, y\_batch: torch.Tensor) → torch.Tensor

Compute forward, loss, backward, and parameter update

Arguments

**Return type**

batch loss (float)

**extract\_input\_outputs\_set\_device** (batch: list) → Tuple

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

Arguments

**Return type**

(tuple) mini-batch on select device

## **dicee.trainer.torch\_trainer\_ddp**

### **Classes**

*TorchDDPTrainer*

A Trainer based on torch.nn.parallel.DistributedDataParallel

*NodeTrainer*

### **Functions**

*make\_iterable\_verbose*(→ Iterable)

### **Module Contents**

dicee.trainer.torch\_trainer\_ddp.**make\_iterable\_verbose** (iterable\_object, verbose,  
desc='Default', position=None, leave=True) → Iterable

```

class dicee.trainer.torch_trainer_ddp.TorchDDPTrainer (args, callbacks)
    Bases: dicee.abstracts.AbstractTrainer

    A Trainer based on torch.nn.parallel.DistributedDataParallel

    Arguments

    entity_idx
        mapping.

    relation_idx
        mapping.

    form
        ?

    store
        ?

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

    Return type
        torch.utils.data.Dataset

    fit (*args, **kwargs)
        Train model

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

    trainer

    local_rank

    global_rank

    optimizer

    train_dataset_loader

    loss_func

    callbacks

    model

    num_epochs

    loss_history = []

    ctx

    scaler

    extract_input_outputs (z: list)

    train ()
        Training loop for DDP

```

## Classes

<i>DICE_Trainer</i>	DICE_Trainer implement
---------------------	------------------------

## Package Contents

**class** `dicee.trainer.DICE_Trainer` (*args*, *is\_continual\_training*, *storage\_path*, *evaluator=None*)

### DICE\_Trainer implement

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

*args*

*is\_continual\_training*:bool

*storage\_path*:str

*evaluator*:

*report*:dict

**report**

**args**

**trainer = None**

**is\_continual\_training**

**storage\_path**

**evaluator**

**form\_of\_labelling = None**

**continual\_start** (*knowledge\_graph*)

- (1) Initialize training.
- (2) Load model
- (3) Load trainer (3) Fit model

### Parameter

**returns**

- *model*
- **form\_of\_labelling** (*str*)

**initialize\_trainer** (*callbacks: List*)

→ `lightning.Trainer` | `dicee.trainer.model_parallelism.MP` | `dicee.trainer.torch_trainer.TorchTrainer` | `dicee.trainer.torch_trainer.TorchTrainer`

Initialize Trainer from input arguments

**initialize\_or\_load\_model** ()

**init\_dataloader** (*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

**init\_dataset** () → torch.utils.data.Dataset

**start** (*knowledge\_graph: dicee.knowledge\_graph.KG | numpy.memmap*)  
→ Tuple[*dicee.models.base\_model.BaseKGE*, str]

Start the training

(1) Initialize Trainer

(2) Initialize or load a pretrained KGE model

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

**k\_fold\_cross\_validation** (*dataset*) → Tuple[*dicee.models.base\_model.BaseKGE*, str]

Perform K-fold Cross-Validation

1. Obtain K train and test splits.

2. **For each split,**

2.1 initialize trainer and model 2.2. Train model with configuration provided in args. 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.

3. Report the mean and average MRR .

#### Parameters

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

#### Returns

model

## 14.2 Attributes

`__version__`

## 14.3 Classes

<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>KeciBase</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

continues on next page

Table 3 – continued from previous page

<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>DICE_Trainer</i>	DICE_Trainer implement
<i>KGE</i>	Knowledge Graph Embedding Class for interactive usage of pre-trained models
<i>Execute</i>	A class for Training, Retraining and Evaluation a model.
<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
<i>QueryGenerator</i>	

## 14.4 Functions

<i>create_recipriocal_triples(x)</i>	Add inverse triples into dask dataframe
<i>get_er_vocab(data[, file_path])</i>	
<i>get_re_vocab(data[, file_path])</i>	
<i>get_ee_vocab(data[, file_path])</i>	
<i>timeit(func)</i>	
<i>save_pickle(*[, data, file_path])</i>	

continues on next page



Table 4 – continued from previous page

<code>load_pickle([file_path])</code>	
<code>load_term_mapping([file_path])</code>	
<code>select_model(args[, is_continual_training, storage_path])</code>	
<code>load_model(→ Tuple[object, Tuple[dict, dict]])</code>	Load weights and initialize pytorch module from namespace arguments
<code>load_model_ensemble(...)</code>	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
<code>save_numpy_ndarray(*, data, file_path)</code>	
<code>numpy_data_type_changer(→ numpy.ndarray)</code>	Detect most efficient data type for a given triples
<code>save_checkpoint_model(→ None)</code>	Store Pytorch model into disk
<code>store(→ None)</code>	Store trained_model model and save embeddings into csv file.
<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>	

continues on next page

Table 4 – continued from previous page

<code>mapping_from_first_two_cols_to_third(train_se</code>	
<code>timeit(func)</code>	
<code>load_term_mapping([file_path])</code>	
<code>reload_dataset(path, form_of_labelling, ...)</code>	Reload the files from disk to construct the Pytorch dataset
<code>construct_dataset(→ torch.utils.data.Dataset)</code>	

## 14.5 Package Contents

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

Bases: `dicee.models.base_model.BaseKGE`

A Physical Embedding Model for Knowledge Graphs

**name** = 'Pyke'

**dist\_func**

**margin** = 1.0

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

**Parameters**

**x**

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

Bases: `dicee.models.base_model.BaseKGE`

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

**name** = 'DistMult'

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

**Parameters**

• **emb\_h**

• **emb\_r**

• **emb\_E**

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

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

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

**class** `dicee.KeciBase` (*args*)

Bases: `Keci`

Without learning dimension scaling

**name** = 'KeciBase'

```
requires_grad_for_interactions = False
```

```
class dicee.Keci (args)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### Note

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

#### Variables

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

```
name = 'Keci'
```

```
p
```

```
q
```

```
r
```

```
requires_grad_for_interactions = True
```

```
compute_sigma_pp (hp, rp)
```

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

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

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

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

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

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

Yet, this computation would be quite inefficient. Instead, we compute interactions along all  $p$ , e.g.,  $e_1e_1$ ,  $e_1e_2$ ,  $e_1e_3$ ,

$e_2e_1$ ,  $e_2e_2$ ,  $e_2e_3$ ,  $e_3e_1$ ,  $e_3e_2$ ,  $e_3e_3$

Then select the triangular matrix without diagonals:  $e_1e_2$ ,  $e_1e_3$ ,  $e_2e_3$ .

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

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

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

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

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

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

Yet, this computation would be quite inefficient. Instead, we compute interactions along all  $p$ , e.g.,  $e_1e_1$ ,  $e_1e_2$ ,  $e_1e_3$ ,

$e_2e_1$ ,  $e_2e_2$ ,  $e_2e_3$ ,  $e_3e_1$ ,  $e_3e_2$ ,  $e_3e_3$

Then select the triangular matrix without diagonals:  $e_1e_2$ ,  $e_1e_3$ ,  $e_2e_3$ .

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

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

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

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

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

print(sigma\_pq.shape)

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

Multiplying a base vector with its scalar coefficient

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

Compute our CL multiplication

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

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

$e_j$

$h \ r = \text{sigma}_0 + \text{sigma}_p + \text{sigma}_q + \text{sigma}_{pp} + \text{sigma}_q + \text{sigma}_{pq}$  where

(1)  $\text{sigma}_0 = h_0 \ r_0 + \sum_{i=1}^p (h_0 \ r_i - h_i \ r_0) e_i - \sum_{j=p+1}^{p+q} (h_j \ r_j) e_j$

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

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

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

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

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

**construct\_cl\_multivector** ( $x$ : torch.FloatTensor,  $r$ : int,  $p$ : int,  $q$ : int)

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

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

## Parameter

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

### returns

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

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

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

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

Kvsall training

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

forward\_k\_vs\_with\_explicit and this functions are identical Parameter ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, **IEI**) shape

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

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

## Parameter

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

### returns

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

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

## Parameter

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

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

### rtype

torch.FloatTensor with (n, k) shape

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

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

## Parameter

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

### rtype

torch.FloatTensor with (n) shape

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

Bases: `dicee.models.base_model.BaseKGE`

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

```
name = 'TransE'
```

```
margin = 4
```

```
score(head_ent_emb, rel_ent_emb, tail_ent_emb)
```

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

```
class dicee.DeCaL(args)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

### Note

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

## Variables

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

```
name = 'DeCaL'
```

**entity\_embeddings**

**relation\_embeddings**

**p**

**q**

**r**

**re**

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

### Parameter

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

**rtype**

torch.FloatTensor with (n) shape

**cl\_pqr** (*a: torch.tensor*) → torch.tensor

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

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

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

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

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

and return:

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

**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:

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

$\sigma_{\{pp\}}$  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

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



$\sigma_{\{q\}}$  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.,  $e_1 e_1, e_1 e_2, e_1 e_3$ ,

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

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.DualE(args)
    Bases: dicee.models.base_model.BaseKGE

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

    name = 'DualE'

    entity_embeddings

    relation_embeddings

    num_ent

    kvsall_score(e_1_h, e_2_h, e_3_h, e_4_h, e_5_h, e_6_h, e_7_h, e_8_h, e_1_t, e_2_t, e_3_t, e_4_t, e_5_t, e_6_t, e_7_t, e_8_t, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8) → 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 to nest them in a tree structure. You can assign the submodules as regular attributes:

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

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

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

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

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

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

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

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

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

#### Note

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

#### Variables

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

**name** = 'ConvO'

**conv2d**

**fc\_num\_input**

**fc1**

**bn\_conv2d**

**norm\_fc1**

**feature\_map\_dropout**

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

**residual\_convolution** ( $O_1, O_2$ )

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

#### Parameters

**x**

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

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

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

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

Convolutional ComplEx Knowledge Graph Embeddings

**name** = 'ConEx'

**conv2d**

**fc\_num\_input**

**fc1**

**norm\_fc1**

**bn\_conv2d**

**feature\_map\_dropout**

**residual\_convolution** ( $C_1$ : *Tuple[torch.Tensor, torch.Tensor]*,  
 $C_2$ : *Tuple[torch.Tensor, torch.Tensor]*) → *torch.FloatTensor*

Compute residual score of two complex-valued embeddings. :param C\_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C\_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

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

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

#### Parameters

**x**

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

**class** *dicee.QMult* (*args*)

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
```

(continues on next page)

(continued from previous page)

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

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

Submodules assigned in this way will be registered, and will have their parameters converted too 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.,  
 $[score(h,r,x)|x \text{ in Entities}] \Rightarrow [0.0,0.1,...,0.8]$ , shape=> (1, **Entities**) Given a batch of head entities and  
relations => shape (size of batch,| Entities|)

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

#### Note

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

#### Variables

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

```

name = 'OMult'

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

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

k_vs_all_score(bpe_head_ent_emb, bpe_rel_ent_emb, E)

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

```

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

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

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

```

name = 'Shallom'

shallom

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

forward_k_vs_all(x) → torch.FloatTensor

forward_triples(x) → torch.FloatTensor

```

**Parameters**

**x**

**Returns**

**class** dicee.LFMult(*args*)

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

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

```

name = 'LFMult'

entity_embeddings

relation_embeddings

degree

m

x_values

forward_triples(idx_triple)

```

**Parameters**

**x**

```

construct_multi_coeff(x)

```

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

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

**linear** (*x, w, b*)

**scalar\_batch\_NN** (*a, b, c*)

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

**tri\_score** (*coeff\_h, coeff\_r, coeff\_t*)

this part implement the trilinear scoring techniques:

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

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

**vtp\_score** (*h, r, t*)

this part implement the vector triple product scoring techniques:

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

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

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

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

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

This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,...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:

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

```

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

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

```

(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 have their parameters converted too 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 to nest them in a tree structure. You can assign the submodules as regular attributes:

```

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

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

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

```

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

#### Note

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

#### Variables

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

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

`loss`

`selected_optimizer = None`

```

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

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

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

    Parameters
    -----

init_params_with_sanity_checking()

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

    Parameters

        •  $\mathbf{x}$ 

        •  $\mathbf{y\_idx}$ 

        • ordered_bpe_entities

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

    Parameters
         $\mathbf{x}$ 

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

```



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

**Parameters**

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

**get\_bpe\_head\_and\_relation\_representation** (*x: torch.LongTensor*)

→ Tuple[torch.FloatTensor, torch.FloatTensor]

**Parameters**

**x** (*B × 2 × T*)

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

**dicee.create\_recipriocal\_triples** (*x*)

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

**dicee.get\_er\_vocab** (*data, file\_path: str = None*)

**dicee.get\_re\_vocab** (*data, file\_path: str = None*)

**dicee.get\_ee\_vocab** (*data, file\_path: str = None*)

**dicee.timeit** (*func*)

**dicee.save\_pickle** (\*, *data: object = None, file\_path=str*)

**dicee.load\_pickle** (*file\_path=str*)

**dicee.load\_term\_mapping** (*file\_path=str*)

**dicee.select\_model** (*args: dict, is\_continual\_training: bool = None, storage\_path: str = None*)

**dicee.load\_model** (*path\_of\_experiment\_folder: str, model\_name='model.pt', verbose=0*)

→ Tuple[object, Tuple[dict, dict]]

Load weights and initialize pytorch module from namespace arguments

**dicee.load\_model\_ensemble** (*path\_of\_experiment\_folder: str*)

→ Tuple[dicee.models.base\_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]

Construct Ensemble Of weights and initialize pytorch module from namespace arguments

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

**dicee.save\_numpy\_ndarray** (\*, *data: numpy.ndarray, file\_path: str*)

**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(trainer, trained_model, model_name: str = 'model', full_storage_path: str = None,
            save_embeddings_as_csv=False) → None
    Store trained_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param
    full_storage_path: path to save parameters. :param model_name: string representation of the name of the model.
    :param trained_model: an instance of BaseKGE see core.models.base_model . :param save_embeddings_as_csv:
    for easy access of embeddings. :return:

dicee.add_noisy_triples(train_set: pandas.DataFrame, add_noise_rate: float) → pandas.DataFrame
    Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

dicee.read_or_load_kg(args, cls)

dicee.intialize_model(args: dict, verbose=0) → Tuple[object, str]

dicee.load_json(p: str) → dict

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

dicee.random_prediction(pre_trained_kge)

dicee.deploy_triple_prediction(pre_trained_kge, str_subject, str_predicate, str_object)

dicee.deploy_tail_entity_prediction(pre_trained_kge, str_subject, str_predicate, top_k)

dicee.deploy_head_entity_prediction(pre_trained_kge, str_object, str_predicate, top_k)

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

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

dicee.create_experiment_folder(folder_name='Experiments')

dicee.continual_training_setup_executor(executor) → None

dicee.exponential_function(x: numpy.ndarray, lam: float, ascending_order=True) → torch.FloatTensor

dicee.load_numpy(path) → numpy.ndarray

dicee.evaluate(entity_to_idx, scores, easy_answers, hard_answers)
    # @TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

dicee.download_file(url, destination_folder='.')

dicee.download_files_from_url(base_url: str, destination_folder='.') → None

```

### Parameters

- **base\_url** (e.g. “<https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll>”)
- **destination\_folder** (e.g. “*KINSHIP-Keci-dim128-epoch256-KvsAll*”)

```

dicee.download_pretrained_model(url: str) → str

```

```

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

```

### DICE\_Trainer implement

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

```

args
is_continual_training:bool
storage_path:str
evaluator:
report:dict

```

**report**

**args**

**trainer = None**

**is\_continual\_training**

**storage\_path**

**evaluator**

**form\_of\_labelling = None**

**continual\_start** (*knowledge\_graph*)

- (1) Initialize training.
- (2) Load model
- (3) Load trainer (3) Fit model

## Parameter

**returns**

- *model*
- **form\_of\_labelling** (*str*)

**initialize\_trainer** (*callbacks: List*)

→ *lightning.Trainer* | *dicke.trainer.model\_parallelism.MP* | *dicke.trainer.torch\_trainer.TorchTrainer* | *dicke.trainer.torch\_trainer.TorchTrainer*

Initialize Trainer from input arguments

**initialize\_or\_load\_model** ()

**init\_data\_loader** (*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[dicee.models.base\_model.BaseKGE, str]

Perform K-fold Cross-Validation

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

#### Parameters

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

#### Returns

model

**class** dicee.KGE (*path=None, url=None, construct\_ensemble=False, model\_name=None*)

Bases: *[dicee.abstracts.BaseInteractiveKGE](#)*

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

**\_\_str\_\_** ()

**to** (*device: str*) → None

**get\_transductive\_entity\_embeddings** (*indices: torch.LongTensor | List[str], as\_pytorch=False, as\_numpy=False, as\_list=True*) → torch.FloatTensor | numpy.ndarray | List[float]

**create\_vector\_database** (*collection\_name: str, distance: str, location: str = 'localhost', port: int = 6333*)

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

**eval\_lp\_performance** (*dataset=List[Tuple[str, str, str]], filtered=True*)

**predict\_missing\_head\_entity** (*relation: List[str] | str, tail\_entity: List[str] | str, within=None*) → 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*)  
→ 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*) → 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)

Predict missing item in a given triple.

### Parameter

head\_entity: Union[str, List[str]]

String representation of selected entities.

relation: Union[str, List[str]]

String representation of selected relations.

tail\_entity: Union[str, List[str]]

String representation of selected entities.

k: int

Highest ranked k item.

### Returns: Tuple

Highest K scores and items

**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

**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

**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 descending 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)

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

confidence: float

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

topk: int

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

at\_most: int

Stop after finding at\_most missing triples

{(e,r,x) | f(e,r,x) > confidence and (e,r,x)

otin G

**deploy** (*share: bool = False, top\_k: int = 10*)

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

**train\_k\_vs\_all** (*h, r, iteration=1, lr=0.001*)

Train k vs all :param head\_entity: :param relation: :param iteration: :param lr: :return:

**train** (*kg, lr=0.1, epoch=10, batch\_size=32, neg\_sample\_ratio=10, num\_workers=1*) → None

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

**class** dicee.**Execute** (*args, continuous\_training=False*)

A class for Training, Retraining and Evaluation a model.

(1) Loading & Preprocessing & Serializing input data.

(2) Training & Validation & Testing

(3) Storing all necessary info

**args**

**is\_continual\_training**

**trainer = None**

**trained\_model = None**

**knowledge\_graph = None**

**report**

**evaluator = None**

**start\_time = None**

**setup\_executor** () → None

**dept\_read\_preprocess\_index\_serialize\_data** () → None

Read & Preprocess & Index & Serialize Input Data

(1) Read or load the data from disk into memory.

(2) Store the statistics of the data.

## Parameter

**rtype**

None

**save\_trained\_model** () → None

Save a knowledge graph embedding model

(1) Send model to eval mode and cpu.

(2) Store the memory footprint of the model.

(3) Save the model into disk.

(4) Update the stats of KG again ?



## Parameter

**rtype**

None

**end** (*form\_of\_labelling: str*) → dict

End training

(1) Store trained model.

(2) Report runtimes.

(3) Eval model if required.

## Parameter

**rtype**

A dict containing information about the training and/or evaluation

**write\_report** () → None

Report training related information in a report.json file

**start** () → dict

Start training

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

## Parameter

**rtype**

A dict containing information about the training and/or evaluation

`dicee.mapping_from_first_two_cols_to_third(train_set_idx)`

`dicee.timeit(func)`

`dicee.load_term_mapping(file_path=str)`

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

Reload the files from disk to construct the Pytorch dataset

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

**class** `dicee.BPE_NegativeSamplingDataset` (*train\_set: torch.LongTensor, ordered\_shaped\_bpe\_entities: torch.LongTensor, neg\_ratio: int*)

Bases: `torch.utils.data.Dataset`

An abstract class representing a Dataset.

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

#### Note

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

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

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

Bases: `torch.utils.data.Dataset`

An abstract class representing a `Dataset`.

