
DICE Embeddings

Release 0.1.3.2

Caglar Demir

Jun 12, 2025

Contents:

1	Dicee Manual	2
2	Installation	3
2.1	Installation from Source	3
3	Download Knowledge Graphs	3
4	Knowledge Graph Embedding Models	3
5	How to Train	3
6	Creating an Embedding Vector Database	5
6.1	Learning Embeddings	5
6.2	Loading Embeddings into Qdrant Vector Database	6
6.3	Launching Webservice	6
7	Answering Complex Queries	6
8	Predicting Missing Links	8
9	Downloading Pretrained Models	8
10	How to Deploy	8
11	Docker	8
12	Coverage Report	8
13	How to cite	10
14	dicee	12
14.1	Submodules	12
14.2	Attributes	166
14.3	Classes	166
14.4	Functions	167
14.5	Package Contents	169
	Python Module Index	213

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

1 Dicee Manual

Version: dicee 0.1.3.2

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

Publisher and maintainer: Caglar Demir²

Contact: caglar.demir@upb.de

License: OSI Approved :: MIT License

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

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

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

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

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

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

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

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

² <https://github.com/Demirrr>

³ <https://pandas.pydata.org/>

⁴ <https://pytorch.org/>

⁵ <https://huggingface.co/>

⁶ <https://pandas.pydata.org/>

⁷ <https://pytorch.org/>

⁸ <https://pytorch.org/>

⁹ <https://www.pytorchlightning.ai/>

¹⁰ <https://huggingface.co/gradio>

2 Installation

2.1 Installation from Source

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

or

```
pip install dicee
```

3 Download Knowledge Graphs

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

To test the Installation

```
python -m pytest -p no:warnings -x # Runs >114 tests leading to > 15 mins
python -m pytest -p no:warnings --lf # run only the last failed test
python -m pytest -p no:warnings --ff # to run the failures first and then the rest of
↪ the tests.
```

4 Knowledge Graph Embedding Models

1. TransE, DistMult, ComplEx, ConEx, QMult, OMult, ConvO, ConvQ, Keci
2. All 44 models available in <https://github.com/pykeen/pykeen#models>

For more, please refer to examples.

5 How to Train

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

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

where the data is in the following form

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

A KGE model can also be trained from the command line

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

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

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

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

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

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

Similarly, models can be easily trained with torchrun

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

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

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

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

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

where the data is in the following form

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

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

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

For more, please refer to examples.

6 Creating an Embedding Vector Database

6.1 Learning Embeddings

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

6.2 Loading Embeddings into Qdrant Vector Database

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

6.3 Launching Webservice

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

Retrieve and Search

Get embedding of germany

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

Get most similar things to europe

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

7 Answering Complex Queries

```
# pip install dicee
# wget https://files.dice-research.org/datasets/dice-embeddings/KGs.zip --no-check-
↪certificate & unzip KGs.zip
from dicee.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/¹¹

10 How to Deploy

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

11 Docker

To build the Docker image:

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

To test the Docker image:

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

12 Coverage Report

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

Name	Stmts	Miss	Cover	Missing
-----	-----	-----	-----	-----
dicee/___init___ .py	7	0	100%	
dicee/abstracts.py	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)

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

¹² <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

14 dicee

14.1 Submodules

dicee.__main__

dicee.abstracts

Classes

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

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 = False

apply_semantic_constraint = False

configs

get_eval_report () → dict

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

Parameters

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

Return type

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

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

Parameters

triples

set_model_train_mode () → None

Setting the model into training mode

Parameter

set_model_eval_mode () → None

Setting the model into eval mode

Parameter

property name

sample_entity (n: int) → List[str]

sample_relation (n: int) → List[str]

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

save() \rightarrow None

get_entity_index (*x: str*)

get_relation_index (*x: str*)

index_triple (*head_entity: List[str], relation: List[str], tail_entity: List[str]*)
 \rightarrow Tuple[torch.LongTensor, torch.LongTensor, torch.LongTensor]

Index Triple

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

pytorch tensor of triple score

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

get_entity_embeddings (*items: List[str]*)

Return embedding of an entity given its string representation

Parameter

items:

entities

get_relation_embeddings (*items: List[str]*)

Return embedding of a relation given its string representation

Parameter

items:

relations

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

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

parameters()

class dicee.abstracts.**InteractiveQueryDecomposition**

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

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

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

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

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

class dicee.abstracts.**AbstractCallback**

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

Abstract class for Callback class for knowledge graph embedding models

Parameter

on_init_start (*args, **kwargs)

Parameter

trainer:

model:

rtype

None

on_init_end (*args, **kwargs)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_fit_start (trainer, model)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_train_epoch_end (trainer, model)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype

None

on_train_batch_end (*args, **kwargs)

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

Parameter

trainer:

model:

rtype

None

on_fit_end (*args, **kwargs)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

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

Bases: [AbstractCallback](#)

Abstract class for Callback class for knowledge graph embedding models

Parameter

num_epochs

path

sample_counter = 0

epoch_count = 0

alphas = None

on_fit_start (trainer, model)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_fit_end (trainer, model)

Call at the end of the training.

Parameter

trainer:

model:

rtype
None

store_ensemble (*param_ensemble*) → None

dicee.analyse_experiments

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

Classes

Experiment

Functions

get_default_arguments()

analyse(args)

Module Contents

`dicee.analyse_experiments.get_default_arguments()`

class `dicee.analyse_experiments.Experiment`

```
    model_name = []
    callbacks = []
    embedding_dim = []
    num_params = []
    num_epochs = []
    batch_size = []
    lr = []
    byte_pair_encoding = []
    aswa = []
    path_dataset_folder = []
    full_storage_path = []
    pq = []
    train_mrr = []
```

```

train_h1 = []

train_h3 = []

train_h10 = []

val_mrr = []

val_h1 = []

val_h3 = []

val_h10 = []

test_mrr = []

test_h1 = []

test_h3 = []

test_h10 = []

runtime = []

normalization = []

scoring_technique = []

save_experiment(x)

to_df()

dicee.analyse_experiments.analyse(args)

```

dicee.callbacks

Classes

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

Functions

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

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 updates the ensemble model accordingly.

path

num_epochs

initial_eval_setting = None

epoch_count = 0

alphas = []

val_aswa = -1

on_fit_end(*trainer, model*)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

static compute_mrr(*trainer, model*) → float

get_aswa_state_dict(*model*)

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

Perform Hard Update, software or rejection

Parameters

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

on_train_epoch_end(*trainer, model*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype

None

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

Bases: `dicee.abstracts.AbstractCallback`

Abstract class for Callback class for knowledge graph embedding models

Parameter

path

reports = []

epoch_ratio = None

epoch_counter = 0

on_fit_start (*trainer, model*)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

on_fit_end (*trainer, model*)

Call at the end of the training.

Parameter

trainer:

model:

rtype

None

on_train_epoch_end (*trainer, model*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

rtype

None

on_train_batch_end(*args, **kwargs)

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

Parameter

trainer:

model:

rtype

None

class dicee.callbacks.**KronE**

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

f = None

static batch_kronecker_product(a, b)

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

get_kronecker_triple_representation(indexed_triple: torch.LongTensor)

Get kronecker embeddings

on_fit_start(trainer, model)

Call at the beginning of the training.

Parameter

trainer:

model:

rtype

None

class 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 = 'input'

ratio = 0.0

method = None

scaler = None

frequency = None

on_train_batch_start (*trainer, model, batch, batch_idx*)
 Called when the train batch begins.

dicee.config

Classes

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

Module Contents

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

Bases: `argparse.Namespace`

Simple object for storing attributes.

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

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

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

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

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

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

sparql_endpoint = None
 An endpoint of a triple store.

model: str = 'Keci'
 KGE model

optim: str = 'Adam'
 Optimizer

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

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

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

lr: float = 0.1
 Learning rate

add_noise_rate: float = None

The ratio of added random triples into training dataset

gpus = None

Number GPUs to be used during training

callbacks

10}}

Type

Callbacks, e.g., {"PPE"

Type

{ "last_percent_to_consider"

backend: str = 'pandas'

Backend to read, process, and index input knowledge graph. pandas, polars and rdflib available

separator: str = '\\s+'

separator for extracting head, relation and tail from a triple

trainer: str = 'torchCPUTrainer'

Trainer for knowledge graph embedding model

scoring_technique: str = 'KvsAll'

Scoring technique for knowledge graph embedding models

neg_ratio: int = 0

Negative ratio for a true triple in NegSample training_technique

weight_decay: float = 0.0

Weight decay for all trainable params

normalization: str = 'None'

LayerNorm, BatchNorm1d, or None

init_param: str = None

xavier_normal or None

gradient_accumulation_steps: int = 0

Not tested e

num_folds_for_cv: int = 0

Number of folds for CV

eval_model: str = 'train_val_test'

["None", "train", "train_val", "train_val_test", "test"]

Type

Evaluate trained model choices

save_model_at_every_epoch: int = None

Not tested

label_smoothing_rate: float = 0.0

num_core: int = 0

Number of CPUs to be used in the mini-batch loading process

random_seed: int = 0
Random Seed

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

read_only_few: int = None
Read only first few triples

pykeen_model_kwargs
Additional keyword arguments for pykeen models

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

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

p: int = 0
P parameter of Clifford Embeddings

q: int = 1
Q parameter of Clifford Embeddings

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

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

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

byte_pair_encoding: bool = False
Byte pair encoding

Type
WIP

adaptive_swa: bool = False
Adaptive stochastic weight averaging

swa: bool = False
Stochastic weight averaging

block_size: int = None
block size of LLM

continual_learning = None
Path of a pretrained model size of LLM

auto_batch_finding = False
A flag for using auto batch finding

__iter__()

dicee.dataset_classes

Classes

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

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_idxes** – mapping.
- **relation_idxes** – mapping.
- **form** – ?
- **num_workers** – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

Return type

`torch.utils.data.Dataset`

train_data

block_size = 8

num_of_data_points

collate_fn = None

__len__()

__getitem__(*idx*)

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

Bases: `torch.utils.data.Dataset`

Dataset for the 1vsALL training strategy

Parameters

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

Return type

`torch.utils.data.Dataset`

train_data

target_dim

collate_fn = None

__len__()

__getitem__(*idx*)

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

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

Bases: torch.utils.data.Dataset

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

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

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

Note

TODO

train_set_idx

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

entity_idxes

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

relation_idxes

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

self : torch.utils.data.Dataset

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

train_data = None

train_target = None

label_smoothing_rate

collate_fn = None

__len__()

__getitem__(idx)

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

label_smoothing_rate=0.0)
```

Bases: torch.utils.data.Dataset

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

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

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

Note

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

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

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

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

self : torch.utils.data.Dataset

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

train_data = None

train_target = None

label_smoothing_rate

collate_fn = None

target_dim

__len__()

__getitem__(idx)

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

Bases: torch.utils.data.Dataset

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

Parameters

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

train_data

The input data converted into a PyTorch tensor.

Type

torch.Tensor

num_entities

Number of entities in the dataset.

Type

int

num_relations

Number of relations in the dataset.

Type

int

neg_sample_ratio

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

Type

int

label_smoothing_rate

The smoothing factor applied to the labels.

Type

torch.Tensor

collate_fn

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

Type

function, optional

train_data

num_entities

num_relations

neg_sample_ratio = None

label_smoothing_rate

collate_fn = None

__len__()

Returns the number of samples in the dataset.

__getitem__(idx)

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

Parameters

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

Returns

A tuple consisting of:

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

Return type
tuple

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

Bases: torch.utils.data.Dataset

KvsSample a Dataset:

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

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

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

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

train_set_idx

Indexed triples for the training.

entity_idx

mapping.

relation_idx

mapping.

form

?

store

?

label_smoothing_rate

?

torch.utils.data.Dataset

train_data = None

train_target = None

neg_ratio = None

num_entities

label_smoothing_rate

collate_fn = None

max_num_of_classes

__len__()

__getitem__(idx)

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

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

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

Note

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

`neg_sample_ratio`

`train_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_idx

mapping.

relation_idx

mapping.

form

?

store

?

`label_smoothing_rate`

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

```

label_smoothing_rate
neg_sample_ratio
train_set
length
num_entities
num_relations
__len__()
__getitem__(idx)
collate_fn(batch: List[torch.Tensor])

```

```

class dicee.dataset_classes.CVDataModule(train_set_idx: numpy.ndarray, num_entities,
    num_relations, neg_sample_ratio, batch_size, num_workers)

```

Bases: `pytorch_lightning.LightningDataModule`

Create a Dataset for cross validation

Parameters

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

Return type

?

```

train_set_idx
num_entities
num_relations
neg_sample_ratio
batch_size
num_workers

```

```

train_dataloader() → torch.utils.data.DataLoader

```

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

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

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

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

Note

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

`setup(*args, **kwargs)`

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

Parameters

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

Example:

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

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

        # don't do this
        self.something = else

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

`transfer_batch_to_device(*args, **kwargs)`

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

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

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

- tuple

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

Note

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

Parameters

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

Returns

A reference to the data on the new device.

Example:

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

See also

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

prepare_data (*args, **kwargs)

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

Warning

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

Example:

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

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

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

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

Example:

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

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

This is called before requesting the dataloaders:

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

`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: 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: 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 = None

num_entities = None

num_relations = None

args

report

during_training = False

vocab_preparation (*dataset*) → None

A function to wait future objects for the attributes of executor

Return type

None

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

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

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

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

Evaluate model after reciprocal triples are added

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

Evaluate model after reciprocal triples are added

```

eval_with_vs_all (*, train_set, valid_set=None, test_set=None, trained_model, form_of_labelling)
    → None
    Evaluate model after reciprocal triples are added

evaluate_lp_k_vs_all (model, triple_idx, info=None, form_of_labelling=None)
    Filtered link prediction evaluation. :param model: :param triple_idx: test triples :param info: :param
    form_of_labelling: :return:

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

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

    Parameters

    • model

    • triples (List of lists)

    • info

    • form_of_labelling

evaluate_lp (model, triple_idx, info: str)

dummy_eval (trained_model, form_of_labelling: str)

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

```

dicee.executer

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 = False

trainer = None

trained_model = None

knowledge_graph = None

report

evaluator = None

start_time = None

setup_executor() → 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 = None

sparql_endpoint = None

path_single_kg = None

byte_pair_encoding = False

ordered_shaped_bpe_tokens = None

add_noise_rate = None

num_entities = None

num_relations = None

path_for_deserialization = None

add_reciprocal = None

eval_model = None

read_only_few = None

sample_triples_ratio = None

path_for_serialization = None
```

```

entity_to_idx = None

relation_to_idx = None

backend = 'pandas'

training_technique = None

idx_entity_to_bpe_shaped

enc

num_tokens

num_bpe_entities = None

padding = False

dummy_id

max_length_subword_tokens = None

train_set_target = None

target_dim = None

train_target_indices = None

ordered_bpe_entities = None

separator = None

description_of_input = None

describe() → None

property entities_str: List

property relations_str: List

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

__iter__()

__len__()

func_triple_to_bpe_representation(triple: List[str])

```

`dicee.knowledge_graph_embeddings`

Classes

KGE

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

Module Contents

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

Bases: `dicee.abstracts.BaseInteractiveKGE`, `dicee.abstracts.InteractiveQueryDecomposition`

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

```
__str__()
```

```
to (device: str) → None
```

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

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

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

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