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

#### Note

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

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

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

```
Bases: torch.utils.data.Dataset
```

Dataset for the 1vsALL training strategy

#### Parameters

- **train\_set\_idx** – Indexed triples for the training.
- **entity\_idxs** – mapping.
- **relation\_idxs** – mapping.
- **form** – ?
- **num\_workers** – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

#### Return type

torch.utils.data.Dataset

```
train_data
```

```
block_size
```

```
num_of_data_points
```

```
collate_fn = None
```

```
__len__()
```

```
__getitem__(idx)
```

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

```
Bases: torch.utils.data.Dataset
```

Dataset for the 1vsALL training strategy

#### Parameters

- **train\_set\_idx** – Indexed triples for the training.
- **entity\_idxs** – mapping.
- **relation\_idxs** – mapping.
- **form** – ?
- **num\_workers** – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

#### Return type

torch.utils.data.Dataset

```
train_data
```

```
target_dim
```

```
collate_fn = None
```

```
__len__()
```

```
__getitem__(idx)
```

```
class dicee.KvsAll (train_set_idx: numpy.ndarray, entity_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  $[0, 1]^{|E|}$   $\{ |E| \}$  is a binary label.

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

#### Note

TODO

**train\_set\_idx**

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

**entity\_idxes**

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

**relation\_idxes**

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

self : torch.utils.data.Dataset

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

**train\_data** = None

**train\_target** = None

**label\_smoothing\_rate**

**collate\_fn** = None

**\_\_len\_\_**()

**\_\_getitem\_\_**(idx)

```
class dicee.AllvsAll (train_set_idx: numpy.ndarray, entity_idxes, relation_idxes, label_smoothing_rate=0.0)
```

Bases: torch.utils.data.Dataset

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

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

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

#### Note

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

**train\_set\_idx**

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

**entity\_idx**

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

**relation\_idx**

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

self : torch.utils.data.Dataset

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

**train\_data** = None

**train\_target** = None

**label\_smoothing\_rate**

**collate\_fn** = None

**target\_dim**

**\_\_len\_\_**()

**\_\_getitem\_\_**(idx)

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

Bases: torch.utils.data.Dataset

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

### Parameters

- **train\_set** (*np.ndarray*) – A numpy array containing triples of knowledge graph data. Each triple consists of (head\_entity, relation, tail\_entity).
- **num\_entities** (*int*) – The number of unique entities in the knowledge graph.
- **num\_relations** (*int*) – The number of unique relations in the knowledge graph.
- **neg\_sample\_ratio** (*int, optional*) – The number of negative samples to be generated per positive sample. Must be a positive integer and less than num\_entities.
- **label\_smoothing\_rate** (*float, optional*) – A label smoothing rate to apply to the positive and negative labels. Defaults to 0.0.

**train\_data**

The input data converted into a PyTorch tensor.

**Type**

torch.Tensor

**num\_entities**

Number of entities in the dataset.

**Type**

int

**num\_relations**

Number of relations in the dataset.

**Type**

int

**neg\_sample\_ratio**

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

**Type**

int

**label\_smoothing\_rate**

The smoothing factor applied to the labels.

**Type**

torch.Tensor

**collate\_fn**

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

**Type**

function, optional

**train\_data**

**num\_entities**

**num\_relations**

**neg\_sample\_ratio**

**label\_smoothing\_rate**

**collate\_fn = None**

**\_\_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_idxxs, relation_idxxs, 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\_idxxs**  
mapping.

**relation\_idxxs**  
mapping.

**form**  
?

**store**  
?

**label\_smoothing\_rate**  
?

torch.utils.data.Dataset

**train\_data = None**

**train\_target = None**

**neg\_ratio**

**num\_entities**

**label\_smoothing\_rate**

**collate\_fn = None**

**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_set`

`length`

`num_entities`

`num_relations`

`__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.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`

`gen_test`

`seed`

```

max_ans_num = 1000000.0

mode

ent2id

rel2id: Dict

ent_in: Dict

ent_out: Dict

query_name_to_struct

list2tuple(list_data)

tuple2list(x: List | Tuple) → 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](#), 17
- [dicee.callbacks](#), 19
- [dicee.config](#), 25
- [dicee.dataset\\_classes](#), 28
- [dicee.eval\\_static\\_funcs](#), 40
- [dicee.evaluator](#), 41
- [dicee.executer](#), 42
- [dicee.knowledge\\_graph](#), 45
- [dicee.knowledge\\_graph\\_embeddings](#), 46
- [dicee.models](#), 50
  - [dicee.models.base\\_model](#), 50
  - [dicee.models.clifford](#), 59
  - [dicee.models.complex](#), 66
  - [dicee.models.dualE](#), 69
  - [dicee.models.function\\_space](#), 70
  - [dicee.models.octonion](#), 73
  - [dicee.models.pykeen\\_models](#), 76
  - [dicee.models.quaternion](#), 77
  - [dicee.models.real](#), 81
  - [dicee.models.static\\_funcs](#), 82
  - [dicee.models.transformers](#), 82
- [dicee.query\\_generator](#), 136
- [dicee.read\\_preprocess\\_save\\_load\\_kg](#), 137
- [dicee.read\\_preprocess\\_save\\_load\\_kg.preprocess](#), 137
- [dicee.read\\_preprocess\\_save\\_load\\_kg.read\\_from\\_disk](#), 138
- [dicee.read\\_preprocess\\_save\\_load\\_kg.save\\_load\\_disk](#), 139
- [dicee.read\\_preprocess\\_save\\_load\\_kg.util](#), 139
- [dicee.sanity\\_checkers](#), 144
- [dicee.scripts](#), 145
  - [dicee.scripts.index](#), 145
  - [dicee.scripts.run](#), 145
  - [dicee.scripts.serve](#), 145
- [dicee.static\\_funcs](#), 146
- [dicee.static\\_funcs\\_training](#), 150
- [dicee.static\\_preprocess\\_funcs](#), 150
- [dicee.trainer](#), 151
  - [dicee.trainer.dice\\_trainer](#), 151
  - [dicee.trainer.model\\_parallelism](#), 154
  - [dicee.trainer.torch\\_trainer](#), 154
  - [dicee.trainer.torch\\_trainer\\_ddp](#), 156

# Index

## Non-alphabetical

`__call__()` (*dicee.models.base\_model.IdentityClass method*), 59  
`__call__()` (*dicee.models.IdentityClass method*), 99, 110, 116  
`__call__()` (*dicee.trainer.dice\_trainer.EnsembleKGE method*), 152  
`__getattr__()` (*dicee.trainer.dice\_trainer.EnsembleKGE method*), 152  
`__getitem__()` (*dicee.AllvsAll method*), 197  
`__getitem__()` (*dicee.BPE\_NegativeSamplingDataset method*), 194  
`__getitem__()` (*dicee.dataset\_classes.AllvsAll method*), 33  
`__getitem__()` (*dicee.dataset\_classes.BPE\_NegativeSamplingDataset method*), 29  
`__getitem__()` (*dicee.dataset\_classes.KvsAll method*), 32  
`__getitem__()` (*dicee.dataset\_classes.KvsSampleDataset method*), 35  
`__getitem__()` (*dicee.dataset\_classes.MultiClassClassificationDataset method*), 31  
`__getitem__()` (*dicee.dataset\_classes.MultiLabelDataset method*), 30  
`__getitem__()` (*dicee.dataset\_classes.NegSampleDataset method*), 36  
`__getitem__()` (*dicee.dataset\_classes.OnevsAllDataset method*), 31  
`__getitem__()` (*dicee.dataset\_classes.OnevsSample method*), 34  
`__getitem__()` (*dicee.dataset\_classes.TriplePredictionDataset method*), 36  
`__getitem__()` (*dicee.KvsAll method*), 196  
`__getitem__()` (*dicee.KvsSampleDataset method*), 199  
`__getitem__()` (*dicee.MultiClassClassificationDataset method*), 195  
`__getitem__()` (*dicee.MultiLabelDataset method*), 194  
`__getitem__()` (*dicee.NegSampleDataset method*), 200  
`__getitem__()` (*dicee.OnevsAllDataset method*), 195  
`__getitem__()` (*dicee.OnevsSample method*), 198  
`__getitem__()` (*dicee.TriplePredictionDataset method*), 201  
`__iter__()` (*dicee.config.Namespace method*), 28  
`__iter__()` (*dicee.knowledge\_graph.KG method*), 46  
`__iter__()` (*dicee.trainer.dice\_trainer.EnsembleKGE method*), 152  
`__len__()` (*dicee.AllvsAll method*), 197  
`__len__()` (*dicee.BPE\_NegativeSamplingDataset method*), 194  
`__len__()` (*dicee.dataset\_classes.AllvsAll method*), 33  
`__len__()` (*dicee.dataset\_classes.BPE\_NegativeSamplingDataset method*), 29  
`__len__()` (*dicee.dataset\_classes.KvsAll method*), 32  
`__len__()` (*dicee.dataset\_classes.KvsSampleDataset method*), 35  
`__len__()` (*dicee.dataset\_classes.MultiClassClassificationDataset method*), 31  
`__len__()` (*dicee.dataset\_classes.MultiLabelDataset method*), 30  
`__len__()` (*dicee.dataset\_classes.NegSampleDataset method*), 35  
`__len__()` (*dicee.dataset\_classes.OnevsAllDataset method*), 31  
`__len__()` (*dicee.dataset\_classes.OnevsSample method*), 34  
`__len__()` (*dicee.dataset\_classes.TriplePredictionDataset method*), 36  
`__len__()` (*dicee.knowledge\_graph.KG method*), 46  
`__len__()` (*dicee.KvsAll method*), 196  
`__len__()` (*dicee.KvsSampleDataset method*), 199  
`__len__()` (*dicee.MultiClassClassificationDataset method*), 195  
`__len__()` (*dicee.MultiLabelDataset method*), 194  
`__len__()` (*dicee.NegSampleDataset method*), 200  
`__len__()` (*dicee.OnevsAllDataset method*), 195  
`__len__()` (*dicee.OnevsSample method*), 198  
`__len__()` (*dicee.trainer.dice\_trainer.EnsembleKGE method*), 152  
`__len__()` (*dicee.TriplePredictionDataset method*), 201  
`__str__()` (*dicee.KGE method*), 188  
`__str__()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 46  
`__str__()` (*dicee.trainer.dice\_trainer.EnsembleKGE method*), 152  
`__version__` (*in module dicee*), 205

## A

`AbstractCallback` (*class in dicee.abstracts*), 15  
`AbstractPPECallback` (*class in dicee.abstracts*), 17  
`AbstractTrainer` (*class in dicee.abstracts*), 12  
`AccumulateEpochLossCallback` (*class in dicee.callbacks*), 19  
`achieve_answer()` (*dicee.query\_generator.QueryGenerator method*), 136  
`achieve_answer()` (*dicee.QueryGenerator method*), 205  
`AConEx` (*class in dicee*), 171  
`AConEx` (*class in dicee.models*), 106  
`AConEx` (*class in dicee.models.complex*), 67

AConvO (class in *dicee*), 172  
 AConvO (class in *dicee.models*), 118  
 AConvO (class in *dicee.models.octonion*), 76  
 AConvQ (class in *dicee*), 172  
 AConvQ (class in *dicee.models*), 112  
 AConvQ (class in *dicee.models.quaternion*), 80  
 adaptive\_swa (*dicee.config.Namespace* attribute), 28  
 add\_new\_entity\_embeddings() (*dicee.abstracts.BaseInteractiveKGE* method), 15  
 add\_noise\_rate (*dicee.config.Namespace* attribute), 26  
 add\_noise\_rate (*dicee.knowledge\_graph.KG* attribute), 45  
 add\_noisy\_triples() (in module *dicee*), 186  
 add\_noisy\_triples() (in module *dicee.static\_funcs*), 149  
 add\_noisy\_triples\_into\_training() (*dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk.ReadFromDisk* method), 138  
 add\_noisy\_triples\_into\_training() (*dicee.read\_preprocess\_save\_load\_kg.ReadFromDisk* method), 144  
 add\_reciprocal (*dicee.knowledge\_graph.KG* attribute), 45  
 AllvsAll (class in *dicee*), 196  
 AllvsAll (class in *dicee.dataset\_classes*), 32  
 alphas (*dicee.abstracts.AbstractPPECallback* attribute), 17  
 alphas (*dicee.callbacks.ASWA* attribute), 23  
 analyse() (in module *dicee.analyse\_experiments*), 19  
 answer\_multi\_hop\_query() (*dicee.KGE* method), 191  
 answer\_multi\_hop\_query() (*dicee.knowledge\_graph\_embeddings.KGE* method), 49  
 app (in module *dicee.scripts.serve*), 146  
 apply\_coefficients() (*dicee.DeCaL* method), 168  
 apply\_coefficients() (*dicee.Keci* method), 164  
 apply\_coefficients() (*dicee.models.clifford.DeCaL* method), 64  
 apply\_coefficients() (*dicee.models.clifford.Keci* method), 61  
 apply\_coefficients() (*dicee.models.DeCaL* method), 124  
 apply\_coefficients() (*dicee.models.Keci* method), 120  
 apply\_reciprical\_or\_noise() (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 142  
 apply\_semantic\_constraint (*dicee.abstracts.BaseInteractiveKGE* attribute), 14  
 apply\_unit\_norm (*dicee.BaseKGE* attribute), 183  
 apply\_unit\_norm (*dicee.models.base\_model.BaseKGE* attribute), 57  
 apply\_unit\_norm (*dicee.models.BaseKGE* attribute), 97, 100, 103, 108, 114, 127, 130  
 args (*dicee.BaseKGE* attribute), 183  
 args (*dicee.DICE\_Trainer* attribute), 187  
 args (*dicee.evaluator.Evaluator* attribute), 41  
 args (*dicee.Execute* attribute), 192  
 args (*dicee.executer.Execute* attribute), 43  
 args (*dicee.models.base\_model.BaseKGE* attribute), 57  
 args (*dicee.models.base\_model.IdentityClass* attribute), 59  
 args (*dicee.models.BaseKGE* attribute), 96, 100, 103, 108, 114, 126, 130  
 args (*dicee.models.IdentityClass* attribute), 99, 110, 116  
 args (*dicee.models.pykeen\_models.PykeenKGE* attribute), 77  
 args (*dicee.models.PykeenKGE* attribute), 128  
 args (*dicee.PykeenKGE* attribute), 180  
 args (*dicee.trainer.DICE\_Trainer* attribute), 158  
 args (*dicee.trainer.dice\_trainer.DICE\_Trainer* attribute), 152  
 ASWA (class in *dicee.callbacks*), 22  
 aswa (*dicee.analyse\_experiments.Experiment* attribute), 18  
 attn (*dicee.models.transformers.Block* attribute), 87  
 attn\_dropout (*dicee.models.transformers.CausalSelfAttention* attribute), 86  
 attributes (*dicee.abstracts.AbstractTrainer* attribute), 12  
 available\_gpus (*dicee.trainer.torch\_trainer.xMP* attribute), 155

## B

backend (*dicee.config.Namespace* attribute), 26  
 backend (*dicee.knowledge\_graph.KG* attribute), 45  
 BaseInteractiveKGE (class in *dicee.abstracts*), 13  
 BaseKGE (class in *dicee*), 182  
 BaseKGE (class in *dicee.models*), 96, 99, 103, 107, 113, 126, 129  
 BaseKGE (class in *dicee.models.base\_model*), 56  
 BaseKGELightning (class in *dicee.models*), 90  
 BaseKGELightning (class in *dicee.models.base\_model*), 50  
 batch\_kronecker\_product() (*dicee.callbacks.KronE* static method), 25  
 batch\_size (*dicee.analyse\_experiments.Experiment* attribute), 18  
 batch\_size (*dicee.callbacks.PseudoLabellingCallback* attribute), 22



batch\_size (*dicee.config.Namespace* attribute), 26  
 batch\_size (*dicee.CVDataModule* attribute), 201  
 batch\_size (*dicee.dataset\_classes.CVDataModule* attribute), 37  
 bias (*dicee.models.transformers.GPTConfig* attribute), 88  
 bias (*dicee.models.transformers.LayerNorm* attribute), 85  
 Block (class in *dicee.models.transformers*), 87  
 block\_size (*dicee.BaseKGE* attribute), 184  
 block\_size (*dicee.config.Namespace* attribute), 28  
 block\_size (*dicee.dataset\_classes.MultiClassClassificationDataset* attribute), 30  
 block\_size (*dicee.models.base\_model.BaseKGE* attribute), 57  
 block\_size (*dicee.models.BaseKGE* attribute), 97, 100, 104, 109, 115, 127, 131  
 block\_size (*dicee.models.transformers.GPTConfig* attribute), 88  
 block\_size (*dicee.MultiClassClassificationDataset* attribute), 195  
 bn\_conv1 (*dicee.AConvQ* attribute), 173  
 bn\_conv1 (*dicee.ConvQ* attribute), 173  
 bn\_conv1 (*dicee.models.AConvQ* attribute), 113  
 bn\_conv1 (*dicee.models.ConvQ* attribute), 112  
 bn\_conv1 (*dicee.models.quaternion.AConvQ* attribute), 80  
 bn\_conv1 (*dicee.models.quaternion.ConvQ* attribute), 79  
 bn\_conv2 (*dicee.AConvQ* attribute), 173  
 bn\_conv2 (*dicee.ConvQ* attribute), 173  
 bn\_conv2 (*dicee.models.AConvQ* attribute), 113  
 bn\_conv2 (*dicee.models.ConvQ* attribute), 112  
 bn\_conv2 (*dicee.models.quaternion.AConvQ* attribute), 80  
 bn\_conv2 (*dicee.models.quaternion.ConvQ* attribute), 80  
 bn\_conv2d (*dicee.AConEx* attribute), 172  
 bn\_conv2d (*dicee.AConvO* attribute), 172  
 bn\_conv2d (*dicee.ConEx* attribute), 175  
 bn\_conv2d (*dicee.ConvO* attribute), 174  
 bn\_conv2d (*dicee.models.AConEx* attribute), 106  
 bn\_conv2d (*dicee.models.AConvO* attribute), 119  
 bn\_conv2d (*dicee.models.complex.AConEx* attribute), 67  
 bn\_conv2d (*dicee.models.complex.ConEx* attribute), 67  
 bn\_conv2d (*dicee.models.ConEx* attribute), 105  
 bn\_conv2d (*dicee.models.ConvO* attribute), 118  
 bn\_conv2d (*dicee.models.octonion.AConvO* attribute), 76  
 bn\_conv2d (*dicee.models.octonion.ConvO* attribute), 75  
 BPE\_NegativeSamplingDataset (class in *dicee*), 193  
 BPE\_NegativeSamplingDataset (class in *dicee.dataset\_classes*), 29  
 build\_chain\_funcs () (*dicee.models.FMult2* method), 133  
 build\_chain\_funcs () (*dicee.models.function\_space.FMult2* method), 71  
 build\_func () (*dicee.models.FMult2* method), 133  
 build\_func () (*dicee.models.function\_space.FMult2* method), 71  
 Byte (class in *dicee*), 180  
 Byte (class in *dicee.models.transformers*), 83  
 byte\_pair\_encoding (*dicee.analyse\_experiments.Experiment* attribute), 18  
 byte\_pair\_encoding (*dicee.BaseKGE* attribute), 184  
 byte\_pair\_encoding (*dicee.config.Namespace* attribute), 28  
 byte\_pair\_encoding (*dicee.knowledge\_graph.KG* attribute), 45  
 byte\_pair\_encoding (*dicee.models.base\_model.BaseKGE* attribute), 57  
 byte\_pair\_encoding (*dicee.models.BaseKGE* attribute), 97, 100, 104, 109, 115, 127, 130

## C

c\_attn (*dicee.models.transformers.CausalSelfAttention* attribute), 86  
 c\_fc (*dicee.models.transformers.MLP* attribute), 87  
 c\_proj (*dicee.models.transformers.CausalSelfAttention* attribute), 86  
 c\_proj (*dicee.models.transformers.MLP* attribute), 87  
 callbacks (*dicee.abstracts.AbstractTrainer* attribute), 12  
 callbacks (*dicee.analyse\_experiments.Experiment* attribute), 18  
 callbacks (*dicee.config.Namespace* attribute), 26  
 callbacks (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 157  
 CausalSelfAttention (class in *dicee.models.transformers*), 85  
 chain\_func () (*dicee.models.FMult* method), 132  
 chain\_func () (*dicee.models.function\_space.FMult* method), 70  
 chain\_func () (*dicee.models.function\_space.GFMult* method), 71  
 chain\_func () (*dicee.models.GFMult* method), 132  
 cl\_pqr () (*dicee.DeCaL* method), 167