```
predict_missing_head_entity (relation: List[str] | str, tail_entity: List[str] | str, within=None,
      batch_size=2, topk=1, return_indices=False) → Tuple
```

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

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

Parameter

relation: Union[List[str], str]

String representation of selected relations.

tail_entity: Union[List[str], str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

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

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

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

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

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

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

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

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

scores

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

Parameters

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

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

Predict missing item in a given triple.

Returns

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

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

Predict triple score

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

Returns: Tuple

pytorch tensor of triple score

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

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

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

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

Find an answer set for EPFO queries including negation and disjunction

Parameter

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

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

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

tnorm: str The t-norm operator.

neg_norm: str The negation norm.

lambda_: float lambda parameter for sugeno and yager negation norms

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

returns

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

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

Find missing triples

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

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

Return (e,r,x)

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

confidence: float

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

topk: int

Highest ranked k item to select triples with $f(e,r,x) > \text{confidence}$.

at_most: int

Stop after finding at_most missing triples

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

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.

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

Trains the Literal Embeddings model using literal data.

Parameters

- **train_file_path** (*str*) – Path to the training data file.
- **num_epochs** (*int*) – Number of training epochs.
- **lit_lr** (*float*) – Learning rate for the literal model.
- **eval_litreal_preds** (*bool*) – If True, evaluate the model after training.
- **eval_file_path** (*str*) – Path to evaluation data file.
- **norm_type** (*str*) – Normalization type to use ('z-norm', 'min-max', or None).
- **batch_size** (*int*) – Batch size for training.
- **sampling_ratio** (*float*) – Ratio of training triples to use.
- **loader_backend** (*str*) – Backend for loading the dataset ('pandas' or 'rdflib').
- **freeze_entity_embeddings** (*bool*) – If True, freeze the entity embeddings during training.
- **gate_residual** (*bool*) – If True, use gate residual connections in the model.
- **device** (*str*) – Device to use for training ('cuda' or 'cpu'). If None, will use available GPU or CPU.

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

Predicts literal values for given entities and attributes.

Parameters

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

Returns

Predictions for the given entities and attributes.

Return type

numpy ndarray

evaluate_literal_prediction (*eval_file_path: str = None, store_lit_preds: bool = True, eval_literals: bool = True, loader_backend: str = 'pandas', return_attr_error_metrics: bool = False*)

Evaluates the trained literal prediction model on a test file.

Parameters

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

Returns

None

dicee.literal_classes

Classes

<i>GatedLinearUnit</i>	Applies a gated linear unit (GLU) operation:
<i>LiteralEmbeddings</i>	A model for learning and predicting numerical literals using pre-trained KGE.
<i>LiteralDataset</i>	Dataset for loading and processing literal data for training Literal Embedding model.

Module Contents

class `dicee.literal_classes.GatedLinearUnit` (*input_dim, gated_residual=True*)

Bases: `torch.nn.Module`

Applies a gated linear unit (GLU) operation: Splits the input in half along the last dimension, applies a sigmoid gate to one half and multiplies it with the other.

proj

gate_residual = True

forward (*x1, x2*)

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

Bases: `torch.nn.Module`

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

num_of_data_properties

Number of data properties (attributes).

Type

int

embedding_dims

Dimension of the embeddings.

Type

int

entity_embeddings

Pre-trained entity embeddings.

Type

`torch.tensor`

dropout

Dropout rate for regularization.

Type

float

gate_residual

Whether to use gated residual connections.

Type

bool

freeze_entity_embeddings

Whether to freeze the entity embeddings during training.

Type

bool

embedding_dim

num_of_data_properties

hidden_dim

entity_embeddings

data_property_embeddings

fc

fc_out

dropout

residual

layer_norm

forward (*entity_idx*, *attr_idx*)

Parameters

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

Returns

scalar predictions.

Return type

Tensor

property device

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

Bases: torch.utils.data.Dataset

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

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

train_file_path

Path to the training data file.

Type

str

normalization

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

Type

str

normalization_params

Parameters used for normalization.

Type

dict

sampling_ratio

Fraction of the training set to use for ablations.

Type

float

entity_to_idx

Mapping of entities to their indices.

Type

dict

num_entities

Total number of entities.

Type

int

data_property_to_idx

Mapping of data properties to their indices.

Type

dict

num_data_properties

Total number of data properties.

Type

int

loader_backend

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

Type

str

train_file_path

loader_backend = 'pandas'

normalization_type = 'z-norm'

normalization_params

sampling_ratio = None

entity_to_idx = None

num_entities

__getitem__ (*index*)

__len__ ()

static load_and_validate_literal_data (*file_path: str = None, loader_backend: str = 'pandas'*)

→ pandas.DataFrame

Loads and validates the literal data file. :param file_path: Path to the literal data file. :type file_path: str

Returns

DataFrame containing the loaded and validated data.

Return type

pd.DataFrame

static denormalize (*preds_norm, attributes, normalization_params*) → numpy.ndarray

Denormalizes the predictions based on the normalization type.

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

Returns

Denormalized predictions.

Return type

np.ndarray

dicее.models

Submodules

dicее.models.adopt

Classes

ADOPT

Base class for all optimizers.

Functions

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

Module Contents

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

Bases: torch.optim.optimizer.Optimizer

Base class for all optimizers.

Warning

Parameters need to be specified as collections that have a deterministic ordering that is consistent between runs. Examples of objects that don't satisfy those properties are sets and iterators over values of dictionaries.

Parameters

- **params** (*iterable*) – an iterable of `torch.Tensor`s or `dict`s. Specifies what Tensors should be optimized.
- **defaults** – (dict): a dict containing default values of optimization options (used when a parameter group doesn't specify them).

clip_lambda

__setstate__ (*state*)

step (*closure=None*)

Perform a single optimization step.

Parameters

closure (*Callable, optional*) – A closure that reevaluates the model and returns the loss.

```

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

```

Functional API that performs ADOPT algorithm computation.

dicee.models.base_model

Classes

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

Module Contents

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

Bases: `lightning.LightningModule`

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

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

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

```

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

Note

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

Variables

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


```
training_step_outputs = []
```

```
mem_of_model() → Dict
```

Size of model in MB and number of params

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

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

Parameters

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

Returns

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

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

Example:

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

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

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

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

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

Note

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

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

Parameters

- `yhat_batch`
- `y_batch`

`on_train_epoch_end(*args, **kwargs)`

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

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

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

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

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

`test_epoch_end(outputs: List[Any])`

`test_dataloader()` → None

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

- `test()`

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

Note

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

Note

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

`val_dataloader()` → None

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

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

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

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

Note

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

Note

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

`predict_dataloader()` → None

An iterable or collection of iterables specifying prediction samples.

For more information about multiple dataloaders, see this section.

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

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

Note

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

Returns

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

`train_dataloader()` → None

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

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

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

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

Note

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

`configure_optimizers(parameters=None)`

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

Returns

Any of these 6 options.

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

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

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

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

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

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

Note

Some things to know:

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

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

```

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

```

Parameters

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

forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])

byte pair encoded neural link predictors

Parameters

init_params_with_sanity_checking ()

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters

x

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

get_sentence_representation (*x*: torch.LongTensor)

Parameters

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

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

Parameters

x (*B* × 2 × *x* *T*)

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

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

`args = None`

`__call__(x)`

`static forward(x)`

dicee.models.clifford

Classes

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

Module Contents

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

`name = 'Keci'`

`p`

`q`

`r`

`requires_grad_for_interactions = True`

`compute_sigma_pp (hp, rp)`

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

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

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

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

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

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

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

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

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

`compute_sigma_qq (hq, rq)`

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

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

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

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

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

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

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

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

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

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

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

```

    for j in range(q):
        sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
    print(sigma_pq.shape)
apply_coefficients (hp, hq, rp, rq)
    Multiplying a base vector with its scalar coefficient
clifford_multiplication (h0, hp, hq, r0, rp, rq)
    Compute our CL multiplication
    
$$h = h_0 + \sum_{i=1}^p h_i e_i + \sum_{j=p+1}^{p+q} h_j e_j$$


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


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


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

    where
    (1)  $\sigma_0 = h_0 r_0 + \sum_{i=1}^p (h_0 r_i - \sum_{j=p+1}^{p+q} h_j r_j) e_i$ 
    (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 dicee.models.clifford.**CKeci** (*args*)

Bases: *Keci*

Without learning dimension scaling

name = 'CKeci'

requires_grad_for_interactions = False

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

(continues on next page)

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

Note

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

Variables

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

name = 'DeCaL'

entity_embeddings

relation_embeddings

p

q

r

re

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

Parameter

x: torch.LongTensor with (n,) shape

rtype

torch.FloatTensor with (n) shape

cl_pqr (*a: torch.tensor*) → torch.tensor

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

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

compute_sigmas_single (*list_h_emb, list_r_emb, list_t_emb*)

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

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

and return:

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

compute_sigmas_multivect (*list_h_emb, list_r_emb*)

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

For same bases vectors interaction we have

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

For different base vector interactions, we have

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

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

Kvsall training

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

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

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

Multiplying a base vector with its scalar coefficient

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

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

Parameter

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

returns

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

compute_sigma_pp(hp, rp)

Compute .. math:

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

compute_sigma_qq(hq, rq)

Compute

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

compute_sigma_rr(hk, rk)

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

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

Compute

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

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

for j in range(q):

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

```

print(sigma_pq.shape)
compute_sigma_pr(*, hp, hk, rp, rk)
Compute

```

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

```

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

```

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

```

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

```

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

```



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

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

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

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

Parameters

x

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

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

Bases: [dicee.models.base_model.BaseKGE](#)

Additive Convolutional ComplEx Knowledge Graph Embeddings

```
name = 'AConEx'
```

```
conv2d
```

```
fc_num_input
```

```
fc1
```

```
norm_fc1
```

```
bn_conv2d
```

```
feature_map_dropout
```

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

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

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

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

Parameters

x

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

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

Bases: [dicee.models.base_model.BaseKGE](#)

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

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

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

```

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

Note

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

Variables

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

name = 'Complex'

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

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

Parameters

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

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

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

dicee.models.dualE

Classes

DualE

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

Module Contents

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

Bases: `dicee.models.base_model.BaseKGE`

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

name = 'DualE'

entity_embeddings

relation_embeddings

num_ent = None

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

KvsAll scoring function

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

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

Negative Sampling forward pass:

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_k_vs_all (*x*)

KvsAll forward pass

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

T (*x: torch.tensor*) → torch.tensor

Transpose function

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

dicee.models.ensemble

Classes

EnsembleKGE

Module Contents

class dicee.models.ensemble.**EnsembleKGE** (*seed_model=None, pretrained_models: List = None*)

name

train_mode = True

named_children()

property example_input_array

parameters()

modules()

__iter__()

__len__()

eval()

to(device)

mem_of_model()

__call__(x_batch)

step()

get_embeddings()

__str__()

dicee.models.function_space

Classes

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

Module Contents

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

Parameters

x

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

Parameters

x

```

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

```

Parameters

x

```

class dicee.models.function_space.LFMult1(args)

```

Bases: *dicee.models.base_model.BaseKGE*

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

```

name = 'LFMult1'
entity_embeddings
relation_embeddings
forward_triples(idx_triple)

```

Parameters

x

```

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

```

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

Bases: *dicee.models.base_model.BaseKGE*

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

```
name = 'LFMult'
```

```
entity_embeddings
```

```
relation_embeddings
```

```
degree
```

```
m
```

```
x_values
```

```
forward_triples(idx_triple)
```

Parameters

x

```
construct_multi_coeff(x)
```

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

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

```
linear(x, w, b)
```

```
scalar_batch_NN(a, b, c)
```

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

```
tri_score(coeff_h, coeff_r, coeff_t)
```

this part implement the trilinear scoring techniques:

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

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

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

this part implement the vector triple product scoring techniques:

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

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

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

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

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

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

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

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

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

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

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

dicee.models.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 Embeddings

Functions

<i>octonion_mul</i> (*, <i>O_1</i> , <i>O_2</i>)
<i>octonion_mul_norm</i> (*, <i>O_1</i> , <i>O_2</i>)

Module Contents

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

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

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)


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

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

Note

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

Variables

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

name = '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, |Entities|)

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

name = 'ConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

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

residual_convolution (*O_1, O_2*)

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

Parameters

x

forward_k_vs_all (*x: torch.Tensor*)

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

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

Bases: `dicee.models.base_model.BaseKGE`

Additive Convolutional Octonion Knowledge Graph Embeddings

name = 'AConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

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

```
residual_convolution(O_1, O_2)
```

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

Parameters

x

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

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

dicee.models.pykeen_models

Classes

PykeenKGE

A class for using knowledge graph embedding models implemented in Pykeen

Module Contents

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

Bases: *dicee.models.base_model.BaseKGE*

A class for using knowledge graph embedding models implemented in Pykeen

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

model_kwargs

name

model

loss_history = []

args

entity_embeddings = None

relation_embeddings = None

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

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

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

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

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

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

```

else:
    t = self.entity_embeddings.weight

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

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

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

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

abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)

```

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 them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

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

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

```

(continues on next page)

(continued from previous page)

```
self.conv1 = nn.Conv2d(1, 20, 5)
self.conv2 = nn.Conv2d(20, 20, 5)

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

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

Note

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

Variables

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

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

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

Bases: `dicee.models.base_model.BaseKGE`

Additive Convolutional Quaternion Knowledge Graph Embeddings

`name = 'AConvQ'`

```

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution(Q_1, Q_2)

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

```

Parameters

x

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

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

dicee.models.real

Classes

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

Module Contents

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

Bases: *dicee.models.base_model.BaseKGE*

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

```
name = 'DistMult'
```

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

Parameters

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

```

forward_k_vs_all (x: torch.LongTensor)

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

score (h, r, t)

class dicee.models.real.TransE (args)
    Bases: dicee.models.base_model.BaseKGE
    Translating Embeddings for Modeling Multi-relational Data https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf
    name = 'TransE'
    margin = 4
    score (head_ent_emb, rel_ent_emb, tail_ent_emb)
    forward_k_vs_all (x: torch.Tensor) → torch.FloatTensor

class dicee.models.real.Shallom (args)
    Bases: dicee.models.base_model.BaseKGE
    A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
    name = 'Shallom'
    shallom
    get_embeddings () → Tuple[numpy.ndarray, None]
    forward_k_vs_all (x) → torch.FloatTensor
    forward_triples (x) → torch.FloatTensor

    Parameters
    x

    Returns

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

    Parameters
    x

```


dicee.models.static_funcs

Functions

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

Perform quaternion multiplication

Module Contents

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

dicee.models.transformers

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

Classes

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

Module Contents

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)

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

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

Note

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

Variables

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

name = 'Byte'

config

temperature = 0.5

topk = 2

transformer

lm_head

loss_function (*yhat_batch, y_batch*)

Parameters

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

forward (*x: torch.LongTensor*)

Parameters

x (*B by T tensor*)

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

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

training_step (*batch, batch_idx=None*)

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

Parameters

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

Returns

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

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

Example:

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

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

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

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

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

Note

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

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

Bases: `torch.nn.Module`

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

weight

bias

forward (*input*)

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

`c_attn`

`c_proj`

`attn_dropout`

`resid_dropout`

`n_head`

`n_embd`

`dropout`

`flash = True`

`forward(x)`

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

c_fc

gelu

c_proj

dropout

forward (*x*)

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

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)

(continued from previous page)

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

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

Note

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

Variables

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

ln_1

attn

ln_2

mlp

forward (*x*)

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

```
    block_size: int = 1024
```

```
    vocab_size: int = 50304
```

```
    n_layer: int = 12
```

```
    n_head: int = 12
```

```
    n_embd: int = 768
```

```
    dropout: float = 0.0
```

```
    bias: bool = False
```

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

```
    Bases: torch.nn.Module
```

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
```

(continues on next page)

(continued from previous page)

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

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

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

Note

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

Variables

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

config

transformer

lm_head

get_num_params (*non_embedding=True*)

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

forward (*idx, targets=None*)

crop_block_size (*block_size*)

classmethod from_pretrained (*model_type, override_args=None*)

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

estimate_mfu (*fwdbwd_per_iter, dt*)

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

Classes

<i>ADOPT</i>	Base class for all optimizers.
<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

continues on next page

Table 1 – continued from previous page

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

Functions

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

Package Contents

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

Bases: torch.optim.optimizer.Optimizer

Base class for all optimizers.

Warning

Parameters need to be specified as collections that have a deterministic ordering that is consistent between runs. Examples of objects that don't satisfy those properties are sets and iterators over values of dictionaries.

Parameters

- **params** (*iterable*) – an iterable of `torch.Tensor`s or `dict`s. Specifies what Tensors should be optimized.
- **defaults** – (dict): a dict containing default values of optimization options (used when a parameter group doesn't specify them).

clip_lambda

__setstate__ (*state*)

step (*closure=None*)

Perform a single optimization step.

Parameters

closure (*Callable*, *optional*) – A closure that reevaluates the model and returns the loss.