`cl_pqr()` (*dicee.models.clifford.DeCaL method*), 64  
`cl_pqr()` (*dicee.models.DeCaL method*), 123  
`clifford_multiplication()` (*dicee.Keci method*), 164  
`clifford_multiplication()` (*dicee.models.clifford.Keci method*), 61  
`clifford_multiplication()` (*dicee.models.Keci method*), 120  
`collate_fn` (*dicee.AllvsAll attribute*), 197  
`collate_fn` (*dicee.dataset\_classes.AllvsAll attribute*), 32  
`collate_fn` (*dicee.dataset\_classes.KvsAll attribute*), 32  
`collate_fn` (*dicee.dataset\_classes.KvsSampleDataset attribute*), 35  
`collate_fn` (*dicee.dataset\_classes.MultiClassClassificationDataset attribute*), 30  
`collate_fn` (*dicee.dataset\_classes.MultiLabelDataset attribute*), 30  
`collate_fn` (*dicee.dataset\_classes.OnevsAllDataset attribute*), 31  
`collate_fn` (*dicee.dataset\_classes.OnevsSample attribute*), 33, 34  
`collate_fn` (*dicee.KvsAll attribute*), 196  
`collate_fn` (*dicee.KvsSampleDataset attribute*), 199  
`collate_fn` (*dicee.MultiClassClassificationDataset attribute*), 195  
`collate_fn` (*dicee.MultiLabelDataset attribute*), 194  
`collate_fn` (*dicee.OnevsAllDataset attribute*), 195  
`collate_fn` (*dicee.OnevsSample attribute*), 198  
`collate_fn()` (*dicee.BPE\_NegativeSamplingDataset method*), 194  
`collate_fn()` (*dicee.dataset\_classes.BPE\_NegativeSamplingDataset method*), 29  
`collate_fn()` (*dicee.dataset\_classes.TriplePredictionDataset method*), 36  
`collate_fn()` (*dicee.TriplePredictionDataset method*), 201  
`collection_name` (*dicee.scripts.serve.NeuralSearcher attribute*), 146  
`comp_func()` (*dicee.LFMult method*), 179  
`comp_func()` (*dicee.models.function\_space.LFMult method*), 73  
`comp_func()` (*dicee.models.LFMult method*), 134  
`Complex` (*class in dicee*), 170  
`Complex` (*class in dicee.models*), 106  
`Complex` (*class in dicee.models.complex*), 67  
`compute_convergence()` (*in module dicee.callbacks*), 22  
`compute_func()` (*dicee.models.FMult method*), 132  
`compute_func()` (*dicee.models.FMult2 method*), 133  
`compute_func()` (*dicee.models.function\_space.FMult method*), 70  
`compute_func()` (*dicee.models.function\_space.FMult2 method*), 71  
`compute_func()` (*dicee.models.function\_space.GFMult method*), 71  
`compute_func()` (*dicee.models.GFMult method*), 132  
`compute_mrr()` (*dicee.callbacks.ASWA static method*), 23  
`compute_sigma_pp()` (*dicee.DeCaL method*), 168  
`compute_sigma_pp()` (*dicee.Keci method*), 163  
`compute_sigma_pp()` (*dicee.models.clifford.DeCaL method*), 65  
`compute_sigma_pp()` (*dicee.models.clifford.Keci method*), 60  
`compute_sigma_pp()` (*dicee.models.DeCaL method*), 124  
`compute_sigma_pp()` (*dicee.models.Keci method*), 120  
`compute_sigma_pq()` (*dicee.DeCaL method*), 169  
`compute_sigma_pq()` (*dicee.Keci method*), 164  
`compute_sigma_pq()` (*dicee.models.clifford.DeCaL method*), 66  
`compute_sigma_pq()` (*dicee.models.clifford.Keci method*), 61  
`compute_sigma_pq()` (*dicee.models.DeCaL method*), 125  
`compute_sigma_pq()` (*dicee.models.Keci method*), 120  
`compute_sigma_pr()` (*dicee.DeCaL method*), 169  
`compute_sigma_pr()` (*dicee.models.clifford.DeCaL method*), 66  
`compute_sigma_pr()` (*dicee.models.DeCaL method*), 125  
`compute_sigma_qq()` (*dicee.DeCaL method*), 168  
`compute_sigma_qq()` (*dicee.Keci method*), 164  
`compute_sigma_qq()` (*dicee.models.clifford.DeCaL method*), 65  
`compute_sigma_qq()` (*dicee.models.clifford.Keci method*), 60  
`compute_sigma_qq()` (*dicee.models.DeCaL method*), 125  
`compute_sigma_qq()` (*dicee.models.Keci method*), 120  
`compute_sigma_qr()` (*dicee.DeCaL method*), 169  
`compute_sigma_qr()` (*dicee.models.clifford.DeCaL method*), 66  
`compute_sigma_qr()` (*dicee.models.DeCaL method*), 126  
`compute_sigma_rr()` (*dicee.DeCaL method*), 169  
`compute_sigma_rr()` (*dicee.models.clifford.DeCaL method*), 65  
`compute_sigma_rr()` (*dicee.models.DeCaL method*), 125  
`compute_sigmas_multivect()` (*dicee.DeCaL method*), 167  
`compute_sigmas_multivect()` (*dicee.models.clifford.DeCaL method*), 64  
`compute_sigmas_multivect()` (*dicee.models.DeCaL method*), 124

`compute_sigmas_single()` (*dicee.DeCaL method*), 167  
`compute_sigmas_single()` (*dicee.models.clifford.DeCaL method*), 64  
`compute_sigmas_single()` (*dicee.models.DeCaL method*), 123  
`ConEx` (*class in dicee*), 175  
`ConEx` (*class in dicee.models*), 105  
`ConEx` (*class in dicee.models.complex*), 66  
`config` (*dicee.Byte attribute*), 181  
`config` (*dicee.models.transformers.Byte attribute*), 83  
`config` (*dicee.models.transformers.GPT attribute*), 88  
`configs` (*dicee.abstracts.BaseInteractiveKGE attribute*), 14  
`configure_optimizers()` (*dicee.models.base\_model.BaseKGELightning method*), 55  
`configure_optimizers()` (*dicee.models.BaseKGELightning method*), 94  
`configure_optimizers()` (*dicee.models.transformers.GPT method*), 89  
`construct_batch_selected_cl_multivector()` (*dicee.Keci method*), 165  
`construct_batch_selected_cl_multivector()` (*dicee.models.clifford.Keci method*), 62  
`construct_batch_selected_cl_multivector()` (*dicee.models.Keci method*), 121  
`construct_cl_multivector()` (*dicee.DeCaL method*), 168  
`construct_cl_multivector()` (*dicee.Keci method*), 164  
`construct_cl_multivector()` (*dicee.models.clifford.DeCaL method*), 64  
`construct_cl_multivector()` (*dicee.models.clifford.Keci method*), 61  
`construct_cl_multivector()` (*dicee.models.DeCaL method*), 124  
`construct_cl_multivector()` (*dicee.models.Keci method*), 121  
`construct_dataset()` (*in module dicee*), 193  
`construct_dataset()` (*in module dicee.dataset\_classes*), 29  
`construct_ensemble` (*dicee.abstracts.BaseInteractiveKGE attribute*), 14  
`construct_graph()` (*dicee.query\_generator.QueryGenerator method*), 136  
`construct_graph()` (*dicee.QueryGenerator method*), 205  
`construct_input_and_output()` (*dicee.abstracts.BaseInteractiveKGE method*), 15  
`construct_multi_coeff()` (*dicee.LFMMult method*), 178  
`construct_multi_coeff()` (*dicee.models.function\_space.LFMMult method*), 72  
`construct_multi_coeff()` (*dicee.models.LFMMult method*), 134  
`continual_learning` (*dicee.config.Namespace attribute*), 28  
`continual_start()` (*dicee.DICE\_Trainer method*), 187  
`continual_start()` (*dicee.executer.ContinuousExecute method*), 44  
`continual_start()` (*dicee.trainer.DICE\_Trainer method*), 158  
`continual_start()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 153  
`continual_training_setup_executor()` (*in module dicee*), 186  
`continual_training_setup_executor()` (*in module dicee.static\_funcs*), 149  
`ContinuousExecute` (*class in dicee.executer*), 44  
`conv2d` (*dicee.AConEx attribute*), 171  
`conv2d` (*dicee.AConvO attribute*), 172  
`conv2d` (*dicee.AConvQ attribute*), 173  
`conv2d` (*dicee.ConEx attribute*), 175  
`conv2d` (*dicee.ConvO attribute*), 174  
`conv2d` (*dicee.ConvQ attribute*), 173  
`conv2d` (*dicee.models.AConEx attribute*), 106  
`conv2d` (*dicee.models.AConvO attribute*), 118  
`conv2d` (*dicee.models.AConvQ attribute*), 113  
`conv2d` (*dicee.models.complex.AConEx attribute*), 67  
`conv2d` (*dicee.models.complex.ConEx attribute*), 66  
`conv2d` (*dicee.models.ConEx attribute*), 105  
`conv2d` (*dicee.models.ConvO attribute*), 118  
`conv2d` (*dicee.models.ConvQ attribute*), 112  
`conv2d` (*dicee.models.octonion.AConvO attribute*), 76  
`conv2d` (*dicee.models.octonion.ConvO attribute*), 75  
`conv2d` (*dicee.models.quaternion.AConvQ attribute*), 80  
`conv2d` (*dicee.models.quaternion.ConvQ attribute*), 79  
`ConvO` (*class in dicee*), 174  
`ConvO` (*class in dicee.models*), 117  
`ConvO` (*class in dicee.models.octonion*), 74  
`ConvQ` (*class in dicee*), 173  
`ConvQ` (*class in dicee.models*), 112  
`ConvQ` (*class in dicee.models.quaternion*), 79  
`create_constraints()` (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 142  
`create_constraints()` (*in module dicee.static\_preprocess\_funcs*), 151  
`create_experiment_folder()` (*in module dicee*), 186  
`create_experiment_folder()` (*in module dicee.static\_funcs*), 149  
`create_random_data()` (*dicee.callbacks.PseudoLabellingCallback method*), 22

`create_recipriocal_triples()` (in module *dicee*), 185  
`create_recipriocal_triples()` (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 143  
`create_recipriocal_triples()` (in module *dicee.static\_funcs*), 148  
`create_vector_database()` (*dicee.KGE* method), 188  
`create_vector_database()` (*dicee.knowledge\_graph\_embeddings.KGE* method), 46  
`crop_block_size()` (*dicee.models.transformers.GPT* method), 89  
`ctx` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 157  
`CVDataModule` (class in *dicee*), 201  
`CVDataModule` (class in *dicee.dataset\_classes*), 36

## D

`data_module` (*dicee.callbacks.PseudoLabellingCallback* attribute), 22  
`dataset_dir` (*dicee.config.Namespace* attribute), 26  
`dataset_dir` (*dicee.knowledge\_graph.KG* attribute), 45  
`dataset_sanity_checking()` (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 143  
`DeCaL` (class in *dicee*), 166  
`DeCaL` (class in *dicee.models*), 122  
`DeCaL` (class in *dicee.models.clifford*), 63  
`decide()` (*dicee.callbacks.ASWA* method), 23  
`degree` (*dicee.LFMMult* attribute), 178  
`degree` (*dicee.models.function\_space.LFMMult* attribute), 72  
`degree` (*dicee.models.LFMMult* attribute), 134  
`deploy()` (*dicee.KGE* method), 191  
`deploy()` (*dicee.knowledge\_graph\_embeddings.KGE* method), 50  
`deploy_head_entity_prediction()` (in module *dicee*), 186  
`deploy_head_entity_prediction()` (in module *dicee.static\_funcs*), 149  
`deploy_relation_prediction()` (in module *dicee*), 186  
`deploy_relation_prediction()` (in module *dicee.static\_funcs*), 149  
`deploy_tail_entity_prediction()` (in module *dicee*), 186  
`deploy_tail_entity_prediction()` (in module *dicee.static\_funcs*), 149  
`deploy_triple_prediction()` (in module *dicee*), 186  
`deploy_triple_prediction()` (in module *dicee.static\_funcs*), 149  
`dept_read_preprocess_index_serialize_data()` (*dicee.Execute* method), 192  
`dept_read_preprocess_index_serialize_data()` (*dicee.executer.Execute* method), 43  
`describe()` (*dicee.knowledge\_graph.KG* method), 46  
`description_of_input` (*dicee.knowledge\_graph.KG* attribute), 46  
`DICE_Trainer` (class in *dicee*), 186  
`DICE_Trainer` (class in *dicee.trainer*), 158  
`DICE_Trainer` (class in *dicee.trainer.dice\_trainer*), 152  
`dicee`  
    module, 12  
`dicee.__main__`  
    module, 12  
`dicee.abstracts`  
    module, 12  
`dicee.analyse_experiments`  
    module, 17  
`dicee.callbacks`  
    module, 19  
`dicee.config`  
    module, 25  
`dicee.dataset_classes`  
    module, 28  
`dicee.eval_static_funcs`  
    module, 40  
`dicee.evaluator`  
    module, 41  
`dicee.executer`  
    module, 42  
`dicee.knowledge_graph`  
    module, 45  
`dicee.knowledge_graph_embeddings`  
    module, 46  
`dicee.models`  
    module, 50  
`dicee.models.base_model`  
    module, 50

- dicee.models.clifford
  - module, [59](#)
- dicee.models.complex
  - module, [66](#)
- dicee.models.dualE
  - module, [69](#)
- dicee.models.function\_space
  - module, [70](#)
- dicee.models.octonion
  - module, [73](#)
- dicee.models.pykeen\_models
  - module, [76](#)
- dicee.models.quaternion
  - module, [77](#)
- dicee.models.real
  - module, [81](#)
- dicee.models.static\_funcs
  - module, [82](#)
- dicee.models.transformers
  - module, [82](#)
- dicee.query\_generator
  - module, [136](#)
- dicee.read\_preprocess\_save\_load\_kg
  - module, [137](#)
- dicee.read\_preprocess\_save\_load\_kg.preprocess
  - module, [137](#)
- dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk
  - module, [138](#)
- dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk
  - module, [139](#)
- dicee.read\_preprocess\_save\_load\_kg.util
  - module, [139](#)
- dicee.sanity\_checkers
  - module, [144](#)
- dicee.scripts
  - module, [145](#)
- dicee.scripts.index
  - module, [145](#)
- dicee.scripts.run
  - module, [145](#)
- dicee.scripts.serve
  - module, [145](#)
- dicee.static\_funcs
  - module, [146](#)
- dicee.static\_funcs\_training
  - module, [150](#)
- dicee.static\_preprocess\_funcs
  - module, [150](#)
- dicee.trainer
  - module, [151](#)
- dicee.trainer.dice\_trainer
  - module, [151](#)
- dicee.trainer.model\_parallelism
  - module, [154](#)
- dicee.trainer.torch\_trainer
  - module, [154](#)
- dicee.trainer.torch\_trainer\_ddp
  - module, [156](#)
- discrete\_points (*dicee.models.FMult2 attribute*), [133](#)
- discrete\_points (*dicee.models.function\_space.FMult2 attribute*), [71](#)
- dist\_func (*dicee.models.Pyke attribute*), [102](#)
- dist\_func (*dicee.models.real.Pyke attribute*), [82](#)
- dist\_func (*dicee.Pyke attribute*), [162](#)
- DistMult (*class in dicee*), [162](#)
- DistMult (*class in dicee.models*), [101](#)
- DistMult (*class in dicee.models.real*), [81](#)
- download\_file() (*in module dicee*), [186](#)
- download\_file() (*in module dicee.static\_funcs*), [149](#)

`download_files_from_url()` (in module *dicee*), 186  
`download_files_from_url()` (in module *dicee.static\_funcs*), 149  
`download_pretrained_model()` (in module *dicee*), 186  
`download_pretrained_model()` (in module *dicee.static\_funcs*), 149  
`dropout` (*dicee.models.transformers.CausalSelfAttention* attribute), 86  
`dropout` (*dicee.models.transformers.GPTConfig* attribute), 88  
`dropout` (*dicee.models.transformers.MLP* attribute), 87  
`DualE` (class in *dicee*), 169  
`DualE` (class in *dicee.models*), 135  
`DualE` (class in *dicee.models.dualE*), 69  
`dummy_eval()` (*dicee.evaluator.Evaluator* method), 42  
`dummy_id` (*dicee.knowledge\_graph.KG* attribute), 46  
`during_training` (*dicee.evaluator.Evaluator* attribute), 42

## E

`ee_vocab` (*dicee.evaluator.Evaluator* attribute), 41  
`efficient_zero_grad()` (in module *dicee.static\_funcs\_training*), 150  
`embedding_dim` (*dicee.analyse\_experiments.Experiment* attribute), 18  
`embedding_dim` (*dicee.BaseKGE* attribute), 183  
`embedding_dim` (*dicee.config.Namespace* attribute), 26  
`embedding_dim` (*dicee.models.base\_model.BaseKGE* attribute), 57  
`embedding_dim` (*dicee.models.BaseKGE* attribute), 96, 100, 103, 108, 114, 126, 130  
`enable_log` (in module *dicee.static\_preprocess\_funcs*), 151  
`enc` (*dicee.knowledge\_graph.KG* attribute), 45  
`end()` (*dicee.Execute* method), 193  
`end()` (*dicee.executer.Execute* method), 43  
`EnsembleKGE` (class in *dicee.trainer.dice\_trainer*), 152  
`ent2id` (*dicee.query\_generator.QueryGenerator* attribute), 136  
`ent2id` (*dicee.QueryGenerator* attribute), 205  
`ent_in` (*dicee.query\_generator.QueryGenerator* attribute), 136  
`ent_in` (*dicee.QueryGenerator* attribute), 205  
`ent_out` (*dicee.query\_generator.QueryGenerator* attribute), 136  
`ent_out` (*dicee.QueryGenerator* attribute), 205  
`entities_str` (*dicee.knowledge\_graph.KG* property), 46  
`entity_embeddings` (*dicee.AConvQ* attribute), 173  
`entity_embeddings` (*dicee.ConvQ* attribute), 173  
`entity_embeddings` (*dicee.DeCaL* attribute), 166  
`entity_embeddings` (*dicee.DualE* attribute), 170  
`entity_embeddings` (*dicee.LFMult* attribute), 178  
`entity_embeddings` (*dicee.models.AConvQ* attribute), 113  
`entity_embeddings` (*dicee.models.clifford.DeCaL* attribute), 63  
`entity_embeddings` (*dicee.models.ConvQ* attribute), 112  
`entity_embeddings` (*dicee.models.DeCaL* attribute), 123  
`entity_embeddings` (*dicee.models.DualE* attribute), 135  
`entity_embeddings` (*dicee.models.dualE.DualE* attribute), 69  
`entity_embeddings` (*dicee.models.FMult* attribute), 132  
`entity_embeddings` (*dicee.models.FMult2* attribute), 133  
`entity_embeddings` (*dicee.models.function\_space.FMult* attribute), 70  
`entity_embeddings` (*dicee.models.function\_space.FMult2* attribute), 71  
`entity_embeddings` (*dicee.models.function\_space.GFMult* attribute), 71  
`entity_embeddings` (*dicee.models.function\_space.LFMult* attribute), 72  
`entity_embeddings` (*dicee.models.function\_space.LFMult1* attribute), 72  
`entity_embeddings` (*dicee.models.GFMult* attribute), 132  
`entity_embeddings` (*dicee.models.LFMult* attribute), 133  
`entity_embeddings` (*dicee.models.LFMult1* attribute), 133  
`entity_embeddings` (*dicee.models.pykeen\_models.PykeenKGE* attribute), 77  
`entity_embeddings` (*dicee.models.PykeenKGE* attribute), 128  
`entity_embeddings` (*dicee.models.quaternion.AConvQ* attribute), 80  
`entity_embeddings` (*dicee.models.quaternion.ConvQ* attribute), 79  
`entity_embeddings` (*dicee.PykeenKGE* attribute), 180  
`entity_to_idx` (*dicee.knowledge\_graph.KG* attribute), 45  
`epoch_count` (*dicee.abstracts.AbstractPPECallback* attribute), 17  
`epoch_count` (*dicee.callbacks.ASWA* attribute), 22  
`epoch_counter` (*dicee.callbacks.Eval* attribute), 23  
`epoch_counter` (*dicee.callbacks.KGESaveCallback* attribute), 21  
`epoch_ratio` (*dicee.callbacks.Eval* attribute), 23  
`er_vocab` (*dicee.evaluator.Evaluator* attribute), 41