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

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

training_step_outputs = []

mem_of_model() → Dict

Size of model in MB and number of params

training_step (*batch, batch_idx=None*)

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

Parameters

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

Returns

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

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

Example:

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

Note

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

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

Parameters

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

on_train_epoch_end (*args, **kwargs)

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

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

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

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

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

test_epoch_end (outputs: *List[Any]*)

test_dataloader() → `None`

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

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

Note

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

Note

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

`val_dataloader()` → None

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

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

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

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

Note

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

Note

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

`predict_dataloader()` → None

An iterable or collection of iterables specifying prediction samples.

For more information about multiple dataloaders, see this section.

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

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

Note

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

Returns

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

`train_dataloader()` → None

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

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

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning

do not assign state in `prepare_data`

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

Note

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

configure_optimizers (*parameters=None*)

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

Returns

Any of these 6 options.

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

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

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

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

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

Note

Some things to know:

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

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

args

embedding_dim = None

num_entities = None

num_relations = None

```

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

```

Parameters

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

forward_byte_pair_encoded_triple (x: *Tuple[torch.LongTensor, torch.LongTensor]*)

byte pair encoded neural link predictors

Parameters

```
init_params_with_sanity_checking()
```

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

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

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

Parameters

x

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

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

```
get_triple_representation(idx_hrt)
```

```
get_head_relation_representation(indexed_triple)
```

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

Parameters

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

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

Parameters

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

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

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

(continues on next page)

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

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

Note

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

Variables

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

args = `None`

__call__ (*x*)

static forward (*x*)

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

`args`

`embedding_dim` = None

`num_entities` = None

`num_relations` = None

`num_tokens` = None

`learning_rate` = None

`apply_unit_norm` = None

`input_dropout_rate` = None

`hidden_dropout_rate` = None

`optimizer_name` = None

`feature_map_dropout_rate` = None

`kernel_size` = None

`num_of_output_channels` = None

`weight_decay` = None

`loss`

`selected_optimizer` = None

`normalizer_class` = None

`normalize_head_entity_embeddings`

`normalize_relation_embeddings`

`normalize_tail_entity_embeddings`

`hidden_normalizer`

`param_init`

`input_dp_ent_real`

`input_dp_rel_real`

`hidden_dropout`

`loss_history` = []

`byte_pair_encoding`

`max_length_subword_tokens`

`block_size`

```

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

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

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 them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

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

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

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

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

Note

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

Variables

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

args

embedding_dim = None

num_entities = None

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

```

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

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

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

    Parameters
    -----

init_params_with_sanity_checking()

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

    Parameters
        •  $\mathbf{x}$ 
        •  $\mathbf{y\_idx}$ 
        • ordered_bpe_entities

```

```

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

    Parameters
    x

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

get_sentence_representation (x: torch.LongTensor)

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

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

    Parameters
    x (B × 2 × T)

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

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

    Parameters
    x

```



```

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

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

    name = 'AConEx'

    conv2d

    fc_num_input

    fc1

    norm_fc1

    bn_conv2d

    feature_map_dropout

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

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

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

    Parameters
    x

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

```

```

class dicee.models.ComplEx (args)
    Bases: dicee.models.base_model.BaseKGE
    Base class for all neural network modules.
    Your models should also subclass this class.

```

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

```

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

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

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

```

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

Note

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

Variables

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

name = 'Complex'

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

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

Parameters

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

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

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

`dicee.models.quaternion_mul(*, Q_1, Q_2)`
→ *Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]*

Perform quaternion multiplication :param Q_1: :param Q_2: :return:

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

`loss`

`selected_optimizer = None`

`normalizer_class = None`

`normalize_head_entity_embeddings`

`normalize_relation_embeddings`

`normalize_tail_entity_embeddings`

`hidden_normalizer`

`param_init`

`input_dp_ent_real`

`input_dp_rel_real`

```

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

    Parameters
        x (B x 2 x T)

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

    Parameters
    -----

init_params_with_sanity_checking()

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

    Parameters

    • x

    • y_idx

    • ordered_bpe_entities

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

    Parameters
        x

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

get_sentence_representation (x: torch.LongTensor)

    Parameters

    • (b (x shape)

    • 3

    • t)

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

    Parameters
        x (B x 2 x T)

```

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

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

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

`args = None`

`__call__(x)`

`static forward(x)`

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

`class dicee.models.QMult (args)`

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

(continues on next page)

```

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

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

```

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

Note

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

Variables

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

name = 'QMult'

explicit = True

quaternion_multiplication_followed_by_inner_product (*h, r, t*)

Parameters

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

Returns

Triple scores.

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

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

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

Parameters

x – The vector.

Returns

The normalized vector.

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

k_vs_all_score (*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*)

Parameters

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

forward_k_vs_all (*x*)

Parameters

x

forward_k_vs_sample (*x*, *target_entity_idx*)

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

class dicee.models.ConvQ(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings

name = 'ConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution (*Q_1*, *Q_2*)

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

Parameters

x

forward_k_vs_all (*x*: torch.Tensor)

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

class dicee.models.AConvQ(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Quaternion Knowledge Graph Embeddings

```

name = 'AConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution(Q_1, Q_2)

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

```

Parameters

x

forward_k_vs_all (*x: torch.Tensor*)

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

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

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

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

```

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

Note

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

Variables

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

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

`loss`

`selected_optimizer = None`

`normalizer_class = None`

`normalize_head_entity_embeddings`

`normalize_relation_embeddings`

`normalize_tail_entity_embeddings`

`hidden_normalizer`

`param_init`

`input_dp_ent_real`

`input_dp_rel_real`

`hidden_dropout`

```

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

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

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

    Parameters
    -----

init_params_with_sanity_checking ()

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

    Parameters

        •  $\mathbf{x}$ 

        •  $\mathbf{y\_idx}$ 

        • 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

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

        • 3

        •  $\mathbf{t}$ )

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

    Parameters

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

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

```

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

Note

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

Variables

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

args = None

__call__ (*x*)

static forward (*x*)

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

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

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

Note

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

Variables

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

name = 'OMult'

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

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

k_vs_all_score (*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

forward_k_vs_all (*x*)

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

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

Bases: `dicdee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

Note

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

Variables

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

name = 'ConvO'

conv2d

fc_num_input

fc1

bn_conv2d

norm_fc1

feature_map_dropout

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

residual_convolution (*O_1, O_2*)

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

Parameters

x

forward_k_vs_all (*x: torch.Tensor*)

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

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

Bases: `dicee.models.base_model.BaseKGE`

Additive Convolutional Octonion Knowledge Graph Embeddings

name = 'AConvO'

conv2d

fc_num_input

```

fc1

bn_conv2d

norm_fc1

feature_map_dropout

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

residual_convolution(O_1, O_2)

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

```

Parameters

x

forward_k_vs_all (*x: torch.Tensor*)

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

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

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

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

```

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

Note

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

Variables

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

```

name = 'Keci'

p

q

r

requires_grad_for_interactions = True

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

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

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

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

    e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

compute_sigma_qq(hq, rq)
    Compute  $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k$   $\sigma_{qq}$ 
    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.CKeci (args)
```

Bases: *Keci*

Without learning dimension scaling

```
name = 'CKeci'
```

```
requires_grad_for_interactions = False
```

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)

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

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

Note

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

Variables

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

name = '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

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

For different base vector interactions, we have

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

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

Kvsall training

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

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

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

Multiplying a base vector with its scalar coefficient

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

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

Parameter

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

returns

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

compute_sigma_pp (*hp, rp*)

Compute .. math:

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

compute_sigma_qq(hq, rq)

 Compute

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

compute_sigma_rr(hk, rk)

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

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

 Compute

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

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

for j in range(q):

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

print(sigma_pq.shape)

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

 Compute

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

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

```

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

```

$$\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$$

results = []
 sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

```

    for j in range(q):
        sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
    print(sigma_pq.shape)

```

class dicee.models.**BaseKGE** (args: dict)

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

args

embedding_dim = None

num_entities = None

```

num_relations = None

num_tokens = None

learning_rate = None

apply_unit_norm = None

input_dropout_rate = None

hidden_dropout_rate = None

optimizer_name = None

feature_map_dropout_rate = None

kernel_size = None

num_of_output_channels = None

weight_decay = None

loss

selected_optimizer = None

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

```

Parameters

$\mathbf{x} (B \times 2 \times T)$

forward_byte_pair_encoded_triple (x: *Tuple[torch.LongTensor, torch.LongTensor]*)

byte pair encoded neural link predictors

Parameters

```
init_params_with_sanity_checking()
```

```
forward(x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
        y_idx: torch.LongTensor = None)
```

Parameters

- **x**
- **y_idx**
- **ordered_bpe_entities**

```
forward_triples(x: torch.LongTensor) → torch.Tensor
```

Parameters

x

```
forward_k_vs_all(*args, **kwargs)
```

```
forward_k_vs_sample(*args, **kwargs)
```

```
get_triple_representation(idx_hrt)
```

```
get_head_relation_representation(indexed_triple)
```

```
get_sentence_representation(x: torch.LongTensor)
```

Parameters

- **(b (x shape)**
- **3**
- **t)**

```
get_bpe_head_and_relation_representation(x: torch.LongTensor)
→ Tuple[torch.FloatTensor, torch.FloatTensor]
```

Parameters

x ($B \times 2 \times T$)

```
get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]
```

```
class dicee.models.PykeenKGE(args: dict)
```

Bases: `dicee.models.base_model.BaseKGE`

A class for using knowledge graph embedding models implemented in Pykeen

Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Pykeen_HolE: Pykeen_HolE: Pykeen_TransD: Pykeen_TransE: Pykeen_TransF: Pykeen_TransH: Pykeen_TransR:

model_kwargs

name

model

loss_history = []

args

```

entity_embeddings = None

relation_embeddings = None

forward_k_vs_all (x: torch.LongTensor)
    # => Explicit version by this we can apply bn and dropout

    # (1) Retrieve embeddings of heads and relations + apply Dropout & Normalization if given. h, r =
    self.get_head_relation_representation(x) # (2) Reshape (1). if self.last_dim > 0:
        h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim,
        self.last_dim)

    # (3) Reshape all entities. if self.last_dim > 0:
        t = self.entity_embeddings.weight.reshape(self.num_entities, self.embedding_dim, self.last_dim)

    else:
        t = self.entity_embeddings.weight

    # (4) Call the score_t from interactions to generate triple scores. return self.interaction.score_t(h=h, r=r,
    all_entities=t, slice_size=1)

forward_triples (x: torch.LongTensor) → torch.FloatTensor
    # => Explicit version by this we can apply bn and dropout

    # (1) Retrieve embeddings of heads, relations and tails and apply Dropout & Normalization if given. h, r, t =
    self.get_triple_representation(x) # (2) Reshape (1). if self.last_dim > 0:
        h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim,
        self.last_dim) t = t.reshape(len(x), self.embedding_dim, self.last_dim)

    # (3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice_size=None, slice_dim=0)

abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)

```

class dicee.models.BaseKGE (args: dict)

Bases: [BaseKGELightning](#)

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

```


Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training` (*bool*) – Boolean represents whether this module is in training or evaluation mode.

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

`loss`

`selected_optimizer = None`

`normalizer_class = None`

`normalize_head_entity_embeddings`

`normalize_relation_embeddings`

`normalize_tail_entity_embeddings`

`hidden_normalizer`

`param_init`

`input_dp_ent_real`

`input_dp_rel_real`

```

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

    Parameters
         $\mathbf{x}$  ( $B \times 2 \times T$ )

forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors

    Parameters
    -----

init_params_with_sanity_checking()

forward (x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
        y_idx: torch.LongTensor = None)

    Parameters

    •  $\mathbf{x}$ 

    •  $\mathbf{y\_idx}$ 

    • ordered_bpe_entities

forward_triples (x: torch.LongTensor) → torch.Tensor

    Parameters
         $\mathbf{x}$ 

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: $f(x) = \sum_{i=0}^{d-1} a_i x^i$ and use the three different scoring function as in the paper to evaluate the score. We also consider combining with Neural Networks.

name = 'LFMult'

entity_embeddings

relation_embeddings

degree

m

x_values

forward_triples (*idx_triple*)

Parameters

x

construct_multi_coeff (*x*)

poly_NN (*x, coefh, coefr, coeft*)

Constructing a 2 layers NN to represent the embeddings. $h = \sigma(wh^T x + bh)$, $r = \sigma(wr^T x + br)$, $t = \sigma(wt^T x + bt)$

linear (*x, w, b*)

scalar_batch_NN (*a, b, c*)

element wise multiplication between a,b and c: Inputs : a, b, c ==> torch.tensor of size batch_size x m x d
Output : a tensor of size batch_size x d

tri_score (*coeff_h, coeff_r, coeff_t*)

this part implement the trilinear scoring techniques:

$score(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \frac{a_i b_j c_k}{1+(i+j+k)d}$

1. generate the range for i,j and k from [0 d-1]
2. perform $\frac{a_i b_j c_k}{1+(i+j+k)d}$ in parallel for every batch
3. take the sum over each batch

vtp_score (*h, r, t*)

this part implement the vector triple product scoring techniques:

$score(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \frac{a_i c_j b_k - b_i c_j a_k}{(1+(i+j)d)(1+k)}$

1. generate the range for i,j and k from [0 d-1]
2. Compute the first and second terms of the sum
3. Multiply with then denominator and take the sum
4. take the sum over each batch

comp_func (*h, r, t*)

this part implement the function composition scoring techniques: i.e. score = <hor, t>

polynomial (*coeff*, *x*, *degree*)

This function takes a matrix tensor of coefficients (*coeff*), a tensor vector of points *x* and range of integer $[0, 1, \dots, d]$ and return a vector tensor ($\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d$,

$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d$)

pop (*coeff*, *x*, *degree*)

This function allow us to evaluate the composition of two polynomes without for loops :) it takes a matrix tensor of coefficients (*coeff*), a matrix tensor of points *x* and range of integer $[0, 1, \dots, d]$

and return a tensor ($\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d$,

$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d$)

class `dicee.models.DualE` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Dual Quaternion Knowledge Graph Embeddings (<https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657>)

name = 'DualE'

entity_embeddings

relation_embeddings

num_ent = None

kvsall_score (*e_1_h*, *e_2_h*, *e_3_h*, *e_4_h*, *e_5_h*, *e_6_h*, *e_7_h*, *e_8_h*, *e_1_t*, *e_2_t*, *e_3_t*, *e_4_t*, *e_5_t*, *e_6_t*, *e_7_t*, *e_8_t*, *r_1*, *r_2*, *r_3*, *r_4*, *r_5*, *r_6*, *r_7*, *r_8*) → torch.tensor

KvsAll scoring function

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_triples (*idx_triple*: torch.tensor) → torch.tensor

Negative Sampling forward pass:

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

forward_k_vs_all (*x*)

KvsAll forward pass

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

$\mathbf{T}(x: \text{torch.tensor}) \rightarrow \text{torch.tensor}$

Transpose function

Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)

dicee.query_generator

Classes

QueryGenerator

Module Contents

```
class dicee.query_generator.QueryGenerator(train_path: str, val_path: str, test_path: str,  
      ent2id: Dict = None, rel2id: Dict = None, seed: int = 1, gen_valid: bool = False,  
      gen_test: bool = True)
```