`estimate_mfu()` (*dicee.models.transformers.GPT method*), 89  
`estimate_q()` (*in module dicee.callbacks*), 22  
`Eval` (*class in dicee.callbacks*), 23  
`eval()` (*dicee.evaluator.Evaluator method*), 42  
`eval_lp_performance()` (*dicee.KGE method*), 188  
`eval_lp_performance()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 47  
`eval_model` (*dicee.config.Namespace attribute*), 27  
`eval_model` (*dicee.knowledge\_graph.KG attribute*), 45  
`eval_rank_of_head_and_tail_byte_pair_encoded_entity()` (*dicee.evaluator.Evaluator method*), 42  
`eval_rank_of_head_and_tail_entity()` (*dicee.evaluator.Evaluator method*), 42  
`eval_with_bpe_vs_all()` (*dicee.evaluator.Evaluator method*), 42  
`eval_with_byte()` (*dicee.evaluator.Evaluator method*), 42  
`eval_with_data()` (*dicee.evaluator.Evaluator method*), 42  
`eval_with_vs_all()` (*dicee.evaluator.Evaluator method*), 42  
`evaluate()` (*in module dicee*), 186  
`evaluate()` (*in module dicee.static\_funcs*), 149  
`evaluate_bpe_lp()` (*in module dicee.static\_funcs\_training*), 150  
`evaluate_link_prediction_performance()` (*in module dicee.eval\_static\_funcs*), 40  
`evaluate_link_prediction_performance_with_bpe()` (*in module dicee.eval\_static\_funcs*), 41  
`evaluate_link_prediction_performance_with_bpe_reciprocals()` (*in module dicee.eval\_static\_funcs*), 41  
`evaluate_link_prediction_performance_with_reciprocals()` (*in module dicee.eval\_static\_funcs*), 41  
`evaluate_lp()` (*dicee.evaluator.Evaluator method*), 42  
`evaluate_lp()` (*in module dicee.static\_funcs\_training*), 150  
`evaluate_lp_bpe_k_vs_all()` (*dicee.evaluator.Evaluator method*), 42  
`evaluate_lp_bpe_k_vs_all()` (*in module dicee.eval\_static\_funcs*), 41  
`evaluate_lp_k_vs_all()` (*dicee.evaluator.Evaluator method*), 42  
`evaluate_lp_with_byte()` (*dicee.evaluator.Evaluator method*), 42  
`Evaluator` (*class in dicee.evaluator*), 41  
`evaluator` (*dicee.DICE\_Trainer attribute*), 187  
`evaluator` (*dicee.Execute attribute*), 192  
`evaluator` (*dicee.executor.Execute attribute*), 43  
`evaluator` (*dicee.trainer.DICE\_Trainer attribute*), 158  
`evaluator` (*dicee.trainer.dice\_trainer.DICE\_Trainer attribute*), 153  
`every_x_epoch` (*dicee.callbacks.KGESaveCallback attribute*), 21  
`Execute` (*class in dicee*), 192  
`Execute` (*class in dicee.executor*), 43  
`exists()` (*dicee.knowledge\_graph.KG method*), 46  
`Experiment` (*class in dicee.analyse\_experiments*), 18  
`explicit` (*dicee.models.QMult attribute*), 111  
`explicit` (*dicee.models.quaternion.QMult attribute*), 78  
`explicit` (*dicee.QMult attribute*), 176  
`exponential_function()` (*in module dicee*), 186  
`exponential_function()` (*in module dicee.static\_funcs*), 149  
`extract_input_outputs()` (*dicee.trainer.model\_parallelism.MP method*), 154  
`extract_input_outputs()` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer method*), 157  
`extract_input_outputs_set_device()` (*dicee.trainer.torch\_trainer.TorchTrainer method*), 156  
`extract_input_outputs_set_device()` (*dicee.trainer.torch\_trainer.xMP method*), 155

## F

`f` (*dicee.callbacks.KronE attribute*), 25  
`fc1` (*dicee.AConEx attribute*), 171  
`fc1` (*dicee.AConvO attribute*), 172  
`fc1` (*dicee.AConvQ attribute*), 173  
`fc1` (*dicee.ConEx attribute*), 175  
`fc1` (*dicee.ConvO attribute*), 174  
`fc1` (*dicee.ConvQ attribute*), 173  
`fc1` (*dicee.models.AConEx attribute*), 106  
`fc1` (*dicee.models.AConvO attribute*), 118  
`fc1` (*dicee.models.AConvQ attribute*), 113  
`fc1` (*dicee.models.complex.AConEx attribute*), 67  
`fc1` (*dicee.models.complex.ConEx attribute*), 67  
`fc1` (*dicee.models.ConEx attribute*), 105  
`fc1` (*dicee.models.ConvO attribute*), 118  
`fc1` (*dicee.models.ConvQ attribute*), 112  
`fc1` (*dicee.models.octonion.AConvO attribute*), 76  
`fc1` (*dicee.models.octonion.ConvO attribute*), 75  
`fc1` (*dicee.models.quaternion.AConvQ attribute*), 80



`fc1` (*dicee.models.quaternion.ConvQ attribute*), 79  
`fc_num_input` (*dicee.AConEx attribute*), 171  
`fc_num_input` (*dicee.AConvO attribute*), 172  
`fc_num_input` (*dicee.AConvQ attribute*), 173  
`fc_num_input` (*dicee.ConEx attribute*), 175  
`fc_num_input` (*dicee.ConvO attribute*), 174  
`fc_num_input` (*dicee.ConvQ attribute*), 173  
`fc_num_input` (*dicee.models.AConEx attribute*), 106  
`fc_num_input` (*dicee.models.AConvO attribute*), 118  
`fc_num_input` (*dicee.models.AConvQ attribute*), 113  
`fc_num_input` (*dicee.models.complex.AConEx attribute*), 67  
`fc_num_input` (*dicee.models.complex.ConEx attribute*), 67  
`fc_num_input` (*dicee.models.ConEx attribute*), 105  
`fc_num_input` (*dicee.models.ConvO attribute*), 118  
`fc_num_input` (*dicee.models.ConvQ attribute*), 112  
`fc_num_input` (*dicee.models.octonion.AConvO attribute*), 76  
`fc_num_input` (*dicee.models.octonion.ConvO attribute*), 75  
`fc_num_input` (*dicee.models.quaternion.AConvQ attribute*), 80  
`fc_num_input` (*dicee.models.quaternion.ConvQ attribute*), 79  
`feature_map_dropout` (*dicee.AConEx attribute*), 172  
`feature_map_dropout` (*dicee.AConvO attribute*), 172  
`feature_map_dropout` (*dicee.AConvQ attribute*), 173  
`feature_map_dropout` (*dicee.ConEx attribute*), 175  
`feature_map_dropout` (*dicee.ConvO attribute*), 174  
`feature_map_dropout` (*dicee.ConvQ attribute*), 173  
`feature_map_dropout` (*dicee.models.AConEx attribute*), 106  
`feature_map_dropout` (*dicee.models.AConvO attribute*), 119  
`feature_map_dropout` (*dicee.models.AConvQ attribute*), 113  
`feature_map_dropout` (*dicee.models.complex.AConEx attribute*), 67  
`feature_map_dropout` (*dicee.models.complex.ConEx attribute*), 67  
`feature_map_dropout` (*dicee.models.ConEx attribute*), 105  
`feature_map_dropout` (*dicee.models.ConvO attribute*), 118  
`feature_map_dropout` (*dicee.models.ConvQ attribute*), 112  
`feature_map_dropout` (*dicee.models.octonion.AConvO attribute*), 76  
`feature_map_dropout` (*dicee.models.octonion.ConvO attribute*), 75  
`feature_map_dropout` (*dicee.models.quaternion.AConvQ attribute*), 80  
`feature_map_dropout` (*dicee.models.quaternion.ConvQ attribute*), 80  
`feature_map_dropout_rate` (*dicee.BaseKGE attribute*), 183  
`feature_map_dropout_rate` (*dicee.config.Namespace attribute*), 28  
`feature_map_dropout_rate` (*dicee.models.base\_model.BaseKGE attribute*), 57  
`feature_map_dropout_rate` (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130  
`fill_query()` (*dicee.query\_generator.QueryGenerator method*), 136  
`fill_query()` (*dicee.QueryGenerator method*), 205  
`find_missing_triples()` (*dicee.KGE method*), 191  
`find_missing_triples()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 50  
`fit()` (*dicee.trainer.model\_parallelism.MP method*), 154  
`fit()` (*dicee.trainer.torch\_trainer\_ddp.TorchDDPTrainer method*), 157  
`fit()` (*dicee.trainer.torch\_trainer.TorchTrainer method*), 156  
`fit()` (*dicee.trainer.torch\_trainer.xMP method*), 155  
`flash` (*dicee.models.transformers.CausalSelfAttention attribute*), 86  
`FMult` (*class in dicee.models*), 131  
`FMult` (*class in dicee.models.function\_space*), 70  
`FMult2` (*class in dicee.models*), 132  
`FMult2` (*class in dicee.models.function\_space*), 71  
`form_of_labelling` (*dicee.DICE\_Trainer attribute*), 187  
`form_of_labelling` (*dicee.trainer.DICE\_Trainer attribute*), 158  
`form_of_labelling` (*dicee.trainer.dice\_trainer.DICE\_Trainer attribute*), 153  
`forward()` (*dicee.BaseKGE method*), 184  
`forward()` (*dicee.BytE method*), 181  
`forward()` (*dicee.models.base\_model.BaseKGE method*), 58  
`forward()` (*dicee.models.base\_model.IdentityClass static method*), 59  
`forward()` (*dicee.models.BaseKGE method*), 98, 101, 104, 109, 115, 128, 131  
`forward()` (*dicee.models.IdentityClass static method*), 99, 110, 116  
`forward()` (*dicee.models.transformers.Block method*), 88  
`forward()` (*dicee.models.transformers.BytE method*), 84  
`forward()` (*dicee.models.transformers.CausalSelfAttention method*), 86  
`forward()` (*dicee.models.transformers.GPT method*), 89  
`forward()` (*dicee.models.transformers.LayerNorm method*), 85



`forward()` (*dicee.models.transformers.MLP method*), 87  
`forward_backward_update()` (*dicee.trainer.torch\_trainer.TorchTrainer method*), 156  
`forward_backward_update()` (*dicee.trainer.torch\_trainer.xMP method*), 155  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.BaseKGE method*), 184  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.base\_model.BaseKGE method*), 57  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.BaseKGE method*), 97, 100, 104, 109, 115, 127, 131  
`forward_byte_pair_encoded_triple()` (*dicee.BaseKGE method*), 184  
`forward_byte_pair_encoded_triple()` (*dicee.models.base\_model.BaseKGE method*), 58  
`forward_byte_pair_encoded_triple()` (*dicee.models.BaseKGE method*), 97, 101, 104, 109, 115, 127, 131  
`forward_k_vs_all()` (*dicee.AConEx method*), 172  
`forward_k_vs_all()` (*dicee.AConvO method*), 172  
`forward_k_vs_all()` (*dicee.AConvQ method*), 173  
`forward_k_vs_all()` (*dicee.BaseKGE method*), 184  
`forward_k_vs_all()` (*dicee.ComplEx method*), 171  
`forward_k_vs_all()` (*dicee.ConEx method*), 175  
`forward_k_vs_all()` (*dicee.ConvO method*), 175  
`forward_k_vs_all()` (*dicee.ConvQ method*), 173  
`forward_k_vs_all()` (*dicee.DeCaL method*), 167  
`forward_k_vs_all()` (*dicee.DistMult method*), 162  
`forward_k_vs_all()` (*dicee.DualE method*), 170  
`forward_k_vs_all()` (*dicee.Keci method*), 165  
`forward_k_vs_all()` (*dicee.models.AConEx method*), 106  
`forward_k_vs_all()` (*dicee.models.AConvO method*), 119  
`forward_k_vs_all()` (*dicee.models.AConvQ method*), 113  
`forward_k_vs_all()` (*dicee.models.base\_model.BaseKGE method*), 58  
`forward_k_vs_all()` (*dicee.models.BaseKGE method*), 98, 101, 105, 109, 115, 128, 131  
`forward_k_vs_all()` (*dicee.models.clifford.DeCaL method*), 64  
`forward_k_vs_all()` (*dicee.models.clifford.Keci method*), 62  
`forward_k_vs_all()` (*dicee.models.ComplEx method*), 107  
`forward_k_vs_all()` (*dicee.models.complex.AConEx method*), 67  
`forward_k_vs_all()` (*dicee.models.complex.ComplEx method*), 68  
`forward_k_vs_all()` (*dicee.models.complex.ConEx method*), 67  
`forward_k_vs_all()` (*dicee.models.ConEx method*), 105  
`forward_k_vs_all()` (*dicee.models.ConvO method*), 118  
`forward_k_vs_all()` (*dicee.models.ConvQ method*), 112  
`forward_k_vs_all()` (*dicee.models.DeCaL method*), 124  
`forward_k_vs_all()` (*dicee.models.DistMult method*), 102  
`forward_k_vs_all()` (*dicee.models.DualE method*), 135  
`forward_k_vs_all()` (*dicee.models.dualE.DualE method*), 69  
`forward_k_vs_all()` (*dicee.models.Keci method*), 121  
`forward_k_vs_all()` (*dicee.models.octonion.AConvO method*), 76  
`forward_k_vs_all()` (*dicee.models.octonion.ConvO method*), 75  
`forward_k_vs_all()` (*dicee.models.octonion.OMult method*), 74  
`forward_k_vs_all()` (*dicee.models.OMult method*), 117  
`forward_k_vs_all()` (*dicee.models.pykeen\_models.PykeenKGE method*), 77  
`forward_k_vs_all()` (*dicee.models.PykeenKGE method*), 129  
`forward_k_vs_all()` (*dicee.models.QMult method*), 112  
`forward_k_vs_all()` (*dicee.models.quaternion.AConvQ method*), 80  
`forward_k_vs_all()` (*dicee.models.quaternion.ConvQ method*), 80  
`forward_k_vs_all()` (*dicee.models.quaternion.QMult method*), 79  
`forward_k_vs_all()` (*dicee.models.real.DistMult method*), 81  
`forward_k_vs_all()` (*dicee.models.real.Shallom method*), 82  
`forward_k_vs_all()` (*dicee.models.real.TransE method*), 81  
`forward_k_vs_all()` (*dicee.models.Shallom method*), 102  
`forward_k_vs_all()` (*dicee.models.TransE method*), 102  
`forward_k_vs_all()` (*dicee.OMult method*), 178  
`forward_k_vs_all()` (*dicee.PykeenKGE method*), 180  
`forward_k_vs_all()` (*dicee.QMult method*), 177  
`forward_k_vs_all()` (*dicee.Shallom method*), 178  
`forward_k_vs_all()` (*dicee.TransE method*), 166  
`forward_k_vs_sample()` (*dicee.AConEx method*), 172  
`forward_k_vs_sample()` (*dicee.BaseKGE method*), 184  
`forward_k_vs_sample()` (*dicee.ComplEx method*), 171  
`forward_k_vs_sample()` (*dicee.ConEx method*), 175  
`forward_k_vs_sample()` (*dicee.DistMult method*), 162  
`forward_k_vs_sample()` (*dicee.Keci method*), 165  
`forward_k_vs_sample()` (*dicee.models.AConEx method*), 106  
`forward_k_vs_sample()` (*dicee.models.base\_model.BaseKGE method*), 58

`forward_k_vs_sample()` (*dicee.models.BaseKGE method*), 98, 101, 105, 109, 115, 128, 131  
`forward_k_vs_sample()` (*dicee.models.clifford.Keci method*), 62  
`forward_k_vs_sample()` (*dicee.models.ComplEx method*), 107  
`forward_k_vs_sample()` (*dicee.models.complex.AConEx method*), 67  
`forward_k_vs_sample()` (*dicee.models.complex.ComplEx method*), 68  
`forward_k_vs_sample()` (*dicee.models.complex.ConEx method*), 67  
`forward_k_vs_sample()` (*dicee.models.ConEx method*), 105  
`forward_k_vs_sample()` (*dicee.models.DistMult method*), 102  
`forward_k_vs_sample()` (*dicee.models.Keci method*), 122  
`forward_k_vs_sample()` (*dicee.models.pykeen\_models.PykeenKGE method*), 77  
`forward_k_vs_sample()` (*dicee.models.PykeenKGE method*), 129  
`forward_k_vs_sample()` (*dicee.models.QMult method*), 112  
`forward_k_vs_sample()` (*dicee.models.quaternion.QMult method*), 79  
`forward_k_vs_sample()` (*dicee.models.real.DistMult method*), 81  
`forward_k_vs_sample()` (*dicee.PykeenKGE method*), 180  
`forward_k_vs_sample()` (*dicee.QMult method*), 177  
`forward_k_vs_with_explicit()` (*dicee.Keci method*), 165  
`forward_k_vs_with_explicit()` (*dicee.models.clifford.Keci method*), 61  
`forward_k_vs_with_explicit()` (*dicee.models.Keci method*), 121  
`forward_triples()` (*dicee.AConEx method*), 172  
`forward_triples()` (*dicee.AConvO method*), 172  
`forward_triples()` (*dicee.AConvQ method*), 173  
`forward_triples()` (*dicee.BaseKGE method*), 184  
`forward_triples()` (*dicee.ConEx method*), 175  
`forward_triples()` (*dicee.ConvO method*), 175  
`forward_triples()` (*dicee.ConvQ method*), 173  
`forward_triples()` (*dicee.DeCaL method*), 167  
`forward_triples()` (*dicee.DualE method*), 170  
`forward_triples()` (*dicee.Keci method*), 165  
`forward_triples()` (*dicee.LFMult method*), 178  
`forward_triples()` (*dicee.models.AConEx method*), 106  
`forward_triples()` (*dicee.models.AConvO method*), 119  
`forward_triples()` (*dicee.models.AConvQ method*), 113  
`forward_triples()` (*dicee.models.base\_model.BaseKGE method*), 58  
`forward_triples()` (*dicee.models.BaseKGE method*), 98, 101, 104, 109, 115, 128, 131  
`forward_triples()` (*dicee.models.clifford.DeCaL method*), 63  
`forward_triples()` (*dicee.models.clifford.Keci method*), 62  
`forward_triples()` (*dicee.models.complex.AConEx method*), 67  
`forward_triples()` (*dicee.models.complex.ConEx method*), 67  
`forward_triples()` (*dicee.models.ConEx method*), 105  
`forward_triples()` (*dicee.models.ConvO method*), 118  
`forward_triples()` (*dicee.models.ConvQ method*), 112  
`forward_triples()` (*dicee.models.DeCaL method*), 123  
`forward_triples()` (*dicee.models.DualE method*), 135  
`forward_triples()` (*dicee.models.dualE.DualE method*), 69  
`forward_triples()` (*dicee.models.FMult method*), 132  
`forward_triples()` (*dicee.models.FMult2 method*), 133  
`forward_triples()` (*dicee.models.function\_space.FMult method*), 70  
`forward_triples()` (*dicee.models.function\_space.FMult2 method*), 71  
`forward_triples()` (*dicee.models.function\_space.GFMult method*), 71  
`forward_triples()` (*dicee.models.function\_space.LFMult method*), 72  
`forward_triples()` (*dicee.models.function\_space.LFMult1 method*), 72  
`forward_triples()` (*dicee.models.GFMult method*), 132  
`forward_triples()` (*dicee.models.Keci method*), 122  
`forward_triples()` (*dicee.models.LFMult method*), 134  
`forward_triples()` (*dicee.models.LFMult1 method*), 133  
`forward_triples()` (*dicee.models.octonion.AConvO method*), 76  
`forward_triples()` (*dicee.models.octonion.ConvO method*), 75  
`forward_triples()` (*dicee.models.Pyke method*), 102  
`forward_triples()` (*dicee.models.pykeen\_models.PykeenKGE method*), 77  
`forward_triples()` (*dicee.models.PykeenKGE method*), 129  
`forward_triples()` (*dicee.models.quaternion.AConvQ method*), 80  
`forward_triples()` (*dicee.models.quaternion.ConvQ method*), 80  
`forward_triples()` (*dicee.models.real.Pyke method*), 82  
`forward_triples()` (*dicee.models.real.Shallom method*), 82  
`forward_triples()` (*dicee.models.Shallom method*), 102  
`forward_triples()` (*dicee.Pyke method*), 162  
`forward_triples()` (*dicee.PykeenKGE method*), 180