```
    train_path
```

```
    val_path
```

```
    test_path
```

```
    gen_valid = False
```

```
    gen_test = True
```

```
    seed = 1
```

```
    max_ans_num = 1000000.0
```

```
    mode
```

```
    ent2id = None
```

```
    rel2id: Dict = None
```

```
    ent_in: Dict
```

```
    ent_out: Dict
```

```
    query_name_to_struct
```

```
    list2tuple(list_data)
```

```
    tuple2list(x: List | Tuple) → List | Tuple
```

```
        Convert a nested tuple to a nested list.
```

set_global_seed (*seed: int*)
Set seed

construct_graph (*paths: List[str]*) → Tuple[Dict, Dict]
Construct graph from triples Returns dicts with incoming and outgoing edges

fill_query (*query_structure: List[str | List], ent_in: Dict, ent_out: Dict, answer: int*) → bool
Private method for fill_query logic.

achieve_answer (*query: List[str | List], ent_in: Dict, ent_out: Dict*) → set
Private method for achieve_answer logic. @TODO: Document the code

write_links (*ent_out, small_ent_out*)

ground_queries (*query_structure: List[str | List], ent_in: Dict, ent_out: Dict, small_ent_in: Dict, small_ent_out: Dict, gen_num: int, query_name: str*)
Generating queries and achieving answers

unmap (*query_type, queries, tp_answers, fp_answers, fn_answers*)

unmap_query (*query_structure, query, id2ent, id2rel*)

generate_queries (*query_struct: List, gen_num: int, query_type: str*)
Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting queries and answers in return @ TODO: create a class for each single query struct

save_queries (*query_type: str, gen_num: int, save_path: str*)

abstract load_queries (*path*)

get_queries (*query_type: str, gen_num: int*)

static save_queries_and_answers (*path: str, data: List[Tuple[str, Tuple[collections.defaultdict]]]*)
→ None
Save Queries into Disk

static load_queries_and_answers (*path: str*) → List[Tuple[str, Tuple[collections.defaultdict]]]
Load Queries from Disk to Memory

dicee.read_preprocess_save_load_kg

Submodules

dicee.read_preprocess_save_load_kg.preprocess

Classes

PreprocessKG

Preprocess the data in memory

Module Contents

class dicee.read_preprocess_save_load_kg.preprocess.**PreprocessKG** (*kg*)
Preprocess the data in memory

kg

start () → None
Preprocess train, valid and test datasets stored in knowledge graph instance

Parameter

rtype
None

preprocess_with_byte_pair_encoding()

preprocess_with_byte_pair_encoding_with_padding() → None

preprocess_with_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_serve

```
$ docker pull qdrant/qdrant && docker run -p 6333:6333 -p 6334:6334 -v $(pwd)/qdrant_storage:/qdrant/storage:z
qdrant/qdrant $ dicee_vector_db -index -serve -path CountryEmbeddings -collection "countries_vdb"
```

Attributes

`app`

`neural_searcher`

Classes

`NeuralSearcher`

`StringListRequest`

!!! abstract "Usage Documentation"

Functions

`get_default_arguments()`

`index(args)`

`root()`

`search_embeddings(q)`

`retrieve_embeddings(q)`

`search_embeddings_batch(request)`

`serve(args)`

`main()`

Module Contents

```
dicee.scripts.index_serve.get_default_arguments()

dicee.scripts.index_serve.index(args)

dicee.scripts.index_serve.app

dicee.scripts.index_serve.neural_searcher = None

class dicee.scripts.index_serve.NeuralSearcher(args)

    collection_name

    entity_to_idx = None

    qdrant_client

    topk = 5

    retrieve_embedding(entity: str = None, entities: List[str] = None) → List

    search(entity: str)

async dicee.scripts.index_serve.root()

async dicee.scripts.index_serve.search_embeddings(q: str)

async dicee.scripts.index_serve.retrieve_embeddings(q: str)

class dicee.scripts.index_serve.StringListRequest(/, **data: Any)
    Bases: pydantic.BaseModel

    !!! abstract "Usage Documentation"
        [Models](../concepts/models.md)

    A base class for creating Pydantic models.

    __class_vars__
        The names of the class variables defined on the model.

    __private_attributes__
        Metadata about the private attributes of the model.

    __signature__
        The synthesized __init__ [Signature][inspect.Signature] of the model.

    __pydantic_complete__
        Whether model building is completed, or if there are still undefined fields.

    __pydantic_core_schema__
        The core schema of the model.

    __pydantic_custom_init__
        Whether the model has a custom __init__ function.

    __pydantic_decorators__
        Metadata containing the decorators defined on the model. This replaces Model.__validators__ and
        Model.__root_validators__ from Pydantic V1.
```

`__pydantic_generic_metadata__`

Metadata for generic models; contains data used for a similar purpose to `__args__`, `__origin__`, `__parameters__` in typing-module generics. May eventually be replaced by these.

`__pydantic_parent_namespace__`

Parent namespace of the model, used for automatic rebuilding of models.

`__pydantic_post_init__`

The name of the post-init method for the model, if defined.

`__pydantic_root_model__`

Whether the model is a `[RootModel][pydantic.root_model.RootModel]`.

`__pydantic_serializer__`

The *pydantic-core* `SchemaSerializer` used to dump instances of the model.

`__pydantic_validator__`

The *pydantic-core* `SchemaValidator` used to validate instances of the model.

`__pydantic_fields__`

A dictionary of field names and their corresponding `[FieldInfo][pydantic.fields.FieldInfo]` objects.

`__pydantic_computed_fields__`

A dictionary of computed field names and their corresponding `[ComputedFieldInfo][pydantic.fields.ComputedFieldInfo]` objects.

`__pydantic_extra__`

A dictionary containing extra values, if `[extra][pydantic.config.ConfigDict.extra]` is set to `'allow'`.

`__pydantic_fields_set__`

The names of fields explicitly set during instantiation.

`__pydantic_private__`

Values of private attributes set on the model instance.

`queries: List[str]`

`reducer: str | None = None`

`async dicee.scripts.index_serve.search_embeddings_batch` (*request: [StringListRequest](#)*)

`dicee.scripts.index_serve.serve` (*args*)

`dicee.scripts.index_serve.main` ()

`dicee.scripts.run`

Functions

`get_default_arguments` ([*description*])
`main` ()

Extends `pytorch_lightning` Trainer's arguments with ours

Module Contents

`dicee.scripts.run.get_default_arguments` (*description=None*)

Extends pytorch_lightning Trainer's arguments with ours

`dicee.scripts.run.main()`

`dicee.static_funcs`

Functions

<code>create_recipriocal_triples(x)</code>	Add inverse triples into dask 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>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>	
<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>	

continues on next page

Table 2 – continued from previous page

<code>deploy_head_entity_prediction(pre_trained_kge,</code> <code>...)</code>	
<code>deploy_relation_prediction(pre_trained_kge,</code> <code>...)</code>	
<code>vocab_to_parquet(vocab_to_idx, name, ...)</code>	
<code>create_experiment_folder([folder_name])</code>	
<code>continual_training_setup_executor(→ None)</code>	
<code>exponential_function(→ torch.FloatTensor)</code>	
<code>load_numpy(→ numpy.ndarray)</code>	
<code>evaluate(entity_to_idx, scores, easy_answers, # @TODO: CD: Renamed this function</code> <code>hard_answers)</code>	
<code>download_file(url[, destination_folder])</code>	
<code>download_files_from_url(→ None)</code>	
<code>download_pretrained_model(→ str)</code>	
<code>write_csv_from_model_parallel(path)</code>	Create
<code>from_pretrained_model_write_embeddings_int,</code> <code>None)</code>	

Module Contents

`dicee.static_funcs.create_recipriocal_triples(x)`

Add inverse triples into dask dataframe :param x: :return:

`dicee.static_funcs.get_er_vocab(data, file_path: str = None)`

`dicee.static_funcs.get_re_vocab(data, file_path: str = None)`

`dicee.static_funcs.get_ee_vocab(data, file_path: str = None)`

`dicee.static_funcs.timeit(func)`

`dicee.static_funcs.save_pickle(*, data: object = None, file_path=str)`

`dicee.static_funcs.load_pickle(file_path=str)`

`dicee.static_funcs.load_term_mapping(file_path=str)`

`dicee.static_funcs.select_model(args: dict, is_continual_training: bool = None,`
`storage_path: str = None)`

`dicee.static_funcs.load_model(path_of_experiment_folder: str, model_name='model.pt', verbose=0)`
`→ Tuple[object, Tuple[dict, dict]]`

Load weights and initialize pytorch module from namespace arguments

`dicee.static_funcs.load_model_ensemble(path_of_experiment_folder: str)`
`→ Tuple[dicee.models.base_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]`

Construct Ensemble Of weights and initialize pytorch module from namespace arguments

- (1) Detect models under given path
- (2) Accumulate parameters of detected models
- (3) Normalize parameters
- (4) Insert (3) into model.

```
dicee.static_funcs.save_numpy_ndarray (*, data: numpy.ndarray, file_path: str)

dicee.static_funcs.numpy_data_type_changer (train_set: numpy.ndarray, num: int)
    → numpy.ndarray
    Detect most efficient data type for a given triples :param train_set: :param num: :return:

dicee.static_funcs.save_checkpoint_model (model, path: str) → None
    Store Pytorch model into disk

dicee.static_funcs.store (trained_model, model_name: str = 'model', full_storage_path: str = None,
    save_embeddings_as_csv=False) → None

dicee.static_funcs.add_noisy_triples (train_set: pandas.DataFrame, add_noise_rate: float)
    → pandas.DataFrame
    Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

dicee.static_funcs.read_or_load_kg (args, cls)

dicee.static_funcs.intialize_model (args: dict, verbose=0) → Tuple[object, str]

dicee.static_funcs.load_json (p: str) → dict

dicee.static_funcs.save_embeddings (embeddings: numpy.ndarray, indexes, path: str) → None
    Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:

dicee.static_funcs.random_prediction (pre_trained_kge)

dicee.static_funcs.deploy_triple_prediction (pre_trained_kge, str_subject, str_predicate,
    str_object)

dicee.static_funcs.deploy_tail_entity_prediction (pre_trained_kge, str_subject, str_predicate,
    top_k)

dicee.static_funcs.deploy_head_entity_prediction (pre_trained_kge, str_object, str_predicate,
    top_k)

dicee.static_funcs.deploy_relation_prediction (pre_trained_kge, str_subject, str_object, top_k)

dicee.static_funcs.vocab_to_parquet (vocab_to_idx, name, path_for_serialization, print_into)

dicee.static_funcs.create_experiment_folder (folder_name='Experiments')

dicee.static_funcs.continual_training_setup_executor (executor) → None

dicee.static_funcs.exponential_function (x: numpy.ndarray, lam: float, ascending_order=True)
    → torch.FloatTensor

dicee.static_funcs.load_numpy (path) → numpy.ndarray

dicee.static_funcs.evaluate (entity_to_idx, scores, easy_answers, hard_answers)
    # @TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

dicee.static_funcs.download_file (url, destination_folder='.')
```

`dicee.static_funcs.download_files_from_url (base_url: str, destination_folder='.') → None`

Parameters

- **base_url** (e.g. `"https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll"`)
- **destination_folder** (e.g. `"KINSHIP-Keci-dim128-epoch256-KvsAll"`)

`dicee.static_funcs.download_pretrained_model (url: str) → str`

`dicee.static_funcs.write_csv_from_model_parallel (path: str)`

Create

`dicee.static_funcs.from_pretrained_model_write_embeddings_into_csv (path: str) → None`

`dicee.static_funcs_training`

Functions

```
make_iterable_verbose(→ Iterable)
```

```
evaluate_lp([model, triple_idx, num_entities, ...])
```

```
evaluate_bpe_lp(model, triple_idx, ..., info)
```

```
efficient_zero_grad(model)
```

Module Contents

`dicee.static_funcs_training.make_iterable_verbose (iterable_object, verbose, desc='Default', position=None, leave=True) → Iterable`

`dicee.static_funcs_training.evaluate_lp (model=None, triple_idx=None, num_entities=None, er_vocab: Dict[Tuple, List] = None, re_vocab: Dict[Tuple, List] = None, info='Eval Starts', batch_size=128, chunk_size=1000)`

`dicee.static_funcs_training.evaluate_bpe_lp (model, triple_idx: List[Tuple], all_bpe_shaped_entities, er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List], info='Eval Starts')`

`dicee.static_funcs_training.efficient_zero_grad (model)`

`dicee.static_preprocess_funcs`

Attributes

```
enable_log
```

Functions

<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>DICE_Trainer</code>	DICE_Trainer implement
---------------------------	------------------------

Functions

```
load_term_mapping([file_path])
```

```
initialize_trainer(...)
```

```
get_callbacks(args)
```

Module Contents

```
dicee.trainer.dice_trainer.load_term_mapping(file_path=str)
```

```
dicee.trainer.dice_trainer.initialize_trainer(args, callbacks)  
→ dicee.trainer.torch_trainer.TorchTrainer | dicee.trainer.model_parallelism.TensorParallel | dicee.trainer.torch_trainer_ddp
```

```
dicee.trainer.dice_trainer.get_callbacks(args)
```

```
class dicee.trainer.dice_trainer.DICE_Trainer(args, is_continual_training: bool, storage_path,  
evaluator=None)
```

DICE_Trainer implement

- 1- Pytorch Lightning trainer (<https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html>)
- 2- Multi-GPU Trainer(<https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>)
- 3- CPU Trainer

args

is_continual_training:bool

storage_path:str

evaluator:

report:dict

report

args

trainer = None

is_continual_training

storage_path

evaluator = None

form_of_labelling = None

continual_start(knowledge_graph)

- (1) Initialize training.
- (2) Load model
- (3) Load trainer (3) Fit model

Parameter

returns

- *model*
- **form_of_labelling** (*str*)

initialize_trainer (*callbacks: List*)

→ `lightning.Trainer` | `diccee.trainer.model_parallelism.TensorParallel` | `diccee.trainer.torch_trainer.TorchTrainer` | `diccee.t`

Initialize Trainer from input arguments

initialize_or_load_model ()

init_dataloader (*dataset: torch.utils.data.Dataset*) → `torch.utils.data.DataLoader`

init_dataset () → `torch.utils.data.Dataset`

start (*knowledge_graph: diccee.knowledge_graph.KG* | *numpy.memmap*)

→ `Tuple[diccee.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[diccee.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

diccee.trainer.model_parallelism

Classes

TensorParallel

Abstract class for Trainer class for knowledge graph embedding models

Functions

```
extract_input_outputs(z[, device])

find_good_batch_size(train_loader,
tp_ensemble_model)

forward_backward_update_loss(→ float)
```

Module Contents

```
dicee.trainer.model_parallelism.extract_input_outputs (z: list, device=None)

dicee.trainer.model_parallelism.find_good_batch_size (train_loader, tp_ensemble_model)

dicee.trainer.model_parallelism.forward_backward_update_loss (z: Tuple, ensemble_model)
→ float
```

```
class dicee.trainer.model_parallelism.TensorParallel (args, callbacks)
```

Bases: *dicee.abstracts.AbstractTrainer*

Abstract class for Trainer class for knowledge graph embedding models

Parameter

args

[str] ?

callbacks: list

?

fit (*args, **kwargs)

Train model

`dicee.trainer.torch_trainer`

Classes

<i>TorchTrainer</i>	TorchTrainer for using single GPU or multi CPUs on a single node
---------------------	--

Module Contents

```
class dicee.trainer.torch_trainer.TorchTrainer (args, callbacks)
```

Bases: *dicee.abstracts.AbstractTrainer*

TorchTrainer for using single GPU or multi CPUs on a single node

Arguments

callbacks: list of Abstract callback instances

loss_function = None

```

optimizer = None

model = None

train_dataloaders = None

training_step = None

process

fit(*args, train_dataloaders, **kwargs) → None
    Training starts
    Arguments

    kwargs: Tuple
        empty dictionary

    Return type
        batch loss (float)

forward_backward_update(x_batch: torch.Tensor, y_batch: torch.Tensor) → torch.Tensor
    Compute forward, loss, backward, and parameter update
    Arguments

    Return type
        batch loss (float)

extract_input_outputs_set_device(batch: list) → Tuple
    Construct inputs and outputs from a batch of inputs with outputs From a batch of inputs and put
    Arguments

    Return type
        (tuple) mini-batch on select device

```

dicee.trainer.torch_trainer_ddp

Classes

<i>TorchDDPTrainer</i>	A Trainer based on torch.nn.parallel.DistributedDataParallel
<i>NodeTrainer</i>	

Functions

<i>make_iterable_verbose</i> (→ 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_idxes

mapping.

relation_idxes

mapping.

form

?

store

?

label_smoothing_rate

Using hard targets (0,1) drives weights to infinity. An outlier produces enormous gradients.

Return type

`torch.utils.data.Dataset`

fit (*args, **kwargs)

Train model

class `dicee.trainer.torch_trainer_ddp.NodeTrainer(trainer, model: torch.nn.Module, train_dataset_loader: torch.utils.data.DataLoader, callbacks, num_epochs: int)`

trainer

local_rank

global_rank

optimizer

train_dataset_loader

loss_func

callbacks

model

num_epochs

loss_history = []

ctx

scaler

extract_input_outputs (*z: list*)

train()

Training loop for DDP

Classes

DICE_Trainer

DICE_Trainer implement

Package Contents

class dicee.trainer.DICE_Trainer(*args, is_continual_training: bool, storage_path, evaluator=None*)

DICE_Trainer implement

- 1- Pytorch Lightning trainer (<https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html>)
- 2- Multi-GPU Trainer(<https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>)
- 3- CPU Trainer

args

is_continual_training:bool

storage_path:str

evaluator:

report:dict

report

args

trainer = None

is_continual_training

storage_path

evaluator = None

form_of_labelling = None

continual_start (*knowledge_graph*)

- (1) Initialize training.
- (2) Load model
- (3) Load trainer (3) Fit model

Parameter

returns

- *model*
- **form_of_labelling** (*str*)

```

initialize_trainer (callbacks: List)
    → lightning.Trainer | dicee.trainer.model_parallelism.TensorParallel | dicee.trainer.torch_trainer.TorchTrainer | dicee.
    Initialize Trainer from input arguments

initialize_or_load_model ()

init_dataloader (dataset: torch.utils.data.Dataset) → torch.utils.data.DataLoader

init_dataset () → torch.utils.data.Dataset

start (knowledge_graph: dicee.knowledge_graph.KG | numpy.memmap)
    → Tuple[dicee.models.base_model.BaseKGE, str]
    Start the training
    (1) Initialize Trainer
    (2) Initialize or load a pretrained KGE model
    in DDP setup, we need to load the memory map of already read/index KG.

k_fold_cross_validation (dataset) → Tuple[dicee.models.base_model.BaseKGE, str]
    Perform K-fold Cross-Validation
    1. Obtain K train and test splits.
    2. For each split,
        2.1 initialize trainer and model
        2.2. Train model with configuration provided in args.
        2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.
    3. Report the mean and average MRR .