`forward_triples()` (*dicее.Shallom method*), 178  
`frequency` (*dicее.callbacks.Perturb attribute*), 25  
`from_pretrained()` (*dicее.models.transformers.GPT class method*), 89  
`full_storage_path` (*dicее.analyse\_experiments.Experiment attribute*), 18  
`func_triple_to_bpe_representation` (*dicее.evaluator.Evaluator attribute*), 41  
`func_triple_to_bpe_representation()` (*dicее.knowledge\_graph.KG method*), 46  
`function()` (*dicее.models.FMult2 method*), 133  
`function()` (*dicее.models.function\_space.FMult2 method*), 71

## G

`gamma` (*dicее.models.FMult attribute*), 132  
`gamma` (*dicее.models.function\_space.FMult attribute*), 70  
`gelu` (*dicее.models.transformers.MLP attribute*), 87  
`gen_test` (*dicее.query\_generator.QueryGenerator attribute*), 136  
`gen_test` (*dicее.QueryGenerator attribute*), 204  
`gen_valid` (*dicее.query\_generator.QueryGenerator attribute*), 136  
`gen_valid` (*dicее.QueryGenerator attribute*), 204  
`generate()` (*dicее.BytE method*), 181  
`generate()` (*dicее.KGE method*), 188  
`generate()` (*dicее.knowledge\_graph\_embeddings.KGE method*), 46  
`generate()` (*dicее.models.transformers.BytE method*), 84  
`generate_queries()` (*dicее.query\_generator.QueryGenerator method*), 137  
`generate_queries()` (*dicее.QueryGenerator method*), 205  
`get()` (*dicее.scripts.serve.NeuralSearcher method*), 146  
`get_aswa_state_dict()` (*dicее.callbacks.ASWA method*), 23  
`get_bpe_head_and_relation_representation()` (*dicее.BaseKGE method*), 185  
`get_bpe_head_and_relation_representation()` (*dicее.models.base\_model.BaseKGE method*), 58  
`get_bpe_head_and_relation_representation()` (*dicее.models.BaseKGE method*), 98, 101, 105, 109, 115, 128, 131  
`get_bpe_token_representation()` (*dicее.abstracts.BaseInteractiveKGE method*), 14  
`get_callbacks()` (*in module dicее.trainer.dice\_trainer*), 152  
`get_default_arguments()` (*in module dicее.analyse\_experiments*), 18  
`get_default_arguments()` (*in module dicее.scripts.index*), 145  
`get_default_arguments()` (*in module dicее.scripts.run*), 145  
`get_default_arguments()` (*in module dicее.scripts.serve*), 146  
`get_ee_vocab()` (*in module dicее*), 185  
`get_ee_vocab()` (*in module dicее.read\_preprocess\_save\_load\_kg.util*), 142  
`get_ee_vocab()` (*in module dicее.static\_funcs*), 148  
`get_ee_vocab()` (*in module dicее.static\_preprocess\_funcs*), 151  
`get_embeddings()` (*dicее.BaseKGE method*), 185  
`get_embeddings()` (*dicее.models.base\_model.BaseKGE method*), 58  
`get_embeddings()` (*dicее.models.BaseKGE method*), 98, 101, 105, 109, 115, 128, 131  
`get_embeddings()` (*dicее.models.real.Shallom method*), 81  
`get_embeddings()` (*dicее.models.Shallom method*), 102  
`get_embeddings()` (*dicее.Shallom method*), 178  
`get_ensemble()` (*dicее.trainer.model\_parallelism.MP method*), 154  
`get_entity_embeddings()` (*dicее.abstracts.BaseInteractiveKGE method*), 15  
`get_entity_index()` (*dicее.abstracts.BaseInteractiveKGE method*), 14  
`get_er_vocab()` (*in module dicее*), 185  
`get_er_vocab()` (*in module dicее.read\_preprocess\_save\_load\_kg.util*), 142  
`get_er_vocab()` (*in module dicее.static\_funcs*), 148  
`get_er_vocab()` (*in module dicее.static\_preprocess\_funcs*), 151  
`get_eval_report()` (*dicее.abstracts.BaseInteractiveKGE method*), 14  
`get_head_relation_representation()` (*dicее.BaseKGE method*), 184  
`get_head_relation_representation()` (*dicее.models.base\_model.BaseKGE method*), 58  
`get_head_relation_representation()` (*dicее.models.BaseKGE method*), 98, 101, 105, 109, 115, 128, 131  
`get_kronecker_triple_representation()` (*dicее.callbacks.KronE method*), 25  
`get_num_params()` (*dicее.models.transformers.GPT method*), 89  
`get_padded_bpe_triple_representation()` (*dicее.abstracts.BaseInteractiveKGE method*), 14  
`get_queries()` (*dicее.query\_generator.QueryGenerator method*), 137  
`get_queries()` (*dicее.QueryGenerator method*), 205  
`get_re_vocab()` (*in module dicее*), 185  
`get_re_vocab()` (*in module dicее.read\_preprocess\_save\_load\_kg.util*), 142  
`get_re_vocab()` (*in module dicее.static\_funcs*), 148  
`get_re_vocab()` (*in module dicее.static\_preprocess\_funcs*), 151  
`get_relation_embeddings()` (*dicее.abstracts.BaseInteractiveKGE method*), 15  
`get_relation_index()` (*dicее.abstracts.BaseInteractiveKGE method*), 14  
`get_sentence_representation()` (*dicее.BaseKGE method*), 184

`get_sentence_representation()` (*dicee.models.base\_model.BaseKGE method*), 58  
`get_sentence_representation()` (*dicee.models.BaseKGE method*), 98, 101, 105, 109, 115, 128, 131  
`get_transductive_entity_embeddings()` (*dicee.KGE method*), 188  
`get_transductive_entity_embeddings()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 46  
`get_triple_representation()` (*dicee.BaseKGE method*), 184  
`get_triple_representation()` (*dicee.models.base\_model.BaseKGE method*), 58  
`get_triple_representation()` (*dicee.models.BaseKGE method*), 98, 101, 105, 109, 115, 128, 131  
`GFMult` (*class in dicee.models*), 132  
`GFMult` (*class in dicee.models.function\_space*), 70  
`global_rank` (*dicee.abstracts.AbstractTrainer attribute*), 12  
`global_rank` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 157  
`GPT` (*class in dicee.models.transformers*), 88  
`GPTConfig` (*class in dicee.models.transformers*), 88  
`gpus` (*dicee.config.Namespace attribute*), 26  
`gradient_accumulation_steps` (*dicee.config.Namespace attribute*), 27  
`ground_queries()` (*dicee.query\_generator.QueryGenerator method*), 137  
`ground_queries()` (*dicee.QueryGenerator method*), 205

## H

`hidden_dropout` (*dicee.BaseKGE attribute*), 184  
`hidden_dropout` (*dicee.models.base\_model.BaseKGE attribute*), 57  
`hidden_dropout` (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130  
`hidden_dropout_rate` (*dicee.BaseKGE attribute*), 183  
`hidden_dropout_rate` (*dicee.config.Namespace attribute*), 28  
`hidden_dropout_rate` (*dicee.models.base\_model.BaseKGE attribute*), 57  
`hidden_dropout_rate` (*dicee.models.BaseKGE attribute*), 97, 100, 103, 108, 114, 127, 130  
`hidden_normalizer` (*dicee.BaseKGE attribute*), 184  
`hidden_normalizer` (*dicee.models.base\_model.BaseKGE attribute*), 57  
`hidden_normalizer` (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130

## I

`IdentityClass` (*class in dicee.models*), 98, 110, 115  
`IdentityClass` (*class in dicee.models.base\_model*), 58  
`idx_entity_to_bpe_shaped` (*dicee.knowledge\_graph.KG attribute*), 45  
`index_triple()` (*dicee.abstracts.BaseInteractiveKGE method*), 15  
`init_dataloader()` (*dicee.DICE\_Trainer method*), 187  
`init_dataloader()` (*dicee.trainer.DICE\_Trainer method*), 158  
`init_dataloader()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 153  
`init_dataset()` (*dicee.DICE\_Trainer method*), 187  
`init_dataset()` (*dicee.trainer.DICE\_Trainer method*), 159  
`init_dataset()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 153  
`init_param` (*dicee.config.Namespace attribute*), 27  
`init_params_with_sanity_checking()` (*dicee.BaseKGE method*), 184  
`init_params_with_sanity_checking()` (*dicee.models.base\_model.BaseKGE method*), 58  
`init_params_with_sanity_checking()` (*dicee.models.BaseKGE method*), 97, 101, 104, 109, 115, 127, 131  
`initial_eval_setting` (*dicee.callbacks.ASWA attribute*), 22  
`initialize_or_load_model()` (*dicee.DICE\_Trainer method*), 187  
`initialize_or_load_model()` (*dicee.trainer.DICE\_Trainer method*), 158  
`initialize_or_load_model()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 153  
`initialize_trainer()` (*dicee.DICE\_Trainer method*), 187  
`initialize_trainer()` (*dicee.trainer.DICE\_Trainer method*), 158  
`initialize_trainer()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 153  
`initialize_trainer()` (*in module dicee.trainer.dice\_trainer*), 152  
`input_dp_ent_real` (*dicee.BaseKGE attribute*), 184  
`input_dp_ent_real` (*dicee.models.base\_model.BaseKGE attribute*), 57  
`input_dp_ent_real` (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130  
`input_dp_rel_real` (*dicee.BaseKGE attribute*), 184  
`input_dp_rel_real` (*dicee.models.base\_model.BaseKGE attribute*), 57  
`input_dp_rel_real` (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130  
`input_dropout_rate` (*dicee.BaseKGE attribute*), 183  
`input_dropout_rate` (*dicee.config.Namespace attribute*), 28  
`input_dropout_rate` (*dicee.models.base\_model.BaseKGE attribute*), 57  
`input_dropout_rate` (*dicee.models.BaseKGE attribute*), 97, 100, 103, 108, 114, 127, 130  
`intialize_model()` (*in module dicee*), 186  
`intialize_model()` (*in module dicee.static\_funcs*), 149  
`is_continual_training` (*dicee.DICE\_Trainer attribute*), 187  
`is_continual_training` (*dicee.evaluator.Evaluator attribute*), 41

- `is_continual_training` (*dicee.Execute attribute*), 192
- `is_continual_training` (*dicee.executer.Execute attribute*), 43
- `is_continual_training` (*dicee.trainer.DICE\_Trainer attribute*), 158
- `is_continual_training` (*dicee.trainer.dice\_trainer.DICE\_Trainer attribute*), 152
- `is_global_zero` (*dicee.abstracts.AbstractTrainer attribute*), 12
- `is_seen` () (*dicee.abstracts.BaseInteractiveKGE method*), 14
- `is_sparql_endpoint_alive` () (*in module dicee.sanity\_checkers*), 144

## K

- `k` (*dicee.models.FMult attribute*), 132
- `k` (*dicee.models.FMult2 attribute*), 133
- `k` (*dicee.models.function\_space.FMult attribute*), 70
- `k` (*dicee.models.function\_space.FMult2 attribute*), 71
- `k` (*dicee.models.function\_space.GFMult attribute*), 71
- `k` (*dicee.models.GFMult attribute*), 132
- `k_fold_cross_validation` () (*dicee.DICE\_Trainer method*), 187
- `k_fold_cross_validation` () (*dicee.trainer.DICE\_Trainer method*), 159
- `k_fold_cross_validation` () (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 153
- `k_vs_all_score` () (*dicee.ComplEx static method*), 171
- `k_vs_all_score` () (*dicee.DistMult method*), 162
- `k_vs_all_score` () (*dicee.Keci method*), 165
- `k_vs_all_score` () (*dicee.models.clifford.Keci method*), 62
- `k_vs_all_score` () (*dicee.models.ComplEx static method*), 107
- `k_vs_all_score` () (*dicee.models.complex.ComplEx static method*), 68
- `k_vs_all_score` () (*dicee.models.DistMult method*), 101
- `k_vs_all_score` () (*dicee.models.Keci method*), 121
- `k_vs_all_score` () (*dicee.models.octonion.OMult method*), 74
- `k_vs_all_score` () (*dicee.models.OMult method*), 117
- `k_vs_all_score` () (*dicee.models.QMult method*), 112
- `k_vs_all_score` () (*dicee.models.quaternion.QMult method*), 79
- `k_vs_all_score` () (*dicee.models.real.DistMult method*), 81
- `k_vs_all_score` () (*dicee.OMult method*), 178
- `k_vs_all_score` () (*dicee.QMult method*), 177
- `Keci` (*class in dicee*), 163
- `Keci` (*class in dicee.models*), 119
- `Keci` (*class in dicee.models.clifford*), 59
- `KeciBase` (*class in dicee*), 162
- `KeciBase` (*class in dicee.models*), 122
- `KeciBase` (*class in dicee.models.clifford*), 62
- `kernel_size` (*dicee.BaseKGE attribute*), 183
- `kernel_size` (*dicee.config.Namespace attribute*), 27
- `kernel_size` (*dicee.models.base\_model.BaseKGE attribute*), 57
- `kernel_size` (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130
- `KG` (*class in dicee.knowledge\_graph*), 45
- `kg` (*dicee.callbacks.PseudoLabellingCallback attribute*), 22
- `kg` (*dicee.read\_preprocess\_save\_load\_kg.LoadSaveToDisk attribute*), 144
- `kg` (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG attribute*), 143
- `kg` (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG attribute*), 137
- `kg` (*dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk.ReadFromDisk attribute*), 138
- `kg` (*dicee.read\_preprocess\_save\_load\_kg.ReadFromDisk attribute*), 144
- `kg` (*dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk.LoadSaveToDisk attribute*), 139
- `KGE` (*class in dicee*), 188
- `KGE` (*class in dicee.knowledge\_graph\_embeddings*), 46
- `KGESaveCallback` (*class in dicee.callbacks*), 21
- `knowledge_graph` (*dicee.Execute attribute*), 192
- `knowledge_graph` (*dicee.executer.Execute attribute*), 43
- `KronE` (*class in dicee.callbacks*), 24
- `KvsAll` (*class in dicee*), 195
- `KvsAll` (*class in dicee.dataset\_classes*), 31
- `kvsall_score` () (*dicee.DualE method*), 170
- `kvsall_score` () (*dicee.models.DualE method*), 135
- `kvsall_score` () (*dicee.models.dualE.DualE method*), 69
- `KvsSampleDataset` (*class in dicee*), 199
- `KvsSampleDataset` (*class in dicee.dataset\_classes*), 34

## L

- `label_smoothing_rate` (*dicee.AllvsAll attribute*), 197



label\_smoothing\_rate (*dicee.config.Namespace attribute*), 27  
 label\_smoothing\_rate (*dicee.dataset\_classes.AllvsAll attribute*), 32  
 label\_smoothing\_rate (*dicee.dataset\_classes.KvsAll attribute*), 32  
 label\_smoothing\_rate (*dicee.dataset\_classes.KvsSampleDataset attribute*), 35  
 label\_smoothing\_rate (*dicee.dataset\_classes.OnevsSample attribute*), 33, 34  
 label\_smoothing\_rate (*dicee.dataset\_classes.TriplePredictionDataset attribute*), 36  
 label\_smoothing\_rate (*dicee.KvsAll attribute*), 196  
 label\_smoothing\_rate (*dicee.KvsSampleDataset attribute*), 199  
 label\_smoothing\_rate (*dicee.OnevsSample attribute*), 198  
 label\_smoothing\_rate (*dicee.TriplePredictionDataset attribute*), 200  
 LayerNorm (*class in dicee.models.transformers*), 85  
 learning\_rate (*dicee.BaseKGE attribute*), 183  
 learning\_rate (*dicee.models.base\_model.BaseKGE attribute*), 57  
 learning\_rate (*dicee.models.BaseKGE attribute*), 97, 100, 103, 108, 114, 127, 130  
 length (*dicee.dataset\_classes.NegSampleDataset attribute*), 35  
 length (*dicee.dataset\_classes.TriplePredictionDataset attribute*), 36  
 length (*dicee.NegSampleDataset attribute*), 200  
 length (*dicee.TriplePredictionDataset attribute*), 201  
 level (*dicee.callbacks.Perturb attribute*), 25  
 LFMult (*class in dicee*), 178  
 LFMult (*class in dicee.models*), 133  
 LFMult (*class in dicee.models.function\_space*), 72  
 LFMult1 (*class in dicee.models*), 133  
 LFMult1 (*class in dicee.models.function\_space*), 71  
 linear () (*dicee.LFMult method*), 179  
 linear () (*dicee.models.function\_space.LFMult method*), 72  
 linear () (*dicee.models.LFMult method*), 134  
 list2tuple () (*dicee.query\_generator.QueryGenerator method*), 136  
 list2tuple () (*dicee.QueryGenerator method*), 205  
 lm\_head (*dicee.BytE attribute*), 181  
 lm\_head (*dicee.models.transformers.BytE attribute*), 84  
 lm\_head (*dicee.models.transformers.GPT attribute*), 89  
 ln\_1 (*dicee.models.transformers.Block attribute*), 87  
 ln\_2 (*dicee.models.transformers.Block attribute*), 88  
 load () (*dicee.read\_preprocess\_save\_load\_kg.LoadSaveToDisk method*), 144  
 load () (*dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk.LoadSaveToDisk method*), 139  
 load\_json () (*in module dicee*), 186  
 load\_json () (*in module dicee.static\_funcs*), 149  
 load\_model () (*in module dicee*), 185  
 load\_model () (*in module dicee.static\_funcs*), 148  
 load\_model\_ensemble () (*in module dicee*), 185  
 load\_model\_ensemble () (*in module dicee.static\_funcs*), 148  
 load\_numpy () (*in module dicee*), 186  
 load\_numpy () (*in module dicee.static\_funcs*), 149  
 load\_numpy\_ndarray () (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 142  
 load\_pickle () (*in module dicee*), 185  
 load\_pickle () (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 143  
 load\_pickle () (*in module dicee.static\_funcs*), 148  
 load\_queries () (*dicee.query\_generator.QueryGenerator method*), 137  
 load\_queries () (*dicee.QueryGenerator method*), 205  
 load\_queries\_and\_answers () (*dicee.query\_generator.QueryGenerator static method*), 137  
 load\_queries\_and\_answers () (*dicee.QueryGenerator static method*), 205  
 load\_term\_mapping () (*in module dicee*), 185, 193  
 load\_term\_mapping () (*in module dicee.static\_funcs*), 148  
 load\_term\_mapping () (*in module dicee.trainer.dice\_trainer*), 152  
 load\_with\_pandas () (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 142  
 LoadSaveToDisk (*class in dicee.read\_preprocess\_save\_load\_kg*), 144  
 LoadSaveToDisk (*class in dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk*), 139  
 local\_rank (*dicee.abstracts.AbstractTrainer attribute*), 12  
 local\_rank (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 157  
 loss (*dicee.BaseKGE attribute*), 183  
 loss (*dicee.models.base\_model.BaseKGE attribute*), 57  
 loss (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130  
 loss\_func (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 157  
 loss\_function (*dicee.trainer.torch\_trainer.TorchTrainer attribute*), 155  
 loss\_function (*dicee.trainer.torch\_trainer.xMP attribute*), 154  
 loss\_function () (*dicee.BytE method*), 181  
 loss\_function () (*dicee.models.base\_model.BaseKGELightning method*), 52