Parameters
    • self
    • dataset

Returns
    model

```

14.2 Attributes

```
__version__
```

14.3 Classes

<i>Pyke</i>	A Physical Embedding Model for Knowledge Graphs
<i>DistMult</i>	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
<i>CKeci</i>	Without learning dimension scaling
<i>Keci</i>	Base class for all neural network modules.
<i>TransE</i>	Translating Embeddings for Modeling
<i>DeCaL</i>	Base class for all neural network modules.

continues on next page

Table 3 – continued from previous page

<i>DualE</i>	Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)
<i>ComplEx</i>	Base class for all neural network modules.
<i>AConEx</i>	Additive Convolutional ComplEx Knowledge Graph Embeddings
<i>AConvO</i>	Additive Convolutional Octonion Knowledge Graph Embeddings
<i>AConvQ</i>	Additive Convolutional Quaternion Knowledge Graph Embeddings
<i>ConvQ</i>	Convolutional Quaternion Knowledge Graph Embeddings
<i>ConvO</i>	Base class for all neural network modules.
<i>ConEx</i>	Convolutional ComplEx Knowledge Graph Embeddings
<i>QMult</i>	Base class for all neural network modules.
<i>OMult</i>	Base class for all neural network modules.
<i>Shallom</i>	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
<i>LFMult</i>	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
<i>PykeenKGE</i>	A class for using knowledge graph embedding models implemented in Pykeen
<i>Byte</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>EnsembleKGE</i>	
<i>DICE_Trainer</i>	DICE_Trainer implement
<i>KGE</i>	Knowledge Graph Embedding Class for interactive usage of pre-trained models
<i>BPE_NegativeSamplingDataset</i>	An abstract class representing a Dataset.
<i>MultiLabelDataset</i>	An abstract class representing a Dataset.
<i>MultiClassClassificationDataset</i>	Dataset for the 1vsALL training strategy
<i>OnevsAllDataset</i>	Dataset for the 1vsALL training strategy
<i>KvsAll</i>	Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.
<i>AllvsAll</i>	Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.
<i>OnevsSample</i>	A custom PyTorch Dataset class for knowledge graph embeddings, which includes
<i>KvsSampleDataset</i>	KvsSample a Dataset:
<i>NegSampleDataset</i>	An abstract class representing a Dataset.
<i>TriplePredictionDataset</i>	Triple Dataset
<i>CVDDataModule</i>	Create a Dataset for cross validation
<i>QueryGenerator</i>	

14.4 Functions

<i>create_recipriocal_triples(x)</i>	Add inverse triples into dask dataframe
--------------------------------------	---

continues on next page

Table 4 – 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>	
<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 4 – continued from previous page

<code>load_numpy(→ numpy.ndarray)</code>	
<code>evaluate(entity_to_idx, scores, easy_answers, # @TODO: CD: Renamed this function hard_answers)</code>	
<code>download_file(url[, destination_folder])</code>	
<code>download_files_from_url(→ None)</code>	
<code>download_pretrained_model(→ str)</code>	
<code>write_csv_from_model_parallel(path)</code>	Create
<code>from_pretrained_model_write_embeddings_into None)</code>	
<code>mapping_from_first_two_cols_to_third(train_se</code>	
<code>timeit(func)</code>	
<code>load_term_mapping([file_path])</code>	
<code>reload_dataset(path, form_of_labelling, ...)</code>	Reload the files from disk to construct the Pytorch dataset
<code>construct_dataset(→ torch.utils.data.Dataset)</code>	

14.5 Package Contents

class `dicee.Pyke` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

A Physical Embedding Model for Knowledge Graphs

name = 'Pyke'

dist_func

margin = 1.0

forward_triples (*x: torch.LongTensor*)

Parameters

x

class `dicee.DistMult` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Embedding Entities and Relations for Learning and Inference in Knowledge Bases <https://arxiv.org/abs/1412.6575>

name = 'DistMult'

k_vs_all_score (*emb_h: torch.FloatTensor, emb_r: torch.FloatTensor, emb_E: torch.FloatTensor*)

Parameters

- **emb_h**
- **emb_r**
- **emb_E**

```
forward_k_vs_all (x: torch.LongTensor)
```

```
forward_k_vs_sample (x: torch.LongTensor, target_entity_idx: torch.LongTensor)
```

```
score (h, r, t)
```

```
class dicee.CKeci (args)
```

Bases: *Keci*

Without learning dimension scaling

```
name = 'CKeci'
```

```
requires_grad_for_interactions = False
```

```
class dicee.Keci (args)
```

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training (bool)` – Boolean represents whether this module is in training or evaluation mode.

```
name = 'Keci'
```

```
p
```

```
q
```

```
r
```

requires_grad_for_interactions = True

compute_sigma_pp (*hp, rp*)

Compute $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_{ir_k} - h_{kr_i}) e_i e_k$

σ_{pp} captures the interactions between along p bases For instance, let $p = e_1, e_2, e_3$, we compute interactions between $e_1 e_2, e_1 e_3$, and $e_2 e_3$ This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

for k in range(i + 1, p):

 results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., $e_1 e_1, e_1 e_2, e_1 e_3$,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

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 $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_{jr_k} - h_{kr_j}) e_j e_k \sigma_{qq}$ captures the interactions between along q bases For instance, let $q = e_1, e_2, e_3$, we compute interactions between $e_1 e_2, e_1 e_3$, and $e_2 e_3$ This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

for k in range(j + 1, q):

 results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])

sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., $e_1 e_1, e_1 e_2, e_1 e_3$,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals: $e_1 e_2, e_1 e_3, e_2 e_3$.

compute_sigma_pq (**, hp, hq, rp, rq*)

$\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_{ir_j} - h_{jr_i}) e_i e_j$

results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):

for j in range(q):

 sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]

print(sigma_pq.shape)

apply_coefficients (*hp, hq, rp, rq*)

Multiplying a base vector with its scalar coefficient

clifford_multiplication (*h0, hp, hq, r0, rp, rq*)

Compute our CL multiplication

$h = h_0 + \sum_{i=1}^p h_i e_i + \sum_{j=p+1}^{p+q} h_j e_j$
 $r = r_0 + \sum_{i=1}^p r_i e_i + \sum_{j=p+1}^{p+q} r_j e_j$

$e_i^2 = +1$ for $i \leq p$ $e_j^2 = -1$ for $p < j \leq p+q$ $e_i e_j = -e_j e_i$ for i

e_j

$h r = \sigma_0 + \sigma_p + \sigma_q + \sigma_{pp} + \sigma_q + \sigma_{pq}$ where

- (1) $\sigma_0 = h_0 r_0 + \sum_{i=1}^p (h_0 r_i) e_i - \sum_{j=p+1}^{p+q} (h_j r_j) e_j$
- (2) $\sigma_p = \sum_{i=1}^p (h_0 r_i + h_i r_0) e_i$
- (3) $\sigma_q = \sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j$
- (4) $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k$
- (5) $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k$
- (6) $\sigma_{pq} = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$

construct_cl_multivector (*x: torch.FloatTensor, r: int, p: int, q: int*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors $Cl_{\{p,q\}}(\mathbb{R}^d)$

Parameter

x: torch.FloatTensor with (n,d) shape

returns

- **a0** (*torch.FloatTensor with (n,r) shape*)
- **ap** (*torch.FloatTensor with (n,r,p) shape*)
- **aq** (*torch.FloatTensor with (n,r,q) shape*)

forward_k_vs_with_explicit (*x: torch.Tensor*)

k_vs_all_score (*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations \mathbb{R}^d .
- (2) Construct head entity and relation embeddings according to $Cl_{\{p,q\}}(\mathbb{R}^d)$.
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this functions are identical Parameter ——— x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n, **IEI**) shape

construct_batch_selected_cl_multivector (*x: torch.FloatTensor, r: int, p: int, q: int*)
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of batchs multivectors $Cl_{\{p,q\}}(\mathbb{R}^d)$

Parameter

x: torch.FloatTensor with (n,k, d) shape

returns

- **a0** (*torch.FloatTensor with (n,k, m) shape*)
- **ap** (*torch.FloatTensor with (n,k, m, p) shape*)
- **aq** (*torch.FloatTensor with (n,k, m, q) shape*)

forward_k_vs_sample (*x: torch.LongTensor, target_entity_idx: torch.LongTensor*) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,2) shape

target_entity_idx: torch.LongTensor with (n, k) shape k denotes the selected number of examples.

rtype

torch.FloatTensor with (n, k) shape

score (*h, r, t*)

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,3) shape

rtype

torch.FloatTensor with (n) shape

class `dicee.TransE` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Translating Embeddings for Modeling Multi-relational Data <https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf>

name = 'TransE'

margin = 4

score (*head_ent_emb, rel_ent_emb, tail_ent_emb*)

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

class `dicee.DeCaL` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

name = 'DeCaL'

entity_embeddings

relation_embeddings

p

q

r

re

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,) shape

rtype

torch.FloatTensor with (n) shape

cl_pqr (*a: torch.tensor*) → torch.tensor

Input: tensor(batch_size, emb_dim) → output: tensor with 1+p+q+r components with size (batch_size, emb_dim/(1+p+q+r)) each.

1) takes a tensor of size (batch_size, emb_dim), split it into 1 + p + q + r components, hence 1+p+q+r must be a divisor of the emb_dim. 2) Return a list of the 1+p+q+r components vectors, each are tensors of size (batch_size, emb_dim/(1+p+q+r))

compute_sigmas_single (*list_h_emb, list_r_emb, list_t_emb*)

here we compute all the sums with no others vectors interaction taken with the scalar product with t, that is,

$$s0 = h_0 r_0 t_0 s1 = \sum_{i=1}^p h_i r_i t_0 s2 = \sum_{j=p+1}^{p+q} h_j r_j t_0 s3 = \sum_{i=1}^q (h_0 r_i t_i + h_i r_0 t_i) s4 = \sum_{i=p+1}^{p+q} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+1}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i)$$

and return:

$$sigma_0 t = \sigma_0 \cdot t_0 = s0 + s1 - s2 s3, s4 \text{ and } s5$$

compute_sigmas_multivect (*list_h_emb, list_r_emb*)

Here we compute and return all the sums with vectors interaction for the same and different bases.

For same bases vectors interaction we have

$$\sigma_p p = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (h_i r_{i'} - h_{i'} r_i) (model \text{ the interactions between } e_i \text{ and } e_{i'} \text{ for } 1 \leq i, i' \leq p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{j'} r_j)$$

For different base vector interactions, we have

$$\sigma_{pq} = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) (\text{interactions between } e_i \text{ and } e_j \text{ for } 1 \leq i \leq p \text{ and } p+1 \leq j \leq p+q) \sigma_{pr} = \sum_{i=1}^p$$

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:

$$\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{j=i+1}^p (x_{iy_{i'}} - x_{i'} y_{i'})$$

σ_{pp} captures the interactions between along p bases For instance, let $p \in \{e_1, e_2, e_3\}$, we compute interactions between $e_1 e_2, e_1 e_3$, and $e_2 e_3$ This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

for k in range(i + 1, p):

 results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])

sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p, e.g., $e_1 e_1, e_1 e_2, e_1 e_3,$

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

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

$$\sigma_{q,q}^* = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (x_j y_{j'} - x_{j'} y_j) E_{q,16}$$

$\sigma_{q,q}$ 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 = None
    kvsall_score(e_1_h, e_2_h, e_3_h, e_4_h, e_5_h, e_6_h, e_7_h, e_8_h, e_1_t, e_2_t, e_3_t, e_4_t, e_5_t, e_6_t, e_7_t, e_8_t, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8) → torch.tensor
        KvsAll scoring function

    Input
    x: torch.LongTensor with (n, ) shape

    Output
    torch.FloatTensor with (n) shape
    forward_triples(idx_triple: torch.tensor) → torch.tensor
        Negative Sampling forward pass:

    Input
    x: torch.LongTensor with (n, ) shape

    Output
    torch.FloatTensor with (n) shape
    forward_k_vs_all(x)
        KvsAll forward pass

    Input
    x: torch.LongTensor with (n, ) shape

    Output
    torch.FloatTensor with (n) shape
    T(x: torch.tensor) → torch.tensor
        Transpose function
        Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)

```

```
class dicee.ComplEx(args)
```

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training (bool)` – Boolean represents whether this module is in training or evaluation mode.

```
name = 'ComplEx'
```

```
static score(head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor,
             tail_ent_emb: torch.FloatTensor)
```

```
static k_vs_all_score(emb_h: torch.FloatTensor, emb_r: torch.FloatTensor,
                     emb_E: torch.FloatTensor)
```

Parameters

- `emb_h`
- `emb_r`
- `emb_E`

```
forward_k_vs_all(x: torch.LongTensor) -> torch.FloatTensor
```

```
forward_k_vs_sample(x: torch.LongTensor, target_entity_idx: torch.LongTensor)
```

```
class dicee.AConEx(args)
```

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional ComplEx Knowledge Graph Embeddings

```

name = 'AConEx'

conv2d

fc_num_input

fc1

norm_fc1

bn_conv2d

feature_map_dropout

residual_convolution (C_1: Tuple[torch.Tensor, torch.Tensor],
                      C_2: Tuple[torch.Tensor, torch.Tensor]) → torch.FloatTensor
    Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors
    that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds
    complex-valued embeddings :return:

forward_k_vs_all (x: torch.Tensor) → torch.FloatTensor

forward_triples (x: torch.Tensor) → torch.FloatTensor

    Parameters
    x

forward_k_vs_sample (x: torch.Tensor, target_entity_idx: torch.Tensor)

class dicee.AConvO (args: dict)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional Octonion Knowledge Graph Embeddings
    name = 'AConvO'

    conv2d

    fc_num_input

    fc1

    bn_conv2d

    norm_fc1

    feature_map_dropout

    static octonion_normalizer (emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4,
                               emb_rel_e5, emb_rel_e6, emb_rel_e7)

    residual_convolution (O_1, O_2)

    forward_triples (x: torch.Tensor) → torch.Tensor

    Parameters
    x

```

forward_k_vs_all (*x*: *torch.Tensor*)

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **|Entities|**) Given a batch of head entities and relations => shape (size of batch,|Entities|)

class *dicee.AConvQ* (*args*)

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Quaternion Knowledge Graph Embeddings

name = 'AConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

feature_map_dropout

residual_convolution (*Q_1*, *Q_2*)

forward_triples (*indexed_triple*: *torch.Tensor*) → *torch.Tensor*

Parameters

x

forward_k_vs_all (*x*: *torch.Tensor*)

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **|Entities|**) Given a batch of head entities and relations => shape (size of batch,|Entities|)

class *dicee.ConvQ* (*args*)

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings

name = 'ConvQ'

entity_embeddings

relation_embeddings

conv2d

fc_num_input

fc1

bn_conv1

bn_conv2

`feature_map_dropout`

`residual_convolution(Q_1, Q_2)`

`forward_triples(indexed_triple: torch.Tensor) → torch.Tensor`

Parameters

x

`forward_k_vs_all(x: torch.Tensor)`

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **Entities**) Given a batch of head entities and relations => shape (size of batch,|Entities|)

`class dicee.ConvO(args: dict)`

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training (bool)` – Boolean represents whether this module is in training or evaluation mode.