- `loss_function()` (*dicее.models.BaseKGELightning method*), 92
- `loss_function()` (*dicее.models.transformers.BytE method*), 84
- `loss_history` (*dicее.BaseKGE attribute*), 184
- `loss_history` (*dicее.models.base\_model.BaseKGE attribute*), 57
- `loss_history` (*dicее.models.BaseKGE attribute*), 97, 100, 104, 109, 114, 127, 130
- `loss_history` (*dicее.models.pykeen\_models.PykeenKGE attribute*), 77
- `loss_history` (*dicее.models.PykeenKGE attribute*), 128
- `loss_history` (*dicее.PykeenKGE attribute*), 180
- `loss_history` (*dicее.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 157
- `lr` (*dicее.analyse\_experiments.Experiment attribute*), 18
- `lr` (*dicее.config.Namespace attribute*), 26

## M

- `m` (*dicее.LFMult attribute*), 178
- `m` (*dicее.models.function\_space.LFMult attribute*), 72
- `m` (*dicее.models.LFMult attribute*), 134
- `main()` (*in module dicее.scripts.index*), 145
- `main()` (*in module dicее.scripts.run*), 145
- `main()` (*in module dicее.scripts.serve*), 146
- `make_iterable_verbose()` (*in module dicее.static\_funcs\_training*), 150
- `make_iterable_verbose()` (*in module dicее.trainer.torch\_trainer\_ddp*), 156
- `mapping_from_first_two_cols_to_third()` (*in module dicее*), 193
- `mapping_from_first_two_cols_to_third()` (*in module dicее.static\_preprocess\_funcs*), 151
- `margin` (*dicее.models.Pyke attribute*), 102
- `margin` (*dicее.models.real.Pyke attribute*), 82
- `margin` (*dicее.models.real.TransE attribute*), 81
- `margin` (*dicее.models.TransE attribute*), 102
- `margin` (*dicее.Pyke attribute*), 162
- `margin` (*dicее.TransE attribute*), 166
- `max_ans_num` (*dicее.query\_generator.QueryGenerator attribute*), 136
- `max_ans_num` (*dicее.QueryGenerator attribute*), 204
- `max_epochs` (*dicее.callbacks.KGESaveCallback attribute*), 21
- `max_length_subword_tokens` (*dicее.BaseKGE attribute*), 184
- `max_length_subword_tokens` (*dicее.knowledge\_graph.KG attribute*), 46
- `max_length_subword_tokens` (*dicее.models.base\_model.BaseKGE attribute*), 57
- `max_length_subword_tokens` (*dicее.models.BaseKGE attribute*), 97, 100, 104, 109, 115, 127, 131
- `max_num_of_classes` (*dicее.dataset\_classes.KvsSampleDataset attribute*), 35
- `max_num_of_classes` (*dicее.KvsSampleDataset attribute*), 199
- `mem_of_model()` (*dicее.models.base\_model.BaseKGELightning method*), 51
- `mem_of_model()` (*dicее.models.BaseKGELightning method*), 91
- `method` (*dicее.callbacks.Perturb attribute*), 25
- `MLP` (*class in dicее.models.transformers*), 86
- `mlp` (*dicее.models.transformers.Block attribute*), 88
- `mode` (*dicее.query\_generator.QueryGenerator attribute*), 136
- `mode` (*dicее.QueryGenerator attribute*), 205
- `model` (*dicее.config.Namespace attribute*), 26
- `model` (*dicее.models.pykeen\_models.PykeenKGE attribute*), 77
- `model` (*dicее.models.PykeenKGE attribute*), 128
- `model` (*dicее.PykeenKGE attribute*), 179
- `model` (*dicее.scripts.serve.NeuralSearcher attribute*), 146
- `model` (*dicее.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 157
- `model` (*dicее.trainer.torch\_trainer.TorchTrainer attribute*), 155
- `model` (*dicее.trainer.torch\_trainer.xMP attribute*), 155
- `model_kwargs` (*dicее.models.pykeen\_models.PykeenKGE attribute*), 76
- `model_kwargs` (*dicее.models.PykeenKGE attribute*), 128
- `model_kwargs` (*dicее.PykeenKGE attribute*), 179
- `model_name` (*dicее.analyse\_experiments.Experiment attribute*), 18
- `models` (*dicее.trainer.dice\_trainer.EnsembleKGE attribute*), 152
- `module`
  - `dicее`, 12
  - `dicее.__main__`, 12
  - `dicее.abstracts`, 12
  - `dicее.analyse_experiments`, 17
  - `dicее.callbacks`, 19
  - `dicее.config`, 25
  - `dicее.dataset_classes`, 28
  - `dicее.eval_static_funcs`, 40

- dicee.evaluator, 41
- dicee.executer, 42
- dicee.knowledge\_graph, 45
- dicee.knowledge\_graph\_embeddings, 46
- dicee.models, 50
- dicee.models.base\_model, 50
- dicee.models.clifford, 59
- dicee.models.complex, 66
- dicee.models.dualE, 69
- dicee.models.function\_space, 70
- dicee.models.octonion, 73
- dicee.models.pykeen\_models, 76
- dicee.models.quaternion, 77
- dicee.models.real, 81
- dicee.models.static\_funcs, 82
- dicee.models.transformers, 82
- dicee.query\_generator, 136
- dicee.read\_preprocess\_save\_load\_kg, 137
- dicee.read\_preprocess\_save\_load\_kg.preprocess, 137
- dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk, 138
- dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk, 139
- dicee.read\_preprocess\_save\_load\_kg.util, 139
- dicee.sanity\_checkers, 144
- dicee.scripts, 145
- dicee.scripts.index, 145
- dicee.scripts.run, 145
- dicee.scripts.serve, 145
- dicee.static\_funcs, 146
- dicee.static\_funcs\_training, 150
- dicee.static\_preprocess\_funcs, 150
- dicee.trainer, 151
- dicee.trainer.dice\_trainer, 151
- dicee.trainer.model\_parallelism, 154
- dicee.trainer.torch\_trainer, 154
- dicee.trainer.torch\_trainer\_ddp, 156
- MP (*class in dicee.trainer.model\_parallelism*), 154
- MultiClassClassificationDataset (*class in dicee*), 194
- MultiClassClassificationDataset (*class in dicee.dataset\_classes*), 30
- MultiLabelDataset (*class in dicee*), 194
- MultiLabelDataset (*class in dicee.dataset\_classes*), 29

## N

- n (*dicee.models.FMult2 attribute*), 133
- n (*dicee.models.function\_space.FMult2 attribute*), 71
- n\_embd (*dicee.models.transformers.CausalSelfAttention attribute*), 86
- n\_embd (*dicee.models.transformers.GPTConfig attribute*), 88
- n\_head (*dicee.models.transformers.CausalSelfAttention attribute*), 86
- n\_head (*dicee.models.transformers.GPTConfig attribute*), 88
- n\_layer (*dicee.models.transformers.GPTConfig attribute*), 88
- n\_layers (*dicee.models.FMult2 attribute*), 133
- n\_layers (*dicee.models.function\_space.FMult2 attribute*), 71
- name (*dicee.abstracts.BaseInteractiveKGE property*), 14
- name (*dicee.AConEx attribute*), 171
- name (*dicee.AConvO attribute*), 172
- name (*dicee.AConvQ attribute*), 172
- name (*dicee.BytE attribute*), 181
- name (*dicee.ComplEx attribute*), 171
- name (*dicee.ConEx attribute*), 175
- name (*dicee.ConvO attribute*), 174
- name (*dicee.ConvQ attribute*), 173
- name (*dicee.DeCaL attribute*), 166
- name (*dicee.DistMult attribute*), 162
- name (*dicee.DualE attribute*), 170
- name (*dicee.Keci attribute*), 163
- name (*dicee.KeciBase attribute*), 162
- name (*dicee.LFMult attribute*), 178
- name (*dicee.models.AConEx attribute*), 106



name (*dicee.models.AConvO* attribute), 118  
 name (*dicee.models.AConvQ* attribute), 112  
 name (*dicee.models.clifford.DeCaL* attribute), 63  
 name (*dicee.models.clifford.Keci* attribute), 60  
 name (*dicee.models.clifford.KeciBase* attribute), 62  
 name (*dicee.models.ComplEx* attribute), 107  
 name (*dicee.models.complex.AConEx* attribute), 67  
 name (*dicee.models.complex.ComplEx* attribute), 68  
 name (*dicee.models.complex.ConEx* attribute), 66  
 name (*dicee.models.ConEx* attribute), 105  
 name (*dicee.models.ConvO* attribute), 118  
 name (*dicee.models.ConvQ* attribute), 112  
 name (*dicee.models.DeCaL* attribute), 123  
 name (*dicee.models.DistMult* attribute), 101  
 name (*dicee.models.DualE* attribute), 135  
 name (*dicee.models.dualE.DualE* attribute), 69  
 name (*dicee.models.FMult* attribute), 132  
 name (*dicee.models.FMult2* attribute), 132  
 name (*dicee.models.function\_space.FMult* attribute), 70  
 name (*dicee.models.function\_space.FMult2* attribute), 71  
 name (*dicee.models.function\_space.GFMult* attribute), 70  
 name (*dicee.models.function\_space.LFMult* attribute), 72  
 name (*dicee.models.function\_space.LFMult1* attribute), 72  
 name (*dicee.models.GFMult* attribute), 132  
 name (*dicee.models.Keci* attribute), 119  
 name (*dicee.models.KeciBase* attribute), 122  
 name (*dicee.models.LFMult* attribute), 133  
 name (*dicee.models.LFMult1* attribute), 133  
 name (*dicee.models.octonion.AConvO* attribute), 76  
 name (*dicee.models.octonion.ConvO* attribute), 75  
 name (*dicee.models.octonion.OMult* attribute), 74  
 name (*dicee.models.OMult* attribute), 117  
 name (*dicee.models.Pyke* attribute), 102  
 name (*dicee.models.pykeen\_models.PykeenKGE* attribute), 76  
 name (*dicee.models.PykeenKGE* attribute), 128  
 name (*dicee.models.QMult* attribute), 111  
 name (*dicee.models.quaternion.AConvQ* attribute), 80  
 name (*dicee.models.quaternion.ConvQ* attribute), 79  
 name (*dicee.models.quaternion.QMult* attribute), 78  
 name (*dicee.models.real.DistMult* attribute), 81  
 name (*dicee.models.real.Pyke* attribute), 82  
 name (*dicee.models.real.Shallom* attribute), 81  
 name (*dicee.models.real.TransE* attribute), 81  
 name (*dicee.models.Shallom* attribute), 102  
 name (*dicee.models.TransE* attribute), 102  
 name (*dicee.models.transformers.BytE* attribute), 83  
 name (*dicee.OMult* attribute), 177  
 name (*dicee.Pyke* attribute), 162  
 name (*dicee.PykeenKGE* attribute), 179  
 name (*dicee.QMult* attribute), 176  
 name (*dicee.Shallom* attribute), 178  
 name (*dicee.TransE* attribute), 166  
 Namespace (*class in dicee.config*), 26  
 neg\_ratio (*dicee.BPE\_NegativeSamplingDataset* attribute), 194  
 neg\_ratio (*dicee.config.Namespace* attribute), 27  
 neg\_ratio (*dicee.dataset\_classes.BPE\_NegativeSamplingDataset* attribute), 29  
 neg\_ratio (*dicee.dataset\_classes.KvsSampleDataset* attribute), 35  
 neg\_ratio (*dicee.KvsSampleDataset* attribute), 199  
 neg\_sample\_ratio (*dicee.CVDDataModule* attribute), 201  
 neg\_sample\_ratio (*dicee.dataset\_classes.CVDDataModule* attribute), 37  
 neg\_sample\_ratio (*dicee.dataset\_classes.NegSampleDataset* attribute), 35  
 neg\_sample\_ratio (*dicee.dataset\_classes.OnevsSample* attribute), 33, 34  
 neg\_sample\_ratio (*dicee.dataset\_classes.TriplePredictionDataset* attribute), 36  
 neg\_sample\_ratio (*dicee.NegSampleDataset* attribute), 200  
 neg\_sample\_ratio (*dicee.OnevsSample* attribute), 198  
 neg\_sample\_ratio (*dicee.TriplePredictionDataset* attribute), 201  
 negnorm () (*dicee.KGE* method), 190  
 negnorm () (*dicee.knowledge\_graph\_embeddings.KGE* method), 49

NegSampleDataset (class in *dicce*), 199  
 NegSampleDataset (class in *dicce.dataset\_classes*), 35  
 neural\_searcher (in module *dicce.scripts.serve*), 146  
 NeuralSearcher (class in *dicce.scripts.serve*), 146  
 NodeTrainer (class in *dicce.trainer.torch\_trainer\_ddp*), 157  
 norm\_fc1 (*dicce.AConEx* attribute), 172  
 norm\_fc1 (*dicce.AConvO* attribute), 172  
 norm\_fc1 (*dicce.ConEx* attribute), 175  
 norm\_fc1 (*dicce.ConvO* attribute), 174  
 norm\_fc1 (*dicce.models.AConEx* attribute), 106  
 norm\_fc1 (*dicce.models.AConvO* attribute), 119  
 norm\_fc1 (*dicce.models.complex.AConEx* attribute), 67  
 norm\_fc1 (*dicce.models.complex.ConEx* attribute), 67  
 norm\_fc1 (*dicce.models.ConEx* attribute), 105  
 norm\_fc1 (*dicce.models.ConvO* attribute), 118  
 norm\_fc1 (*dicce.models.octonion.AConvO* attribute), 76  
 norm\_fc1 (*dicce.models.octonion.ConvO* attribute), 75  
 normalization (*dicce.analyse\_experiments.Experiment* attribute), 19  
 normalization (*dicce.config.Namespace* attribute), 27  
 normalize\_head\_entity\_embeddings (*dicce.BaseKGE* attribute), 184  
 normalize\_head\_entity\_embeddings (*dicce.models.base\_model.BaseKGE* attribute), 57  
 normalize\_head\_entity\_embeddings (*dicce.models.BaseKGE* attribute), 97, 100, 104, 108, 114, 127, 130  
 normalize\_relation\_embeddings (*dicce.BaseKGE* attribute), 184  
 normalize\_relation\_embeddings (*dicce.models.base\_model.BaseKGE* attribute), 57  
 normalize\_relation\_embeddings (*dicce.models.BaseKGE* attribute), 97, 100, 104, 108, 114, 127, 130  
 normalize\_tail\_entity\_embeddings (*dicce.BaseKGE* attribute), 184  
 normalize\_tail\_entity\_embeddings (*dicce.models.base\_model.BaseKGE* attribute), 57  
 normalize\_tail\_entity\_embeddings (*dicce.models.BaseKGE* attribute), 97, 100, 104, 108, 114, 127, 130  
 normalizer\_class (*dicce.BaseKGE* attribute), 183  
 normalizer\_class (*dicce.models.base\_model.BaseKGE* attribute), 57  
 normalizer\_class (*dicce.models.BaseKGE* attribute), 97, 100, 104, 108, 114, 127, 130  
 num\_bpe\_entities (*dicce.BPE\_NegativeSamplingDataset* attribute), 194  
 num\_bpe\_entities (*dicce.dataset\_classes.BPE\_NegativeSamplingDataset* attribute), 29  
 num\_bpe\_entities (*dicce.knowledge\_graph.KG* attribute), 45  
 num\_core (*dicce.config.Namespace* attribute), 27  
 num\_datapoints (*dicce.BPE\_NegativeSamplingDataset* attribute), 194  
 num\_datapoints (*dicce.dataset\_classes.BPE\_NegativeSamplingDataset* attribute), 29  
 num\_datapoints (*dicce.dataset\_classes.MultiLabelDataset* attribute), 30  
 num\_datapoints (*dicce.MultiLabelDataset* attribute), 194  
 num\_ent (*dicce.DualE* attribute), 170  
 num\_ent (*dicce.models.DualE* attribute), 135  
 num\_ent (*dicce.models.dualE.DualE* attribute), 69  
 num\_entities (*dicce.BaseKGE* attribute), 183  
 num\_entities (*dicce.CVDataModule* attribute), 201  
 num\_entities (*dicce.dataset\_classes.CVDataModule* attribute), 37  
 num\_entities (*dicce.dataset\_classes.KvsSampleDataset* attribute), 35  
 num\_entities (*dicce.dataset\_classes.NegSampleDataset* attribute), 35  
 num\_entities (*dicce.dataset\_classes.OnevsSample* attribute), 33, 34  
 num\_entities (*dicce.dataset\_classes.TriplePredictionDataset* attribute), 36  
 num\_entities (*dicce.evaluator.Evaluator* attribute), 41  
 num\_entities (*dicce.knowledge\_graph.KG* attribute), 45  
 num\_entities (*dicce.KvsSampleDataset* attribute), 199  
 num\_entities (*dicce.models.base\_model.BaseKGE* attribute), 57  
 num\_entities (*dicce.models.BaseKGE* attribute), 96, 100, 103, 108, 114, 126, 130  
 num\_entities (*dicce.NegSampleDataset* attribute), 200  
 num\_entities (*dicce.OnevsSample* attribute), 197, 198  
 num\_entities (*dicce.TriplePredictionDataset* attribute), 201  
 num\_epochs (*dicce.abstracts.AbstractPPECallback* attribute), 17  
 num\_epochs (*dicce.analyse\_experiments.Experiment* attribute), 18  
 num\_epochs (*dicce.callbacks.ASWA* attribute), 22  
 num\_epochs (*dicce.config.Namespace* attribute), 26  
 num\_epochs (*dicce.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 157  
 num\_folds\_for\_cv (*dicce.config.Namespace* attribute), 27  
 num\_of\_data\_points (*dicce.dataset\_classes.MultiClassClassificationDataset* attribute), 30  
 num\_of\_data\_points (*dicce.MultiClassClassificationDataset* attribute), 195  
 num\_of\_epochs (*dicce.callbacks.PseudoLabellingCallback* attribute), 22  
 num\_of\_output\_channels (*dicce.BaseKGE* attribute), 183  
 num\_of\_output\_channels (*dicce.config.Namespace* attribute), 27

- `num_of_output_channels` (*dicee.models.base\_model.BaseKGE attribute*), 57
- `num_of_output_channels` (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130
- `num_params` (*dicee.analyse\_experiments.Experiment attribute*), 18
- `num_relations` (*dicee.BaseKGE attribute*), 183
- `num_relations` (*dicee.CVDataModule attribute*), 201
- `num_relations` (*dicee.dataset\_classes.CVDataModule attribute*), 37
- `num_relations` (*dicee.dataset\_classes.NegSampleDataset attribute*), 35
- `num_relations` (*dicee.dataset\_classes.OnevsSample attribute*), 33, 34
- `num_relations` (*dicee.dataset\_classes.TriplePredictionDataset attribute*), 36
- `num_relations` (*dicee.evaluator.Evaluator attribute*), 41
- `num_relations` (*dicee.knowledge\_graph.KG attribute*), 45
- `num_relations` (*dicee.models.base\_model.BaseKGE attribute*), 57
- `num_relations` (*dicee.models.BaseKGE attribute*), 96, 100, 103, 108, 114, 126, 130
- `num_relations` (*dicee.NegSampleDataset attribute*), 200
- `num_relations` (*dicee.OnevsSample attribute*), 198
- `num_relations` (*dicee.TriplePredictionDataset attribute*), 201
- `num_sample` (*dicee.models.FMult attribute*), 132
- `num_sample` (*dicee.models.function\_space.FMult attribute*), 70
- `num_sample` (*dicee.models.function\_space.GFMult attribute*), 71
- `num_sample` (*dicee.models.GFMult attribute*), 132
- `num_tokens` (*dicee.BaseKGE attribute*), 183
- `num_tokens` (*dicee.knowledge\_graph.KG attribute*), 45
- `num_tokens` (*dicee.models.base\_model.BaseKGE attribute*), 57
- `num_tokens` (*dicee.models.BaseKGE attribute*), 96, 100, 103, 108, 114, 127, 130
- `num_workers` (*dicee.CVDataModule attribute*), 201
- `num_workers` (*dicee.dataset\_classes.CVDataModule attribute*), 37
- `numpy_data_type_changer` () (*in module dicee*), 185
- `numpy_data_type_changer` () (*in module dicee.static\_funcs*), 148