`name = 'ConvO'`

`conv2d`

`fc_num_input`

`fc1`

```

bn_conv2d

norm_fc1

feature_map_dropout

static octonion_normalizer(emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4,
                           emb_rel_e5, emb_rel_e6, emb_rel_e7)

residual_convolution(O_1, O_2)

forward_triples(x: torch.Tensor) → torch.Tensor

    Parameters
    x

forward_k_vs_all(x: torch.Tensor)
    Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] =>
    [0.0,0.1,...,0.8], shape=> (1, Entities) Given a batch of head entities and relations => shape (size of batch,|
    Entities)

class dicee.ConEx(args)
    Bases: dicee.models.base_model.BaseKGE
    Convolutional ComplEx Knowledge Graph Embeddings
    name = 'ConEx'

    conv2d

    fc_num_input

    fc1

    norm_fc1

    bn_conv2d

    feature_map_dropout

    residual_convolution(C_1: Tuple[torch.Tensor, torch.Tensor],
                        C_2: Tuple[torch.Tensor, torch.Tensor]) → torch.FloatTensor
        Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors
        that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds
        complex-valued embeddings :return:

    forward_k_vs_all(x: torch.Tensor) → torch.FloatTensor

    forward_triples(x: torch.Tensor) → torch.FloatTensor

    Parameters
    x

    forward_k_vs_sample(x: torch.Tensor, target_entity_idx: torch.Tensor)

class dicee.QMult(args)
    Bases: dicee.models.base_model.BaseKGE
    Base class for all neural network modules.
    Your models should also subclass this class.

```

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

name = 'QMult'

explicit = True

quaternion_multiplication_followed_by_inner_product (*h, r, t*)

Parameters

- **h** – shape: (**batch_dims*, dim) The head representations.
- **r** – shape: (**batch_dims*, dim) The head representations.
- **t** – shape: (**batch_dims*, dim) The tail representations.

Returns

Triple scores.

static quaternion_normalizer (*x: torch.FloatTensor*) → torch.FloatTensor

Normalize the length of relation vectors, if the forward constraint has not been applied yet.

Absolute value of a quaternion

$$|a + bi + cj + dk| = \sqrt{a^2 + b^2 + c^2 + d^2}$$

L2 norm of quaternion vector:

$$\|x\|^2 = \sum_{i=1}^d |x_i|^2 = \sum_{i=1}^d (x_i.re^2 + x_i.im_1^2 + x_i.im_2^2 + x_i.im_3^2)$$

Parameters

x – The vector.

Returns

The normalized vector.

score (*head_ent_emb*: torch.FloatTensor, *rel_ent_emb*: torch.FloatTensor,
tail_ent_emb: torch.FloatTensor)

k_vs_all_score (*bpe_head_ent_emb*, *bpe_rel_ent_emb*, *E*)

Parameters

- **bpe_head_ent_emb**
- **bpe_rel_ent_emb**
- **E**

forward_k_vs_all (*x*)

Parameters

x

forward_k_vs_sample (*x*, *target_entity_idx*)

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples,i.e., [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **Entities**) Given a batch of head entities and relations => shape (size of batch,| Entities|)

class dicee.OMult (*args*)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.


```

name = 'OMult'

static octonion_normalizer(emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4,
                           emb_rel_e5, emb_rel_e6, emb_rel_e7)

score(head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor,
      tail_ent_emb: torch.FloatTensor)

k_vs_all_score(bpe_head_ent_emb, bpe_rel_ent_emb, E)

forward_k_vs_all(x)
    Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples,i.e.,
    [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, Entities) Given a batch of head entities and
    relations => shape (size of batch,| Entities|)

class dicee.Shallom(args)
    Bases: dicee.models.base_model.BaseKGE
    A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
    name = 'Shallom'
    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, coefr*)

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: Pykeen_HolE: Pykeen_HolE: Pykeen_TransD: Pykeen_TransE: Pykeen_TransF: Pykeen_TransH: Pykeen_TransR:

model_kwargs

```

name

model

loss_history = []

args

entity_embeddings = None

relation_embeddings = None

forward_k_vs_all (x: torch.LongTensor)
    # => Explicit version by this we can apply bn and dropout

    # (1) Retrieve embeddings of heads and relations + apply Dropout & Normalization if given. h, r =
    self.get_head_relation_representation(x) # (2) Reshape (1). if self.last_dim > 0:
        h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim,
        self.last_dim)

    # (3) Reshape all entities. if self.last_dim > 0:
        t = self.entity_embeddings.weight.reshape(self.num_entities, self.embedding_dim, self.last_dim)

    else:
        t = self.entity_embeddings.weight

    # (4) Call the score_t from interactions to generate triple scores. return self.interaction.score_t(h=h, r=r,
    all_entities=t, slice_size=1)

forward_triples (x: torch.LongTensor) → torch.FloatTensor
    # => Explicit version by this we can apply bn and dropout

    # (1) Retrieve embeddings of heads, relations and tails and apply Dropout & Normalization if given. h, r, t =
    self.get_triple_representation(x) # (2) Reshape (1). if self.last_dim > 0:
        h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim,
        self.last_dim) t = t.reshape(len(x), self.embedding_dim, self.last_dim)

    # (3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice_size=None, slice_dim=0)

abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)

```

```

class dicee.BytE (*args, **kwargs)
    Bases: dicee.models.base_model.BaseKGE

```

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()

```

(continues on next page)

(continued from previous page)

```
self.conv1 = nn.Conv2d(1, 20, 5)
self.conv2 = nn.Conv2d(20, 20, 5)

def forward(self, x):
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

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: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing them to be nested in a tree structure. You can assign the sub-modules as regular attributes:

```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will also have their parameters converted when you call `to()`, etc.

Note

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

`training (bool)` – Boolean represents whether this module is in training or evaluation mode.

`args`

`embedding_dim = None`

`num_entities = None`

`num_relations = None`

`num_tokens = None`

`learning_rate = None`

`apply_unit_norm = None`

`input_dropout_rate = None`

`hidden_dropout_rate = None`

`optimizer_name = None`

`feature_map_dropout_rate = None`

`kernel_size = None`

`num_of_output_channels = None`

`weight_decay = None`

`loss`

`selected_optimizer = None`

```

normalizer_class = None

normalize_head_entity_embeddings

normalize_relation_embeddings

normalize_tail_entity_embeddings

hidden_normalizer

param_init

input_dp_ent_real

input_dp_rel_real

hidden_dropout

loss_history = []

byte_pair_encoding

max_length_subword_tokens

block_size

forward_byte_pair_encoded_k_vs_all (x: torch.LongTensor)

    Parameters
         $\mathbf{x}$  ( $B \times 2 \times T$ )

forward_byte_pair_encoded_triple (x: Tuple[torch.LongTensor, torch.LongTensor])
    byte pair encoded neural link predictors

    Parameters
    -----

init_params_with_sanity_checking()

forward (x: torch.LongTensor | Tuple[torch.LongTensor, torch.LongTensor],
        y_idx: torch.LongTensor = None)

    Parameters

        •  $\mathbf{x}$ 

        •  $\mathbf{y\_idx}$ 

        • ordered_bpe_entities

forward_triples (x: torch.LongTensor) → torch.Tensor

    Parameters
         $\mathbf{x}$ 

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

```

```

get_sentence_representation(x: torch.LongTensor)

    Parameters
        • (b (x shape)
        • 3
        • t)

get_bpe_head_and_relation_representation(x: torch.LongTensor)
    → Tuple[torch.FloatTensor, torch.FloatTensor]

    Parameters
        x (B x 2 x T)

get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]

class dicee.EnsembleKGE(seed_model=None, pretrained_models: List = None)

    name

    train_mode = True

    named_children()

    property example_input_array

    parameters()

    modules()

    __iter__()

    __len__()

    eval()

    to(device)

    mem_of_model()

    __call__(x_batch)

    step()

    get_embeddings()

    __str__()

dicee.create_recipriocal_triples(x)
    Add inverse triples into dask dataframe :param x: :return:

dicee.get_er_vocab(data, file_path: str = None)

dicee.get_re_vocab(data, file_path: str = None)

dicee.get_ee_vocab(data, file_path: str = None)

dicee.timeit(func)

dicee.save_pickle(*, data: object = None, file_path=str)

```



```

dicee.load_pickle (file_path=str)

dicee.load_term_mapping (file_path=str)

dicee.select_model (args: dict, is_continual_training: bool = None, storage_path: str = None)

dicee.load_model (path_of_experiment_folder: str, model_name='model.pt', verbose=0)
    → Tuple[object, Tuple[dict, dict]]
    Load weights and initialize pytorch module from namespace arguments

dicee.load_model_ensemble (path_of_experiment_folder: str)
    → Tuple[dicee.models.base_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]
    Construct Ensemble Of weights and initialize pytorch module from namespace arguments

    (1) Detect models under given path
    (2) Accumulate parameters of detected models
    (3) Normalize parameters
    (4) Insert (3) into model.

dicee.save_numpy_ndarray (*, data: numpy.ndarray, file_path: str)

dicee.numpy_data_type_changer (train_set: numpy.ndarray, num: int) → numpy.ndarray
    Detect most efficient data type for a given triples :param train_set: :param num: :return:

dicee.save_checkpoint_model (model, path: str) → None
    Store Pytorch model into disk

dicee.store (trained_model, model_name: str = 'model', full_storage_path: str = None,
    save_embeddings_as_csv=False) → None

dicee.add_noisy_triples (train_set: pandas.DataFrame, add_noise_rate: float) → pandas.DataFrame
    Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

dicee.read_or_load_kg (args, cls)

dicee.intialize_model (args: dict, verbose=0) → Tuple[object, str]

dicee.load_json (p: str) → dict

dicee.save_embeddings (embeddings: numpy.ndarray, indexes, path: str) → None
    Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:

dicee.random_prediction (pre_trained_kge)

dicee.deploy_triple_prediction (pre_trained_kge, str_subject, str_predicate, str_object)

dicee.deploy_tail_entity_prediction (pre_trained_kge, str_subject, str_predicate, top_k)

dicee.deploy_head_entity_prediction (pre_trained_kge, str_object, str_predicate, top_k)

dicee.deploy_relation_prediction (pre_trained_kge, str_subject, str_object, top_k)

dicee.vocab_to_parquet (vocab_to_idx, name, path_for_serialization, print_into)

dicee.create_experiment_folder (folder_name='Experiments')

dicee.continual_training_setup_executor (executor) → None

```

`dicee.exponential_function(x: numpy.ndarray, lam: float, ascending_order=True) → torch.FloatTensor`

`dicee.load_numpy(path) → numpy.ndarray`

`dicee.evaluate(entity_to_idx, scores, easy_answers, hard_answers)`

@TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

`dicee.download_file(url, destination_folder='.')`

`dicee.download_files_from_url(base_url: str, destination_folder='.') → None`

Parameters

- **base_url** (e.g. [“https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll”](https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll))
- **destination_folder** (e.g. `"KINSHIP-Keci-dim128-epoch256-KvsAll"`)

`dicee.download_pretrained_model(url: str) → str`

`dicee.write_csv_from_model_parallel(path: str)`

Create

`dicee.from_pretrained_model_write_embeddings_into_csv(path: str) → None`

class `dicee.DICE_Trainer(args, is_continual_training: bool, storage_path, evaluator=None)`

DICE_Trainer implement

- 1- Pytorch Lightning trainer (<https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html>)
- 2- Multi-GPU Trainer(<https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>)
- 3- CPU Trainer

args

is_continual_training:bool

storage_path:str

evaluator:

report:dict

report

args

trainer = None

is_continual_training

storage_path

evaluator = None

form_of_labelling = None

continual_start (*knowledge_graph*)

- (1) Initialize training.
- (2) Load model
- (3) Load trainer (3) Fit model

Parameter

returns

- *model*
- **form_of_labelling** (*str*)

initialize_trainer (*callbacks: List*)

→ `lightning.Trainer` | `dicee.trainer.model_parallelism.TensorParallel` | `dicee.trainer.torch_trainer.TorchTrainer` | `dicee.`

Initialize Trainer from input arguments

initialize_or_load_model ()

init_dataloader (*dataset: torch.utils.data.Dataset*) → `torch.utils.data.DataLoader`

init_dataset () → `torch.utils.data.Dataset`

start (*knowledge_graph: dicee.knowledge_graph.KG* | *numpy.memmap*)

→ `Tuple[dicee.models.base_model.BaseKGE, str]`

Start the training

- (1) Initialize Trainer
- (2) Initialize or load a pretrained KGE model

in DDP setup, we need to load the memory map of already read/index KG.

k_fold_cross_validation (*dataset*) → `Tuple[dicee.models.base_model.BaseKGE, str]`

Perform K-fold Cross-Validation

1. Obtain K train and test splits.
2. **For each split**,
 - 2.1 initialize trainer and model
 - 2.2. Train model with configuration provided in args.
 - 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.
3. Report the mean and average MRR .

Parameters

- **self**
- **dataset**

Returns

model

class `dicee.KGE` (*path=None, url=None, construct_ensemble=False, model_name=None*)

Bases: `dicee.abstracts.BaseInteractiveKGE`, `dicee.abstracts.InteractiveQueryDecomposition`

Knowledge Graph Embedding Class for interactive usage of pre-trained models

__str__ ()

to (*device: str*) → `None`

get_transductive_entity_embeddings (*indices: torch.LongTensor* | *List[str]*, *as_pytorch=False*, *as_numpy=False*, *as_list=True*) → `torch.FloatTensor` | `numpy.ndarray` | `List[float]`

create_vector_database (*collection_name: str, distance: str, location: str = 'localhost',
port: int = 6333*)

generate (*h="", r=""*)

eval_lp_performance (*dataset=List[Tuple[str, str, str]], filtered=True*)

predict_missing_head_entity (*relation: List[str] | str, tail_entity: List[str] | str, within=None,
batch_size=2, topk=1, return_indices=False*) → Tuple

Given a relation and a tail entity, return top k ranked head entity.

$\text{argmax}_{\{e \in E\}} f(e, r, t)$, where $r \in R, t \in E$.

Parameter

relation: Union[List[str], str]

String representation of selected relations.

tail_entity: Union[List[str], str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_relations (*head_entity: List[str] | str, tail_entity: List[str] | str, within=None,
batch_size=2, topk=1, return_indices=False*) → Tuple

Given a head entity and a tail entity, return top k ranked relations.

$\text{argmax}_{\{r \in R\}} f(h, r, t)$, where $h, t \in E$.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_tail_entity (*head_entity: List[str] | str, relation: List[str] | str,
within: List[str] = None, batch_size=2, topk=1, return_indices=False*) → torch.FloatTensor

Given a head entity and a relation, return top k ranked entities

$\text{argmax}_{\{e \in E\}} f(h, r, e)$, where $h \in E$ and $r \in R$.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

scores

predict (*, h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, within=None, logits=True) → torch.FloatTensor

Parameters

- **logits**
- **h**
- **r**
- **t**
- **within**

predict_topk (*, h: str | List[str] = None, r: str | List[str] = None, t: str | List[str] = None, topk: int = 10, within: List[str] = None, batch_size: int = 1024)

Predict missing item in a given triple.