## O

- `octonion_mul` () (*in module dicee.models*), 116
- `octonion_mul` () (*in module dicee.models.octonion*), 74
- `octonion_mul_norm` () (*in module dicee.models*), 116
- `octonion_mul_norm` () (*in module dicee.models.octonion*), 74
- `octonion_normalizer` () (*dicee.AConvO static method*), 172
- `octonion_normalizer` () (*dicee.ConvO static method*), 174
- `octonion_normalizer` () (*dicee.models.AConvO static method*), 119
- `octonion_normalizer` () (*dicee.models.ConvO static method*), 118
- `octonion_normalizer` () (*dicee.models.octonion.AConvO static method*), 76
- `octonion_normalizer` () (*dicee.models.octonion.ConvO static method*), 75
- `octonion_normalizer` () (*dicee.models.octonion.OMult static method*), 74
- `octonion_normalizer` () (*dicee.models.OMult static method*), 117
- `octonion_normalizer` () (*dicee.OMult static method*), 178
- `OMult` (*class in dicee*), 177
- `OMult` (*class in dicee.models*), 116
- `OMult` (*class in dicee.models.octonion*), 74
- `on_epoch_end` () (*dicee.callbacks.KGESaveCallback method*), 22
- `on_epoch_end` () (*dicee.callbacks.PseudoLabellingCallback method*), 22
- `on_fit_end` () (*dicee.abstracts.AbstractCallback method*), 16
- `on_fit_end` () (*dicee.abstracts.AbstractPPECallback method*), 17
- `on_fit_end` () (*dicee.abstracts.AbstractTrainer method*), 13
- `on_fit_end` () (*dicee.callbacks.AccumulateEpochLossCallback method*), 20
- `on_fit_end` () (*dicee.callbacks.ASWA method*), 23
- `on_fit_end` () (*dicee.callbacks.Eval method*), 24
- `on_fit_end` () (*dicee.callbacks.KGESaveCallback method*), 22
- `on_fit_end` () (*dicee.callbacks.PrintCallback method*), 20
- `on_fit_start` () (*dicee.abstracts.AbstractCallback method*), 16
- `on_fit_start` () (*dicee.abstracts.AbstractPPECallback method*), 17
- `on_fit_start` () (*dicee.abstracts.AbstractTrainer method*), 12
- `on_fit_start` () (*dicee.callbacks.Eval method*), 24
- `on_fit_start` () (*dicee.callbacks.KGESaveCallback method*), 21
- `on_fit_start` () (*dicee.callbacks.KronE method*), 25
- `on_fit_start` () (*dicee.callbacks.PrintCallback method*), 20
- `on_init_end` () (*dicee.abstracts.AbstractCallback method*), 16
- `on_init_start` () (*dicee.abstracts.AbstractCallback method*), 15
- `on_train_batch_end` () (*dicee.abstracts.AbstractCallback method*), 16
- `on_train_batch_end` () (*dicee.abstracts.AbstractTrainer method*), 13

- `on_train_batch_end()` (*dicee.callbacks.Eval method*), 24
- `on_train_batch_end()` (*dicee.callbacks.KGESaveCallback method*), 21
- `on_train_batch_end()` (*dicee.callbacks.PrintCallback method*), 20
- `on_train_batch_start()` (*dicee.callbacks.Perturb method*), 25
- `on_train_epoch_end()` (*dicee.abstracts.AbstractCallback method*), 16
- `on_train_epoch_end()` (*dicee.abstracts.AbstractTrainer method*), 13
- `on_train_epoch_end()` (*dicee.callbacks.ASWA method*), 23
- `on_train_epoch_end()` (*dicee.callbacks.Eval method*), 24
- `on_train_epoch_end()` (*dicee.callbacks.KGESaveCallback method*), 21
- `on_train_epoch_end()` (*dicee.callbacks.PrintCallback method*), 21
- `on_train_epoch_end()` (*dicee.models.base\_model.BaseKGELightning method*), 52
- `on_train_epoch_end()` (*dicee.models.BaseKGELightning method*), 92
- `OnevsAllDataset` (*class in dicee*), 195
- `OnevsAllDataset` (*class in dicee.dataset\_classes*), 31
- `OnevsSample` (*class in dicee*), 197
- `OnevsSample` (*class in dicee.dataset\_classes*), 33
- `optim` (*dicee.config.Namespace attribute*), 26
- `optimizer` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 157
- `optimizer` (*dicee.trainer.torch\_trainer.TorchTrainer attribute*), 155
- `optimizer` (*dicee.trainer.torch\_trainer.xMP attribute*), 155
- `optimizer_name` (*dicee.BaseKGE attribute*), 183
- `optimizer_name` (*dicee.models.base\_model.BaseKGE attribute*), 57
- `optimizer_name` (*dicee.models.BaseKGE attribute*), 97, 100, 103, 108, 114, 127, 130
- `optimizers` (*dicee.trainer.dice\_trainer.EnsembleKGE attribute*), 152
- `ordered_bpe_entities` (*dicee.BPE\_NegativeSamplingDataset attribute*), 194
- `ordered_bpe_entities` (*dicee.dataset\_classes.BPE\_NegativeSamplingDataset attribute*), 29
- `ordered_bpe_entities` (*dicee.knowledge\_graph.KG attribute*), 46
- `ordered_shaped_bpe_tokens` (*dicee.knowledge\_graph.KG attribute*), 45

## P

- `p` (*dicee.config.Namespace attribute*), 28
- `p` (*dicee.DeCaL attribute*), 167
- `p` (*dicee.Keci attribute*), 163
- `p` (*dicee.models.clifford.DeCaL attribute*), 63
- `p` (*dicee.models.clifford.Keci attribute*), 60
- `p` (*dicee.models.DeCaL attribute*), 123
- `p` (*dicee.models.Keci attribute*), 120
- `padding` (*dicee.knowledge\_graph.KG attribute*), 46
- `pandas_dataframe_indexer()` (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 141
- `param_init` (*dicee.BaseKGE attribute*), 184
- `param_init` (*dicee.models.base\_model.BaseKGE attribute*), 57
- `param_init` (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130
- `parameters()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
- `path` (*dicee.abstracts.AbstractPPECallback attribute*), 17
- `path` (*dicee.callbacks.AccumulateEpochLossCallback attribute*), 20
- `path` (*dicee.callbacks.ASWA attribute*), 22
- `path` (*dicee.callbacks.Eval attribute*), 23
- `path` (*dicee.callbacks.KGESaveCallback attribute*), 21
- `path_dataset_folder` (*dicee.analyse\_experiments.Experiment attribute*), 18
- `path_for_deserialization` (*dicee.knowledge\_graph.KG attribute*), 45
- `path_for_serialization` (*dicee.knowledge\_graph.KG attribute*), 45
- `path_single_kg` (*dicee.config.Namespace attribute*), 26
- `path_single_kg` (*dicee.knowledge\_graph.KG attribute*), 45
- `path_to_store_single_run` (*dicee.config.Namespace attribute*), 26
- `Perturb` (*class in dicee.callbacks*), 25
- `polars_dataframe_indexer()` (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 140
- `poly_NN()` (*dicee.LFMult method*), 178
- `poly_NN()` (*dicee.models.function\_space.LFMult method*), 72
- `poly_NN()` (*dicee.models.LFMult method*), 134
- `polynomial()` (*dicee.LFMult method*), 179
- `polynomial()` (*dicee.models.function\_space.LFMult method*), 73
- `polynomial()` (*dicee.models.LFMult method*), 134
- `pop()` (*dicee.LFMult method*), 179
- `pop()` (*dicee.models.function\_space.LFMult method*), 73
- `pop()` (*dicee.models.LFMult method*), 134
- `pq` (*dicee.analyse\_experiments.Experiment attribute*), 18
- `predict()` (*dicee.KGE method*), 189

predict () (*dicee.knowledge\_graph\_embeddings.KGE method*), 48  
 predict\_data\_loader () (*dicee.models.base\_model.BaseKGELightning method*), 54  
 predict\_data\_loader () (*dicee.models.BaseKGELightning method*), 94  
 predict\_missing\_head\_entity () (*dicee.KGE method*), 188  
 predict\_missing\_head\_entity () (*dicee.knowledge\_graph\_embeddings.KGE method*), 47  
 predict\_missing\_relations () (*dicee.KGE method*), 189  
 predict\_missing\_relations () (*dicee.knowledge\_graph\_embeddings.KGE method*), 47  
 predict\_missing\_tail\_entity () (*dicee.KGE method*), 189  
 predict\_missing\_tail\_entity () (*dicee.knowledge\_graph\_embeddings.KGE method*), 47  
 predict\_topk () (*dicee.KGE method*), 189  
 predict\_topk () (*dicee.knowledge\_graph\_embeddings.KGE method*), 48  
 prepare\_data () (*dicee.CVDataModule method*), 203  
 prepare\_data () (*dicee.dataset\_classes.CVDataModule method*), 39  
 preprocess\_with\_byte\_pair\_encoding () (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method*), 143  
 preprocess\_with\_byte\_pair\_encoding () (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method*), 137  
 preprocess\_with\_byte\_pair\_encoding\_with\_padding () (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method*), 143  
 preprocess\_with\_byte\_pair\_encoding\_with\_padding () (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method*), 138  
 preprocess\_with\_pandas () (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method*), 143  
 preprocess\_with\_pandas () (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method*), 138  
 preprocess\_with\_polars () (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method*), 144  
 preprocess\_with\_polars () (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method*), 138  
 preprocesses\_input\_args () (*in module dicee.static\_preprocess\_funcs*), 151  
 PreprocessKG (*class in dicee.read\_preprocess\_save\_load\_kg*), 143  
 PreprocessKG (*class in dicee.read\_preprocess\_save\_load\_kg.preprocess*), 137  
 PrintCallback (*class in dicee.callbacks*), 20  
 process (*dicee.trainer.torch\_trainer.TorchTrainer attribute*), 156  
 process (*dicee.trainer.torch\_trainer.xMP attribute*), 155  
 PseudoLabellingCallback (*class in dicee.callbacks*), 22  
 Pyke (*class in dicee*), 162  
 Pyke (*class in dicee.models*), 102  
 Pyke (*class in dicee.models.real*), 82  
 pykeen\_model\_kwargs (*dicee.config.Namespace attribute*), 27  
 PykeenKGE (*class in dicee*), 179  
 PykeenKGE (*class in dicee.models*), 128  
 PykeenKGE (*class in dicee.models.pykeen\_models*), 76

## Q

q (*dicee.config.Namespace attribute*), 28  
 q (*dicee.DeCaL attribute*), 167  
 q (*dicee.Keci attribute*), 163  
 q (*dicee.models.clifford.DeCaL attribute*), 63  
 q (*dicee.models.clifford.Keci attribute*), 60  
 q (*dicee.models.DeCaL attribute*), 123  
 q (*dicee.models.Keci attribute*), 120  
 qdrant\_client (*dicee.scripts.serve.NeuralSearcher attribute*), 146  
 QMult (*class in dicee*), 175  
 QMult (*class in dicee.models*), 110  
 QMult (*class in dicee.models.quaternion*), 78  
 quaternion\_mul () (*in module dicee.models*), 107  
 quaternion\_mul () (*in module dicee.models.static\_funcs*), 82  
 quaternion\_mul\_with\_unit\_norm () (*in module dicee.models*), 110  
 quaternion\_mul\_with\_unit\_norm () (*in module dicee.models.quaternion*), 78  
 quaternion\_multiplication\_followed\_by\_inner\_product () (*dicee.models.QMult method*), 111  
 quaternion\_multiplication\_followed\_by\_inner\_product () (*dicee.models.quaternion.QMult method*), 78  
 quaternion\_multiplication\_followed\_by\_inner\_product () (*dicee.QMult method*), 176  
 quaternion\_normalizer () (*dicee.models.QMult static method*), 111  
 quaternion\_normalizer () (*dicee.models.quaternion.QMult static method*), 79  
 quaternion\_normalizer () (*dicee.QMult static method*), 176  
 query\_name\_to\_struct (*dicee.query\_generator.QueryGenerator attribute*), 136  
 query\_name\_to\_struct (*dicee.QueryGenerator attribute*), 205  
 QueryGenerator (*class in dicee*), 204  
 QueryGenerator (*class in dicee.query\_generator*), 136

## R

r (*dicee.DeCaL attribute*), 167  
 r (*dicee.Keci attribute*), 163  
 r (*dicee.models.clifford.DeCaL attribute*), 63



- `r` (*dicee.models.clifford.Keci* attribute), 60
- `r` (*dicee.models.DeCaL* attribute), 123
- `r` (*dicee.models.Keci* attribute), 120
- `random_prediction()` (in module *dicee*), 186
- `random_prediction()` (in module *dicee.static\_funcs*), 149
- `random_seed` (*dicee.config.Namespace* attribute), 27
- `ratio` (*dicee.callbacks.Perturb* attribute), 25
- `re` (*dicee.DeCaL* attribute), 167
- `re` (*dicee.models.clifford.DeCaL* attribute), 63
- `re` (*dicee.models.DeCaL* attribute), 123
- `re_vocab` (*dicee.evaluator.Evaluator* attribute), 41
- `read_from_disk()` (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 142
- `read_from_triple_store()` (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 142
- `read_only_few` (*dicee.config.Namespace* attribute), 27
- `read_only_few` (*dicee.knowledge\_graph.KG* attribute), 45
- `read_or_load_kg()` (in module *dicee*), 186
- `read_or_load_kg()` (in module *dicee.static\_funcs*), 149
- `read_with_pandas()` (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 142
- `read_with_polars()` (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 142
- `ReadFromDisk` (class in *dicee.read\_preprocess\_save\_load\_kg*), 144
- `ReadFromDisk` (class in *dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk*), 138
- `rel2id` (*dicee.query\_generator.QueryGenerator* attribute), 136
- `rel2id` (*dicee.QueryGenerator* attribute), 205
- `relation_embeddings` (*dicee.AConvQ* attribute), 173
- `relation_embeddings` (*dicee.ConvQ* attribute), 173
- `relation_embeddings` (*dicee.DeCaL* attribute), 167
- `relation_embeddings` (*dicee.DualE* attribute), 170
- `relation_embeddings` (*dicee.LFMult* attribute), 178
- `relation_embeddings` (*dicee.models.AConvQ* attribute), 113
- `relation_embeddings` (*dicee.models.clifford.DeCaL* attribute), 63
- `relation_embeddings` (*dicee.models.ConvQ* attribute), 112
- `relation_embeddings` (*dicee.models.DeCaL* attribute), 123
- `relation_embeddings` (*dicee.models.DualE* attribute), 135
- `relation_embeddings` (*dicee.models.dualE.DualE* attribute), 69
- `relation_embeddings` (*dicee.models.FMult* attribute), 132
- `relation_embeddings` (*dicee.models.FMult2* attribute), 133
- `relation_embeddings` (*dicee.models.function\_space.FMult* attribute), 70
- `relation_embeddings` (*dicee.models.function\_space.FMult2* attribute), 71
- `relation_embeddings` (*dicee.models.function\_space.GFMult* attribute), 71
- `relation_embeddings` (*dicee.models.function\_space.LFMult* attribute), 72
- `relation_embeddings` (*dicee.models.function\_space.LFMult1* attribute), 72
- `relation_embeddings` (*dicee.models.GFMult* attribute), 132
- `relation_embeddings` (*dicee.models.LFMult* attribute), 134
- `relation_embeddings` (*dicee.models.LFMult1* attribute), 133
- `relation_embeddings` (*dicee.models.pykeen\_models.PykeenKGE* attribute), 77
- `relation_embeddings` (*dicee.models.PykeenKGE* attribute), 128
- `relation_embeddings` (*dicee.models.quaternion.AConvQ* attribute), 80
- `relation_embeddings` (*dicee.models.quaternion.ConvQ* attribute), 79
- `relation_embeddings` (*dicee.PykeenKGE* attribute), 180
- `relation_to_idx` (*dicee.knowledge\_graph.KG* attribute), 45
- `relations_str` (*dicee.knowledge\_graph.KG* property), 46
- `reload_dataset()` (in module *dicee*), 193
- `reload_dataset()` (in module *dicee.dataset\_classes*), 29
- `report` (*dicee.DICE\_Trainer* attribute), 187
- `report` (*dicee.evaluator.Evaluator* attribute), 41
- `report` (*dicee.Execute* attribute), 192
- `report` (*dicee.executer.Execute* attribute), 43
- `report` (*dicee.trainer.DICE\_Trainer* attribute), 158
- `report` (*dicee.trainer.dice\_trainer.DICE\_Trainer* attribute), 152
- `reports` (*dicee.callbacks.Eval* attribute), 23
- `requires_grad_for_interactions` (*dicee.Keci* attribute), 163
- `requires_grad_for_interactions` (*dicee.KeciBase* attribute), 162
- `requires_grad_for_interactions` (*dicee.models.clifford.Keci* attribute), 60
- `requires_grad_for_interactions` (*dicee.models.clifford.KeciBase* attribute), 62
- `requires_grad_for_interactions` (*dicee.models.Keci* attribute), 120
- `requires_grad_for_interactions` (*dicee.models.KeciBase* attribute), 122
- `resid_dropout` (*dicee.models.transformers.CausalSelfAttention* attribute), 86
- `residual_convolution()` (*dicee.AConEx* method), 172

`residual_convolution()` (*dicее.ACovO method*), 172  
`residual_convolution()` (*dicее.ACovQ method*), 173  
`residual_convolution()` (*dicее.ConEx method*), 175  
`residual_convolution()` (*dicее.ConvO method*), 174  
`residual_convolution()` (*dicее.ConvQ method*), 173  
`residual_convolution()` (*dicее.models.AConEx method*), 106  
`residual_convolution()` (*dicее.models.ACovO method*), 119  
`residual_convolution()` (*dicее.models.ACovQ method*), 113  
`residual_convolution()` (*dicее.models.complex.AConEx method*), 67  
`residual_convolution()` (*dicее.models.complex.ConEx method*), 67  
`residual_convolution()` (*dicее.models.ConEx method*), 105  
`residual_convolution()` (*dicее.models.ConvO method*), 118  
`residual_convolution()` (*dicее.models.ConvQ method*), 112  
`residual_convolution()` (*dicее.models.octonion.ACovO method*), 76  
`residual_convolution()` (*dicее.models.octonion.ConvO method*), 75  
`residual_convolution()` (*dicее.models.quaternion.ACovQ method*), 80  
`residual_convolution()` (*dicее.models.quaternion.ConvQ method*), 80  
`retrieve_embeddings()` (*in module dicее.scripts.serve*), 146  
`return_multi_hop_query_results()` (*dicее.KGE method*), 190  
`return_multi_hop_query_results()` (*dicее.knowledge\_graph\_embeddings.KGE method*), 49  
`root()` (*in module dicее.scripts.serve*), 146  
`roots` (*dicее.models.FMult attribute*), 132  
`roots` (*dicее.models.function\_space.FMult attribute*), 70  
`roots` (*dicее.models.function\_space.GFMult attribute*), 71  
`roots` (*dicее.models.GFMult attribute*), 132  
`runtime` (*dicее.analyse\_experiments.Experiment attribute*), 19