Returns

- If you query a single (h, r, ?) or (?, r, t) or (h, ?, t), returns List[(item, score)]
- If you query a batch of B, returns List of B such lists.

triple_score (h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, logits=False) → torch.FloatTensor

Predict triple score

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

Returns: Tuple

pytorch tensor of triple score

return_multi_hop_query_results (*aggregated_query_for_all_entities*, *k*: int, *only_scores*)

single_hop_query_answering (*query*: tuple, *only_scores*: bool = True, *k*: int = None)

answer_multi_hop_query (*query_type*: str = None, *query*: Tuple[str | Tuple[str, str], Ellipsis] = None, *queries*: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, *tnorm*: str = 'prod', *neg_norm*: str = 'standard', *lambda_*: float = 0.0, *k*: int = 10, *only_scores*=False) → List[Tuple[str, torch.Tensor]]

@TODO: Refactoring is needed # @TODO: Score computation for each query type should be done in a static function

Find an answer set for EPFO queries including negation and disjunction

Parameter

query_type: str The type of the query, e.g., “2p”.

query: Union[str, Tuple[str, Tuple[str, str]]] The query itself, either a string or a nested tuple.

queries: List of Tuple[Union[str, Tuple[str, str]], ...]

tnorm: str The t-norm operator.

neg_norm: str The negation norm.

lambda_: float lambda parameter for sugeno and yager negation norms

k: int The top-k substitutions for intermediate variables.

returns

- List[Tuple[str, torch.Tensor]]
- Entities and corresponding scores sorted in the descening order of scores

find_missing_triples (*confidence*: float, *entities*: List[str] = None, *relations*: List[str] = None, *topk*: int = 10, *at_most*: int = sys.maxsize) → Set

Find missing triples

Iterative over a set of entities E and a set of relation R :

orall e in E and orall r in R f(e,r,x)

Return (e,r,x)

otin G and f(e,r,x) > confidence

confidence: float

A threshold for an output of a sigmoid function given a triple.

topk: int

Highest ranked k item to select triples with f(e,r,x) > confidence .

at_most: int

Stop after finding at_most missing triples

{(e,r,x) | 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.

train_literals (*train_file_path: str = None, num_epochs: int = 100, lit_lr: float = 0.001, eval_litreal_preds: bool = True, eval_file_path: str = None, lit_normalization_type: str = 'z-norm', batch_size: int = 1024, sampling_ratio: float = None, random_seed=1, loader_backend: str = 'pandas', freeze_entity_embeddings: bool = True, gate_residual: bool = True, device: str = None*)

Trains the Literal Embeddings model using literal data.

Parameters

- **train_file_path** (*str*) – Path to the training data file.
- **num_epochs** (*int*) – Number of training epochs.
- **lit_lr** (*float*) – Learning rate for the literal model.
- **eval_litreal_preds** (*bool*) – If True, evaluate the model after training.
- **eval_file_path** (*str*) – Path to evaluation data file.
- **norm_type** (*str*) – Normalization type to use ('z-norm', 'min-max', or None).
- **batch_size** (*int*) – Batch size for training.
- **sampling_ratio** (*float*) – Ratio of training triples to use.
- **loader_backend** (*str*) – Backend for loading the dataset ('pandas' or 'rdflib').
- **freeze_entity_embeddings** (*bool*) – If True, freeze the entity embeddings during training.
- **gate_residual** (*bool*) – If True, use gate residual connections in the model.
- **device** (*str*) – Device to use for training ('cuda' or 'cpu'). If None, will use available GPU or CPU.

predict_literals (*entity: List[str] | str = None, attribute: List[str] | str = None, denormalize_preds: bool = True*) → numpy.ndarray

Predicts literal values for given entities and attributes.

Parameters

- **entity** (*Union[List[str], str]*) – Entity or list of entities to predict literals for.
- **attribute** (*Union[List[str], str]*) – Attribute or list of attributes to predict literals for.
- **denormalize_preds** (*bool*) – If True, denormalizes the predictions.

Returns

Predictions for the given entities and attributes.

Return type

numpy ndarray

```
evaluate_literal_prediction (eval_file_path: str = None, store_lit_preds: bool = True,
                             eval_literals: bool = True, loader_backend: str = 'pandas',
                             return_attr_error_metrics: bool = False)
```

Evaluates the trained literal prediction model on a test file.

Parameters

- **eval_file_path** (str) – Path to the evaluation file.
- **store_lit_preds** (bool) – If True, stores the predictions in a CSV file.
- **eval_literals** (bool) – If True, evaluates the literal predictions and prints error metrics.
- **loader_backend** (str) – Backend for loading the dataset ('pandas' or 'rdflib').

Returns

None

```
dicee.mapping_from_first_two_cols_to_third(train_set_idx)
```

```
dicee.timeit(func)
```

```
dicee.load_term_mapping(file_path=str)
```

```
dicee.reload_dataset(path: str, form_of_labelling, scoring_technique, neg_ratio, label_smoothing_rate)
```

Reload the files from disk to construct the Pytorch dataset

```
dicee.construct_dataset(*, train_set: numpy.ndarray | list, valid_set=None, test_set=None,
                        ordered_bpe_entities=None, train_target_indices=None, target_dim: int = None, entity_to_idx: dict,
                        relation_to_idx: dict, form_of_labelling: str, scoring_technique: str, neg_ratio: int,
                        label_smoothing_rate: float, byte_pair_encoding=None, block_size: int = None)
                        → torch.utils.data.Dataset
```

```
class dicee.BPE_NegativeSamplingDataset(train_set: torch.LongTensor,
                                         ordered_shaped_bpe_entities: torch.LongTensor, neg_ratio: int)
```

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

`DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

train_set

ordered_bpe_entities

num_bpe_entities

neg_ratio

num_datapoints


```

__len__()

__getitem__(idx)

collate_fn(batch_shaped_bpe_triples: List[Tuple[torch.Tensor, torch.Tensor]])

```

```

class dicee.MultiLabelDataset (train_set: torch.LongTensor, train_indices_target: torch.LongTensor,
                               target_dim: int, torch_ordered_shaped_bpe_entities: torch.LongTensor)

```

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

`DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

```

train_set

train_indices_target

target_dim

num_datapoints

torch_ordered_shaped_bpe_entities

collate_fn = None

__len__()

__getitem__(idx)

```

```

class dicee.MultiClassClassificationDataset (subword_units: numpy.ndarray, block_size: int = 8)

```

Bases: torch.utils.data.Dataset

Dataset for the 1vsALL training strategy

Parameters

- **train_set_idx** – Indexed triples for the training.
- **entity_idx**s – mapping.
- **relation_idx**s – mapping.
- **form** – ?
- **num_workers** – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

Return type

torch.utils.data.Dataset

```

train_data

block_size = 8

num_of_data_points

collate_fn = None

__len__()

__getitem__(idx)

```

```
class dicee.OnevsAllDataset (train_set_idx: numpy.ndarray, entity_idxes)
```

Bases: torch.utils.data.Dataset

Dataset for the 1vsALL training strategy

Parameters

- **train_set_idx** – Indexed triples for the training.
- **entity_idxes** – mapping.
- **relation_idxes** – mapping.
- **form** – ?
- **num_workers** – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

Return type

torch.utils.data.Dataset

```

train_data

target_dim

collate_fn = None

__len__()

__getitem__(idx)

```

```
class dicee.KvsAll (train_set_idx: numpy.ndarray, entity_idxes, relation_idxes, form, store=None,
                    label_smoothing_rate: float = 0.0)
```

Bases: torch.utils.data.Dataset

Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.

Let D denote a dataset for KvsAll training and be defined as $D := \{(x, y)_i\}_{i=1}^N$, 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 $[0, 1]^{|E|}$ **[EI]** 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_idxxs

[dictionary] string representation of an entity to its integer id

relation_idxxs

[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_idxxs, relation_idxxs, label_smoothing_rate=0.0)

Bases: torch.utils.data.Dataset

Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.

Let D denote a dataset for AllvsAll training and be defined as $D := \{(x, y)_i\}_i^N$, where $x: (h, r)$ is a possible unique tuple of an entity h in E and a relation r in R . Hence $N = |E| \times |R|$ y : denotes a multi-label vector in $[0, 1]^{|E|}$ is a binary label.

orall $y_i = 1$ s.t. $(h, r) \in KG$

Note

AllvsAll extends KvsAll via none existing (h,r). Hence, it adds data points that are labelled without 1s, only with 0s.

train_set_idx

[numpy.ndarray] n by 3 array representing n triples

entity_idxxs

[dictionary] string representation of an entity to its integer id

relation_idxxs

[dictionary] string representation of a relation to its integer id

self : torch.utils.data.Dataset

```
>>> a = AllvsAll()
>>> a
? array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

train_data = None

```

train_target = None

label_smoothing_rate

collate_fn = None

target_dim

__len__()

__getitem__(idx)

```

```

class dicee.OnevsSample(train_set: numpy.ndarray, num_entities, num_relations,
                        neg_sample_ratio: int = None, label_smoothing_rate: float = 0.0)

```

Bases: torch.utils.data.Dataset

A custom PyTorch Dataset class for knowledge graph embeddings, which includes both positive and negative sampling for a given dataset for multi-class classification problem..

Parameters

- **train_set** (*np.ndarray*) – A numpy array containing triples of knowledge graph data. Each triple consists of (head_entity, relation, tail_entity).
- **num_entities** (*int*) – The number of unique entities in the knowledge graph.
- **num_relations** (*int*) – The number of unique relations in the knowledge graph.
- **neg_sample_ratio** (*int, optional*) – The number of negative samples to be generated per positive sample. Must be a positive integer and less than num_entities.
- **label_smoothing_rate** (*float, optional*) – A label smoothing rate to apply to the positive and negative labels. Defaults to 0.0.

train_data

The input data converted into a PyTorch tensor.

Type

torch.Tensor

num_entities

Number of entities in the dataset.

Type

int

num_relations

Number of relations in the dataset.

Type

int

neg_sample_ratio

Ratio of negative samples to be drawn for each positive sample.

Type

int

label_smoothing_rate

The smoothing factor applied to the labels.

Type

torch.Tensor

collate_fn

A function that can be used to collate data samples into batches (set to None by default).

Type

function, optional

train_data

num_entities

num_relations

neg_sample_ratio = None

label_smoothing_rate

collate_fn = None

__len__()

Returns the number of samples in the dataset.

__getitem__(idx)

Retrieves a single data sample from the dataset at the given index.

Parameters

idx (*int*) – The index of the sample to retrieve.

Returns

A tuple consisting of:

- **x** (torch.Tensor): The head and relation part of the triple.
- **y_idx** (torch.Tensor): The concatenated indices of the true object (tail entity) and the indices of the negative samples.
- **y_vec** (torch.Tensor): A vector containing the labels for the positive and negative samples, with label smoothing applied.

Return type

tuple

```
class dicee.KvsSampleDataset (train_set_idx: numpy.ndarray, entity_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_idxs
    mapping.

form
    ?

store
    ?

label_smoothing_rate
    ?

```

```
torch.utils.data.Dataset
```

```

train_data = None

train_target = None

neg_ratio = None

num_entities

label_smoothing_rate

collate_fn = None

max_num_of_classes

__len__()

__getitem__(idx)

```

```

class dicee.NegSampleDataset(train_set: numpy.ndarray, num_entities: int, num_relations: int,
                             neg_sample_ratio: int = 1)

```

```
Bases: torch.utils.data.Dataset
```

An abstract class representing a Dataset.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note

`DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

```

neg_sample_ratio

train_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 = False

gen_test = True

seed = 1

max_ans_num = 1000000.0

mode

ent2id = None

rel2id: Dict = None

ent_in: Dict

ent_out: Dict

query_name_to_struct

list2tuple(list_data)

```

tuple2list (x: List | Tuple) → List | Tuple
    Convert a nested tuple to a nested list.

set_global_seed (seed: int)
    Set seed

construct_graph (paths: List[str]) → Tuple[Dict, Dict]
    Construct graph from triples Returns dicts with incoming and outgoing edges

fill_query (query_structure: List[str | List], ent_in: Dict, ent_out: Dict, answer: int) → bool
    Private method for fill_query logic.

achieve_answer (query: List[str | List], ent_in: Dict, ent_out: Dict) → set
    Private method for achieve_answer logic. @TODO: Document the code

write_links (ent_out, small_ent_out)

ground_queries (query_structure: List[str | List], ent_in: Dict, ent_out: Dict, small_ent_in: Dict,
    small_ent_out: Dict, gen_num: int, query_name: str)
    Generating queries and achieving answers

unmap (query_type, queries, tp_answers, fp_answers, fn_answers)

unmap_query (query_structure, query, id2ent, id2rel)

generate_queries (query_struct: List, gen_num: int, query_type: str)
    Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting
    queries and answers in return @ TODO: create a class for each single query struct

save_queries (query_type: str, gen_num: int, save_path: str)

abstract load_queries (path)

get_queries (query_type: str, gen_num: int)

static save_queries_and_answers (path: str, data: List[Tuple[str, Tuple[collections.defaultdict]]])
    → None
    Save Queries into Disk

static load_queries_and_answers (path: str) → List[Tuple[str, Tuple[collections.defaultdict]]]
    Load Queries from Disk to Memory

dicee.__version__ = '0.1.5'

```

Python Module Index

d

- `dicee`, 12
- `dicee.__main__`, 12
- `dicee.abstracts`, 12
- `dicee.analyse_experiments`, 18
- `dicee.callbacks`, 19
- `dicee.config`, 26
- `dicee.dataset_classes`, 29
- `dicee.eval_static_funcs`, 40
- `dicee.evaluator`, 42
- `dicee.executer`, 43
- `dicee.knowledge_graph`, 45
- `dicee.knowledge_graph_embeddings`, 46
- `dicee.literal_classes`, 51
- `dicee.models`, 55
 - `adopt`, 55
 - `base_model`, 56
 - `clifford`, 65
 - `complex`, 72
 - `dualE`, 74
 - `ensemble`, 76
 - `function_space`, 76
 - `octonion`, 80
 - `pykeen_models`, 83
 - `quaternion`, 84
 - `real`, 87
 - `static_funcs`, 89
 - `transformers`, 89
- `dicee.query_generator`, 143
- `dicee.read_preprocess_save_load_kg`, 144
- `dicee.read_preprocess_save_load_kg.preprocess`, 144
- `dicee.read_preprocess_save_load_kg.read_from_disk`, 145
- `dicee.read_preprocess_save_load_kg.save_load_disk`, 146
- `dicee.read_preprocess_save_load_kg.util`, 146
- `dicee.sanity_checkers`, 151
- `dicee.scripts`, 152
 - `index_serve`, 152
 - `run`, 154
- `dicee.static_funcs`, 155
- `dicee.static_funcs_training`, 158
- `dicee.static_preprocess_funcs`, 158
- `dicee.trainer`, 159
 - `dice_trainer`, 159
 - `model_parallelism`, 161
 - `torch_trainer`, 162
 - `torch_trainer_ddp`, 163