## S

`sample_counter` (*dicее.abstracts.AbstractPPECallback attribute*), 17  
`sample_entity()` (*dicее.abstracts.BaseInteractiveKGE method*), 14  
`sample_relation()` (*dicее.abstracts.BaseInteractiveKGE method*), 14  
`sample_triples_ratio` (*dicее.config.Namespace attribute*), 27  
`sample_triples_ratio` (*dicее.knowledge\_graph.KG attribute*), 45  
`sanity_checking_with_arguments()` (*in module dicее.sanity\_checkers*), 145  
`save()` (*dicее.abstracts.BaseInteractiveKGE method*), 14  
`save()` (*dicее.read\_preprocess\_save\_load\_kg.LoadSaveToDisk method*), 144  
`save()` (*dicее.read\_preprocess\_save\_load\_kg.save\_load\_disk.LoadSaveToDisk method*), 139  
`save_checkpoint()` (*dicее.abstracts.AbstractTrainer static method*), 13  
`save_checkpoint_model()` (*in module dicее*), 185  
`save_checkpoint_model()` (*in module dicее.static\_funcs*), 148  
`save_embeddings()` (*in module dicее*), 186  
`save_embeddings()` (*in module dicее.static\_funcs*), 149  
`save_embeddings_as_csv` (*dicее.config.Namespace attribute*), 26  
`save_experiment()` (*dicее.analyse\_experiments.Experiment method*), 19  
`save_model_at_every_epoch` (*dicее.config.Namespace attribute*), 27  
`save_numpy_ndarray()` (*in module dicее*), 185  
`save_numpy_ndarray()` (*in module dicее.read\_preprocess\_save\_load\_kg.util*), 142  
`save_numpy_ndarray()` (*in module dicее.static\_funcs*), 148  
`save_pickle()` (*in module dicее*), 185  
`save_pickle()` (*in module dicее.read\_preprocess\_save\_load\_kg.util*), 143  
`save_pickle()` (*in module dicее.static\_funcs*), 148  
`save_queries()` (*dicее.query\_generator.QueryGenerator method*), 137  
`save_queries()` (*dicее.QueryGenerator method*), 205  
`save_queries_and_answers()` (*dicее.query\_generator.QueryGenerator static method*), 137  
`save_queries_and_answers()` (*dicее.QueryGenerator static method*), 205  
`save_trained_model()` (*dicее.Execute method*), 192  
`save_trained_model()` (*dicее.executer.Execute method*), 43  
`scalar_batch_NN()` (*dicее.LFMult method*), 179  
`scalar_batch_NN()` (*dicее.models.function\_space.LFMult method*), 72  
`scalar_batch_NN()` (*dicее.models.LFMult method*), 134  
`scaler` (*dicее.callbacks.Perturb attribute*), 25  
`scaler` (*dicее.trainer.torch\_trainer\_ddp.NodeTrainer attribute*), 157  
`score()` (*dicее.ComplEx static method*), 171  
`score()` (*dicее.DistMult method*), 162  
`score()` (*dicее.Keci method*), 165  
`score()` (*dicее.models.clifford.Keci method*), 62  
`score()` (*dicее.models.ComplEx static method*), 107

`score()` (*dicee.models.complex.ComplEx static method*), 68  
`score()` (*dicee.models.DistMult method*), 102  
`score()` (*dicee.models.Keci method*), 122  
`score()` (*dicee.models.octonion.OMult method*), 74  
`score()` (*dicee.models.OMult method*), 117  
`score()` (*dicee.models.QMult method*), 111  
`score()` (*dicee.models.quaternion.QMult method*), 79  
`score()` (*dicee.models.real.DistMult method*), 81  
`score()` (*dicee.models.real.TransE method*), 81  
`score()` (*dicee.models.TransE method*), 102  
`score()` (*dicee.OMult method*), 178  
`score()` (*dicee.QMult method*), 176  
`score()` (*dicee.TransE method*), 166  
`score_func` (*dicee.models.FMult2 attribute*), 133  
`score_func` (*dicee.models.function\_space.FMult2 attribute*), 71  
`scoring_technique` (*dicee.analyse\_experiments.Experiment attribute*), 19  
`scoring_technique` (*dicee.config.Namespace attribute*), 27  
`search()` (*dicee.scripts.serve.NeuralSearcher method*), 146  
`search_embeddings()` (in module *dicee.scripts.serve*), 146  
`seed` (*dicee.query\_generator.QueryGenerator attribute*), 136  
`seed` (*dicee.QueryGenerator attribute*), 204  
`select_model()` (in module *dicee*), 185  
`select_model()` (in module *dicee.static\_funcs*), 148  
`selected_optimizer` (*dicee.BaseKGE attribute*), 183  
`selected_optimizer` (*dicee.models.base\_model.BaseKGE attribute*), 57  
`selected_optimizer` (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130  
`separator` (*dicee.config.Namespace attribute*), 27  
`separator` (*dicee.knowledge\_graph.KG attribute*), 46  
`sequential_vocabulary_construction()` (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method*), 144  
`sequential_vocabulary_construction()` (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method*), 138  
`set_global_seed()` (*dicee.query\_generator.QueryGenerator method*), 136  
`set_global_seed()` (*dicee.QueryGenerator method*), 205  
`set_model_eval_mode()` (*dicee.abstracts.BaseInteractiveKGE method*), 14  
`set_model_train_mode()` (*dicee.abstracts.BaseInteractiveKGE method*), 14  
`setup()` (*dicee.CVDataModule method*), 202  
`setup()` (*dicee.dataset\_classes.CVDataModule method*), 38  
`setup_executor()` (*dicee.Execute method*), 192  
`setup_executor()` (*dicee.executer.Execute method*), 43  
`Shallom` (class in *dicee*), 178  
`Shallom` (class in *dicee.models*), 102  
`Shallom` (class in *dicee.models.real*), 81  
`shallom` (*dicee.models.real.Shallom attribute*), 81  
`shallom` (*dicee.models.Shallom attribute*), 102  
`shallom` (*dicee.Shallom attribute*), 178  
`single_hop_query_answering()` (*dicee.KGE method*), 190  
`single_hop_query_answering()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 49  
`sparql_endpoint` (*dicee.config.Namespace attribute*), 26  
`sparql_endpoint` (*dicee.knowledge\_graph.KG attribute*), 45  
`start()` (*dicee.DICE\_Trainer method*), 187  
`start()` (*dicee.Execute method*), 193  
`start()` (*dicee.executer.Execute method*), 44  
`start()` (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method*), 143  
`start()` (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method*), 137  
`start()` (*dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk.ReadFromDisk method*), 138  
`start()` (*dicee.read\_preprocess\_save\_load\_kg.ReadFromDisk method*), 144  
`start()` (*dicee.trainer.DICE\_Trainer method*), 159  
`start()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 153  
`start_time` (*dicee.callbacks.PrintCallback attribute*), 20  
`start_time` (*dicee.Execute attribute*), 192  
`start_time` (*dicee.executer.Execute attribute*), 43  
`storage_path` (*dicee.config.Namespace attribute*), 26  
`storage_path` (*dicee.DICE\_Trainer attribute*), 187  
`storage_path` (*dicee.trainer.DICE\_Trainer attribute*), 158  
`storage_path` (*dicee.trainer.dice\_trainer.DICE\_Trainer attribute*), 152  
`store()` (in module *dicee*), 185  
`store()` (in module *dicee.static\_funcs*), 148  
`store_ensemble()` (*dicee.abstracts.AbstractPPECallback method*), 17  
`strategy` (*dicee.abstracts.AbstractTrainer attribute*), 12



swa (*dicee.config.Namespace* attribute), 28

## T

`T()` (*dicee.DualE* method), 170  
`T()` (*dicee.models.DualE* method), 135  
`T()` (*dicee.models.dualE.DualE* method), 70  
`t_conorm()` (*dicee.KGE* method), 190  
`t_conorm()` (*dicee.knowledge\_graph\_embeddings.KGE* method), 49  
`t_norm()` (*dicee.KGE* method), 190  
`t_norm()` (*dicee.knowledge\_graph\_embeddings.KGE* method), 49  
`target_dim` (*dicee.AllvsAll* attribute), 197  
`target_dim` (*dicee.dataset\_classes.AllvsAll* attribute), 33  
`target_dim` (*dicee.dataset\_classes.MultiLabelDataset* attribute), 30  
`target_dim` (*dicee.dataset\_classes.OnevsAllDataset* attribute), 31  
`target_dim` (*dicee.knowledge\_graph.KG* attribute), 46  
`target_dim` (*dicee.MultiLabelDataset* attribute), 194  
`target_dim` (*dicee.OnevsAllDataset* attribute), 195  
`temperature` (*dicee.BytE* attribute), 181  
`temperature` (*dicee.models.transformers.BytE* attribute), 83  
`tensor_t_norm()` (*dicee.KGE* method), 190  
`tensor_t_norm()` (*dicee.knowledge\_graph\_embeddings.KGE* method), 49  
`test_dataloader()` (*dicee.models.base\_model.BaseKGELightning* method), 53  
`test_dataloader()` (*dicee.models.BaseKGELightning* method), 92  
`test_epoch_end()` (*dicee.models.base\_model.BaseKGELightning* method), 53  
`test_epoch_end()` (*dicee.models.BaseKGELightning* method), 92  
`test_h1` (*dicee.analyse\_experiments.Experiment* attribute), 19  
`test_h3` (*dicee.analyse\_experiments.Experiment* attribute), 19  
`test_h10` (*dicee.analyse\_experiments.Experiment* attribute), 19  
`test_mrr` (*dicee.analyse\_experiments.Experiment* attribute), 19  
`test_path` (*dicee.query\_generator.QueryGenerator* attribute), 136  
`test_path` (*dicee.QueryGenerator* attribute), 204  
`timeit()` (*in module dicee*), 185, 193  
`timeit()` (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 142  
`timeit()` (*in module dicee.static\_funcs*), 148  
`timeit()` (*in module dicee.static\_preprocess\_funcs*), 151  
`to()` (*dicee.KGE* method), 188  
`to()` (*dicee.knowledge\_graph\_embeddings.KGE* method), 46  
`to_df()` (*dicee.analyse\_experiments.Experiment* method), 19  
`topk` (*dicee.BytE* attribute), 181  
`topk` (*dicee.models.transformers.BytE* attribute), 83  
`torch_ordered_shaped_bpe_entities` (*dicee.dataset\_classes.MultiLabelDataset* attribute), 30  
`torch_ordered_shaped_bpe_entities` (*dicee.MultiLabelDataset* attribute), 194  
`TorchDDPTrainer` (class in *dicee.trainer.torch\_trainer\_ddp*), 156  
`TorchTrainer` (class in *dicee.trainer.torch\_trainer*), 155  
`train()` (*dicee.KGE* method), 192  
`train()` (*dicee.knowledge\_graph\_embeddings.KGE* method), 50  
`train()` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* method), 157  
`train_data` (*dicee.AllvsAll* attribute), 197  
`train_data` (*dicee.dataset\_classes.AllvsAll* attribute), 32  
`train_data` (*dicee.dataset\_classes.KvsAll* attribute), 32  
`train_data` (*dicee.dataset\_classes.KvsSampleDataset* attribute), 35  
`train_data` (*dicee.dataset\_classes.MultiClassClassificationDataset* attribute), 30  
`train_data` (*dicee.dataset\_classes.OnevsAllDataset* attribute), 31  
`train_data` (*dicee.dataset\_classes.OnevsSample* attribute), 33, 34  
`train_data` (*dicee.KvsAll* attribute), 196  
`train_data` (*dicee.KvsSampleDataset* attribute), 199  
`train_data` (*dicee.MultiClassClassificationDataset* attribute), 195  
`train_data` (*dicee.OnevsAllDataset* attribute), 195  
`train_data` (*dicee.OnevsSample* attribute), 197, 198  
`train_dataloader()` (*dicee.CVDataModule* method), 201  
`train_dataloader()` (*dicee.dataset\_classes.CVDataModule* method), 37  
`train_dataloader()` (*dicee.models.base\_model.BaseKGELightning* method), 54  
`train_dataloader()` (*dicee.models.BaseKGELightning* method), 94  
`train_dataloaders` (*dicee.trainer.torch\_trainer.TorchTrainer* attribute), 155  
`train_dataloaders` (*dicee.trainer.torch\_trainer.xMP* attribute), 155  
`train_dataset_loader` (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 157  
`train_h1` (*dicee.analyse\_experiments.Experiment* attribute), 18

train\_h3 (*dicee.analyse\_experiments.Experiment* attribute), 18  
 train\_h10 (*dicee.analyse\_experiments.Experiment* attribute), 18  
 train\_indices\_target (*dicee.dataset\_classes.MultiLabelDataset* attribute), 30  
 train\_indices\_target (*dicee.MultiLabelDataset* attribute), 194  
 train\_k\_vs\_all () (*dicee.KGE* method), 191  
 train\_k\_vs\_all () (*dicee.knowledge\_graph\_embeddings.KGE* method), 50  
 train\_mrr (*dicee.analyse\_experiments.Experiment* attribute), 18  
 train\_path (*dicee.query\_generator.QueryGenerator* attribute), 136  
 train\_path (*dicee.QueryGenerator* attribute), 204  
 train\_set (*dicee.BPE\_NegativeSamplingDataset* attribute), 194  
 train\_set (*dicee.dataset\_classes.BPE\_NegativeSamplingDataset* attribute), 29  
 train\_set (*dicee.dataset\_classes.MultiLabelDataset* attribute), 30  
 train\_set (*dicee.dataset\_classes.NegSampleDataset* attribute), 35  
 train\_set (*dicee.dataset\_classes.TriplePredictionDataset* attribute), 36  
 train\_set (*dicee.MultiLabelDataset* attribute), 194  
 train\_set (*dicee.NegSampleDataset* attribute), 200  
 train\_set (*dicee.TriplePredictionDataset* attribute), 201  
 train\_set\_idx (*dicee.CVDDataModule* attribute), 201  
 train\_set\_idx (*dicee.dataset\_classes.CVDDataModule* attribute), 37  
 train\_set\_target (*dicee.knowledge\_graph.KG* attribute), 46  
 train\_target (*dicee.AllvsAll* attribute), 197  
 train\_target (*dicee.dataset\_classes.AllvsAll* attribute), 32  
 train\_target (*dicee.dataset\_classes.KvsAll* attribute), 32  
 train\_target (*dicee.dataset\_classes.KvsSampleDataset* attribute), 35  
 train\_target (*dicee.KvsAll* attribute), 196  
 train\_target (*dicee.KvsSampleDataset* attribute), 199  
 train\_target\_indices (*dicee.knowledge\_graph.KG* attribute), 46  
 train\_triples () (*dicee.KGE* method), 191  
 train\_triples () (*dicee.knowledge\_graph\_embeddings.KGE* method), 50  
 trained\_model (*dicee.Execute* attribute), 192  
 trained\_model (*dicee.executer.Execute* attribute), 43  
 trainer (*dicee.config.Namespace* attribute), 27  
 trainer (*dicee.DICE\_Trainer* attribute), 187  
 trainer (*dicee.Execute* attribute), 192  
 trainer (*dicee.executer.Execute* attribute), 43  
 trainer (*dicee.trainer.DICE\_Trainer* attribute), 158  
 trainer (*dicee.trainer.dice\_trainer.DICE\_Trainer* attribute), 152  
 trainer (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer* attribute), 157  
 training\_step (*dicee.trainer.torch\_trainer.TorchTrainer* attribute), 155  
 training\_step (*dicee.trainer.torch\_trainer.xMP* attribute), 155  
 training\_step () (*dicee.BytE* method), 181  
 training\_step () (*dicee.models.base\_model.BaseKGELightning* method), 51  
 training\_step () (*dicee.models.BaseKGELightning* method), 91  
 training\_step () (*dicee.models.transformers.BytE* method), 84  
 training\_step\_outputs (*dicee.models.base\_model.BaseKGELightning* attribute), 51  
 training\_step\_outputs (*dicee.models.BaseKGELightning* attribute), 91  
 training\_technique (*dicee.knowledge\_graph.KG* attribute), 45  
 TransE (*class in dicee*), 166  
 TransE (*class in dicee.models*), 102  
 TransE (*class in dicee.models.real*), 81  
 transfer\_batch\_to\_device () (*dicee.CVDDataModule* method), 202  
 transfer\_batch\_to\_device () (*dicee.dataset\_classes.CVDDataModule* method), 38  
 transformer (*dicee.BytE* attribute), 181  
 transformer (*dicee.models.transformers.BytE* attribute), 84  
 transformer (*dicee.models.transformers.GPT* attribute), 89  
 trapezoid () (*dicee.models.FMult2* method), 133  
 trapezoid () (*dicee.models.function\_space.FMult2* method), 71  
 tri\_score () (*dicee.LFMult* method), 179  
 tri\_score () (*dicee.models.function\_space.LFMult* method), 72  
 tri\_score () (*dicee.models.function\_space.LFMult1* method), 72  
 tri\_score () (*dicee.models.LFMult* method), 134  
 tri\_score () (*dicee.models.LFMult1* method), 133  
 triple\_score () (*dicee.KGE* method), 190  
 triple\_score () (*dicee.knowledge\_graph\_embeddings.KGE* method), 48  
 TriplePredictionDataset (*class in dicee*), 200  
 TriplePredictionDataset (*class in dicee.dataset\_classes*), 36  
 tuple2list () (*dicee.query\_generator.QueryGenerator* method), 136  
 tuple2list () (*dicee.QueryGenerator* method), 205

## U

`unlabelled_size` (*dicee.callbacks.PseudoLabellingCallback attribute*), 22  
`unmap()` (*dicee.query\_generator.QueryGenerator method*), 137  
`unmap()` (*dicee.QueryGenerator method*), 205  
`unmap_query()` (*dicee.query\_generator.QueryGenerator method*), 137  
`unmap_query()` (*dicee.QueryGenerator method*), 205

## V

`val_aswa` (*dicee.callbacks.ASWA attribute*), 23  
`val_data_loader()` (*dicee.models.base\_model.BaseKGELighting method*), 53  
`val_data_loader()` (*dicee.models.BaseKGELighting method*), 93  
`val_h1` (*dicee.analyse\_experiments.Experiment attribute*), 18  
`val_h3` (*dicee.analyse\_experiments.Experiment attribute*), 18  
`val_h10` (*dicee.analyse\_experiments.Experiment attribute*), 18  
`val_mrr` (*dicee.analyse\_experiments.Experiment attribute*), 18  
`val_path` (*dicee.query\_generator.QueryGenerator attribute*), 136  
`val_path` (*dicee.QueryGenerator attribute*), 204  
`validate_knowledge_graph()` (in module *dicee.sanity\_checkers*), 144  
`vocab_preparation()` (*dicee.evaluator.Evaluator method*), 42  
`vocab_size` (*dicee.models.transformers.GPTConfig attribute*), 88  
`vocab_to_parquet()` (in module *dicee*), 186  
`vocab_to_parquet()` (in module *dicee.static\_funcs*), 149  
`vtp_score()` (*dicee.LFMMult method*), 179  
`vtp_score()` (*dicee.models.function\_space.LFMMult method*), 73  
`vtp_score()` (*dicee.models.function\_space.LFMMultI method*), 72  
`vtp_score()` (*dicee.models.LFMMult method*), 134  
`vtp_score()` (*dicee.models.LFMMultI method*), 133

## W

`weight` (*dicee.models.transformers.LayerNorm attribute*), 85  
`weight_decay` (*dicee.BaseKGE attribute*), 183  
`weight_decay` (*dicee.config.Namespace attribute*), 27  
`weight_decay` (*dicee.models.base\_model.BaseKGE attribute*), 57  
`weight_decay` (*dicee.models.BaseKGE attribute*), 97, 100, 104, 108, 114, 127, 130  
`weights` (*dicee.models.FMMult attribute*), 132  
`weights` (*dicee.models.function\_space.FMMult attribute*), 70  
`weights` (*dicee.models.function\_space.GFMMult attribute*), 71  
`weights` (*dicee.models.GFMMult attribute*), 132  
`write_links()` (*dicee.query\_generator.QueryGenerator method*), 137  
`write_links()` (*dicee.QueryGenerator method*), 205  
`write_report()` (*dicee.Execute method*), 193  
`write_report()` (*dicee.executer.Execute method*), 44

## X

`x_values` (*dicee.LFMMult attribute*), 178  
`x_values` (*dicee.models.function\_space.LFMMult attribute*), 72  
`x_values` (*dicee.models.LFMMult attribute*), 134  
`xMP` (class in *dicee.trainer.torch\_trainer*), 154