Index

Non-alphabetical

`__call__()` (*dicee.EnsembleKGE method*), 192
`__call__()` (*dicee.models.base_model.IdentityClass method*), 65
`__call__()` (*dicee.models.ensemble.EnsembleKGE method*), 76
`__call__()` (*dicee.models.IdentityClass method*), 106, 117, 123
`__class_vars__` (*dicee.scripts.index_serve.StringListRequest attribute*), 153
`__getitem__()` (*dicee.AllvsAll method*), 204
`__getitem__()` (*dicee.BPE_NegativeSamplingDataset method*), 201
`__getitem__()` (*dicee.dataset_classes.AllvsAll method*), 33
`__getitem__()` (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 30
`__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*), 37
`__getitem__()` (*dicee.KvsAll method*), 203
`__getitem__()` (*dicee.KvsSampleDataset method*), 206
`__getitem__()` (*dicee.literal_classes.LiteralDataset method*), 54
`__getitem__()` (*dicee.MultiClassClassificationDataset method*), 202
`__getitem__()` (*dicee.MultiLabelDataset method*), 201
`__getitem__()` (*dicee.NegSampleDataset method*), 207
`__getitem__()` (*dicee.OnevsAllDataset method*), 202
`__getitem__()` (*dicee.OnevsSample method*), 205
`__getitem__()` (*dicee.TriplePredictionDataset method*), 207
`__iter__()` (*dicee.config.Namespace method*), 28
`__iter__()` (*dicee.EnsembleKGE method*), 192
`__iter__()` (*dicee.knowledge_graph.KG method*), 46
`__iter__()` (*dicee.models.ensemble.EnsembleKGE method*), 76
`__len__()` (*dicee.AllvsAll method*), 204
`__len__()` (*dicee.BPE_NegativeSamplingDataset method*), 200
`__len__()` (*dicee.dataset_classes.AllvsAll method*), 33
`__len__()` (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 30
`__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*), 36
`__len__()` (*dicee.dataset_classes.OnevsAllDataset method*), 31
`__len__()` (*dicee.dataset_classes.OnevsSample method*), 34
`__len__()` (*dicee.dataset_classes.TriplePredictionDataset method*), 37
`__len__()` (*dicee.EnsembleKGE method*), 192
`__len__()` (*dicee.knowledge_graph.KG method*), 46
`__len__()` (*dicee.KvsAll method*), 203
`__len__()` (*dicee.KvsSampleDataset method*), 206
`__len__()` (*dicee.literal_classes.LiteralDataset method*), 54
`__len__()` (*dicee.models.ensemble.EnsembleKGE method*), 76
`__len__()` (*dicee.MultiClassClassificationDataset method*), 202
`__len__()` (*dicee.MultiLabelDataset method*), 201
`__len__()` (*dicee.NegSampleDataset method*), 206
`__len__()` (*dicee.OnevsAllDataset method*), 202
`__len__()` (*dicee.OnevsSample method*), 205
`__len__()` (*dicee.TriplePredictionDataset method*), 207
`__private_attributes__` (*dicee.scripts.index_serve.StringListRequest attribute*), 153
`__pydantic_complete__` (*dicee.scripts.index_serve.StringListRequest attribute*), 153
`__pydantic_computed_fields__` (*dicee.scripts.index_serve.StringListRequest attribute*), 154
`__pydantic_core_schema__` (*dicee.scripts.index_serve.StringListRequest attribute*), 153
`__pydantic_custom_init__` (*dicee.scripts.index_serve.StringListRequest attribute*), 153
`__pydantic_decorators__` (*dicee.scripts.index_serve.StringListRequest attribute*), 153
`__pydantic_extra__` (*dicee.scripts.index_serve.StringListRequest attribute*), 154
`__pydantic_fields__` (*dicee.scripts.index_serve.StringListRequest attribute*), 154
`__pydantic_fields_set__` (*dicee.scripts.index_serve.StringListRequest attribute*), 154
`__pydantic_generic_metadata__` (*dicee.scripts.index_serve.StringListRequest attribute*), 153

- `__pydantic_parent_namespace__` (*dicee.scripts.index_serve.StringListRequest attribute*), 154
- `__pydantic_post_init__` (*dicee.scripts.index_serve.StringListRequest attribute*), 154
- `__pydantic_private__` (*dicee.scripts.index_serve.StringListRequest attribute*), 154
- `__pydantic_root_model__` (*dicee.scripts.index_serve.StringListRequest attribute*), 154
- `__pydantic_serializer__` (*dicee.scripts.index_serve.StringListRequest attribute*), 154
- `__pydantic_validator__` (*dicee.scripts.index_serve.StringListRequest attribute*), 154
- `__setstate__` () (*dicee.models.ADOPT method*), 97
- `__setstate__` () (*dicee.models.adopt.ADOPT method*), 55
- `__signature__` (*dicee.scripts.index_serve.StringListRequest attribute*), 153
- `__str__` () (*dicee.EnsembleKGE method*), 192
- `__str__` () (*dicee.KGE method*), 195
- `__str__` () (*dicee.knowledge_graph_embeddings.KGE method*), 47
- `__str__` () (*dicee.models.ensemble.EnsembleKGE method*), 76
- `__version__` (*in module dicee*), 212

A

- `AbstractCallback` (*class in dicee.abstracts*), 16
- `AbstractPPECallback` (*class in dicee.abstracts*), 17
- `AbstractTrainer` (*class in dicee.abstracts*), 12
- `AccumulateEpochLossCallback` (*class in dicee.callbacks*), 20
- `achieve_answer` () (*dicee.query_generator.QueryGenerator method*), 144
- `achieve_answer` () (*dicee.QueryGenerator method*), 212
- `AConEx` (*class in dicee*), 178
- `AConEx` (*class in dicee.models*), 113
- `AConEx` (*class in dicee.models.complex*), 73
- `AConvO` (*class in dicee*), 179
- `AConvO` (*class in dicee.models*), 125
- `AConvO` (*class in dicee.models.octonion*), 82
- `AConvQ` (*class in dicee*), 180
- `AConvQ` (*class in dicee.models*), 119
- `AConvQ` (*class in dicee.models.quaternion*), 86
- `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*), 193
- `add_noisy_triples` () (*in module dicee.static_funcs*), 157
- `add_noisy_triples_into_training` () (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method*), 146
- `add_noisy_triples_into_training` () (*dicee.read_preprocess_save_load_kg.ReadFromDisk method*), 151
- `add_reciprocal` (*dicee.knowledge_graph.KG attribute*), 45
- `ADOPT` (*class in dicee.models*), 97
- `ADOPT` (*class in dicee.models.adopt*), 55
- `adopt` () (*in module dicee.models.adopt*), 55
- `AllvsAll` (*class in dicee*), 203
- `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*), 198
- `answer_multi_hop_query` () (*dicee.knowledge_graph_embeddings.KGE method*), 49
- `app` (*in module dicee.scripts.index_serve*), 153
- `apply_coefficients` () (*dicee.DeCaL method*), 175
- `apply_coefficients` () (*dicee.Keci method*), 171
- `apply_coefficients` () (*dicee.models.clifford.DeCaL method*), 70
- `apply_coefficients` () (*dicee.models.clifford.Keci method*), 67
- `apply_coefficients` () (*dicee.models.DeCaL method*), 131
- `apply_coefficients` () (*dicee.models.Keci method*), 127
- `apply_reciprical_or_noise` () (*in module dicee.read_preprocess_save_load_kg.util*), 149
- `apply_semantic_constraint` (*dicee.abstracts.BaseInteractiveKGE attribute*), 14
- `apply_unit_norm` (*dicee.BaseKGE attribute*), 190
- `apply_unit_norm` (*dicee.models.base_model.BaseKGE attribute*), 62
- `apply_unit_norm` (*dicee.models.BaseKGE attribute*), 104, 107, 110, 115, 121, 134, 137
- `args` (*dicee.BaseKGE attribute*), 190
- `args` (*dicee.DICE_Trainer attribute*), 194
- `args` (*dicee.evaluator.Evaluator attribute*), 42
- `args` (*dicee.executer.Execute attribute*), 43
- `args` (*dicee.models.base_model.BaseKGE attribute*), 62

- args (*dicee.models.base_model.IdentityClass* attribute), 65
- args (*dicee.models.BaseKGE* attribute), 103, 107, 110, 115, 121, 133, 137
- args (*dicee.models.IdentityClass* attribute), 106, 117, 123
- args (*dicee.models.pykeen_models.PykeenKGE* attribute), 83
- args (*dicee.models.PykeenKGE* attribute), 135
- args (*dicee.PykeenKGE* attribute), 187
- args (*dicee.trainer.DICE_Trainer* attribute), 165
- args (*dicee.trainer.dice_trainer.DICE_Trainer* attribute), 160
- ASWA (*class in dicee.callbacks*), 23
- aswa (*dicee.analyse_experiments.Experiment* attribute), 18
- attn (*dicee.models.transformers.Block* attribute), 94
- attn_dropout (*dicee.models.transformers.CausalSelfAttention* attribute), 92
- attributes (*dicee.abstracts.AbstractTrainer* attribute), 12
- auto_batch_finding (*dicee.config.Namespace* attribute), 28

B

- backend (*dicee.config.Namespace* attribute), 27
- backend (*dicee.knowledge_graph.KG* attribute), 46
- BaseInteractiveKGE (*class in dicee.abstracts*), 14
- BaseKGE (*class in dicee*), 189
- BaseKGE (*class in dicee.models*), 103, 106, 110, 114, 120, 133, 136
- BaseKGE (*class in dicee.models.base_model*), 61
- BaseKGELightning (*class in dicee.models*), 97
- BaseKGELightning (*class in dicee.models.base_model*), 56
- 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), 208
- batch_size (*dicee.dataset_classes.CVDataModule* attribute), 37
- bias (*dicee.models.transformers.GPTConfig* attribute), 94
- bias (*dicee.models.transformers.LayerNorm* attribute), 91
- Block (*class in dicee.models.transformers*), 93
- block_size (*dicee.BaseKGE* attribute), 191
- block_size (*dicee.config.Namespace* attribute), 28
- block_size (*dicee.dataset_classes.MultiClassClassificationDataset* attribute), 31
- block_size (*dicee.models.base_model.BaseKGE* attribute), 63
- block_size (*dicee.models.BaseKGE* attribute), 104, 107, 111, 116, 122, 134, 138
- block_size (*dicee.models.transformers.GPTConfig* attribute), 94
- block_size (*dicee.MultiClassClassificationDataset* attribute), 202
- bn_conv1 (*dicee.AConvQ* attribute), 180
- bn_conv1 (*dicee.ConvQ* attribute), 180
- bn_conv1 (*dicee.models.AConvQ* attribute), 120
- bn_conv1 (*dicee.models.ConvQ* attribute), 119
- bn_conv1 (*dicee.models.quaternion.AConvQ* attribute), 87
- bn_conv1 (*dicee.models.quaternion.ConvQ* attribute), 86
- bn_conv2 (*dicee.AConvQ* attribute), 180
- bn_conv2 (*dicee.ConvQ* attribute), 180
- bn_conv2 (*dicee.models.AConvQ* attribute), 120
- bn_conv2 (*dicee.models.ConvQ* attribute), 119
- bn_conv2 (*dicee.models.quaternion.AConvQ* attribute), 87
- bn_conv2 (*dicee.models.quaternion.ConvQ* attribute), 86
- bn_conv2d (*dicee.AConEx* attribute), 179
- bn_conv2d (*dicee.AConvO* attribute), 179
- bn_conv2d (*dicee.ConEx* attribute), 182
- bn_conv2d (*dicee.ConvO* attribute), 181
- bn_conv2d (*dicee.models.AConEx* attribute), 113
- bn_conv2d (*dicee.models.AConvO* attribute), 126
- bn_conv2d (*dicee.models.complex.AConEx* attribute), 73
- bn_conv2d (*dicee.models.complex.ConEx* attribute), 72
- bn_conv2d (*dicee.models.ConEx* attribute), 112
- bn_conv2d (*dicee.models.ConvO* attribute), 125
- bn_conv2d (*dicee.models.octonion.AConvO* attribute), 82
- bn_conv2d (*dicee.models.octonion.ConvO* attribute), 82
- BPE_NegativeSamplingDataset (*class in dicee*), 200
- BPE_NegativeSamplingDataset (*class in dicee.dataset_classes*), 29
- build_chain_funcs () (*dicee.models.FMult2* method), 140

build_chain_funcs() (*dicee.models.function_space.FMult2 method*), 78
 build_func() (*dicee.models.FMult2 method*), 140
 build_func() (*dicee.models.function_space.FMult2 method*), 78
 Byte (*class in dicee*), 187
 Byte (*class in dicee.models.transformers*), 89
 byte_pair_encoding (*dicee.analyse_experiments.Experiment attribute*), 18
 byte_pair_encoding (*dicee.BaseKGE attribute*), 191
 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*), 63
 byte_pair_encoding (*dicee.models.BaseKGE attribute*), 104, 107, 111, 116, 122, 134, 138

C

c_attn (*dicee.models.transformers.CausalSelfAttention attribute*), 92
 c_fc (*dicee.models.transformers.MLP attribute*), 93
 c_proj (*dicee.models.transformers.CausalSelfAttention attribute*), 92
 c_proj (*dicee.models.transformers.MLP attribute*), 93
 callbacks (*dicee.abstracts.AbstractTrainer attribute*), 12
 callbacks (*dicee.analyse_experiments.Experiment attribute*), 18
 callbacks (*dicee.config.Namespace attribute*), 27
 callbacks (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 164
 CausalSelfAttention (*class in dicee.models.transformers*), 91
 chain_func() (*dicee.models.FMult method*), 139
 chain_func() (*dicee.models.function_space.FMult method*), 77
 chain_func() (*dicee.models.function_space.GFMult method*), 77
 chain_func() (*dicee.models.GFMult method*), 139
 CKeci (*class in dicee*), 170
 CKeci (*class in dicee.models*), 129
 CKeci (*class in dicee.models.clifford*), 68
 cl_pqr() (*dicee.DeCaL method*), 174
 cl_pqr() (*dicee.models.clifford.DeCaL method*), 69
 cl_pqr() (*dicee.models.DeCaL method*), 130
 clifford_multiplication() (*dicee.Keci method*), 171
 clifford_multiplication() (*dicee.models.clifford.Keci method*), 67
 clifford_multiplication() (*dicee.models.Keci method*), 127
 clip_lambda (*dicee.models.ADOPT attribute*), 97
 clip_lambda (*dicee.models.adapt.ADOPT attribute*), 55
 collate_fn (*dicee.AllvsAll attribute*), 204
 collate_fn (*dicee.dataset_classes.AllvsAll attribute*), 33
 collate_fn (*dicee.dataset_classes.KvsAll attribute*), 32
 collate_fn (*dicee.dataset_classes.KvsSampleDataset attribute*), 35
 collate_fn (*dicee.dataset_classes.MultiClassClassificationDataset attribute*), 31
 collate_fn (*dicee.dataset_classes.MultiLabelDataset attribute*), 30
 collate_fn (*dicee.dataset_classes.OnevsAllDataset attribute*), 31
 collate_fn (*dicee.dataset_classes.OnevsSample attribute*), 34
 collate_fn (*dicee.KvsAll attribute*), 203
 collate_fn (*dicee.KvsSampleDataset attribute*), 206
 collate_fn (*dicee.MultiClassClassificationDataset attribute*), 202
 collate_fn (*dicee.MultiLabelDataset attribute*), 201
 collate_fn (*dicee.OnevsAllDataset attribute*), 202
 collate_fn (*dicee.OnevsSample attribute*), 204, 205
 collate_fn() (*dicee.BPE_NegativeSamplingDataset method*), 201
 collate_fn() (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 30
 collate_fn() (*dicee.dataset_classes.TriplePredictionDataset method*), 37
 collate_fn() (*dicee.TriplePredictionDataset method*), 207
 collection_name (*dicee.scripts.index_serve.NeuralSearcher attribute*), 153
 comp_func() (*dicee.LFMult method*), 186
 comp_func() (*dicee.models.function_space.LFMult method*), 79
 comp_func() (*dicee.models.LFMult method*), 141
 Complex (*class in dicee*), 177
 Complex (*class in dicee.models*), 113
 Complex (*class in dicee.models.complex*), 73
 compute_convergence() (*in module dicee.callbacks*), 23
 compute_func() (*dicee.models.FMult method*), 139
 compute_func() (*dicee.models.FMult2 method*), 140
 compute_func() (*dicee.models.function_space.FMult method*), 77
 compute_func() (*dicee.models.function_space.FMult2 method*), 78

`compute_func()` (*dicee.models.function_space.GFMult method*), 77
`compute_func()` (*dicee.models.GFMult method*), 139
`compute_mrr()` (*dicee.callbacks.ASWA static method*), 23
`compute_sigma_pp()` (*dicee.DeCaL method*), 175
`compute_sigma_pp()` (*dicee.Keci method*), 171
`compute_sigma_pp()` (*dicee.models.clifford.DeCaL method*), 70
`compute_sigma_pp()` (*dicee.models.clifford.Keci method*), 66
`compute_sigma_pp()` (*dicee.models.DeCaL method*), 131
`compute_sigma_pp()` (*dicee.models.Keci method*), 127
`compute_sigma_pq()` (*dicee.DeCaL method*), 176
`compute_sigma_pq()` (*dicee.Keci method*), 171
`compute_sigma_pq()` (*dicee.models.clifford.DeCaL method*), 71
`compute_sigma_pq()` (*dicee.models.clifford.Keci method*), 66
`compute_sigma_pq()` (*dicee.models.DeCaL method*), 132
`compute_sigma_pq()` (*dicee.models.Keci method*), 127
`compute_sigma_pr()` (*dicee.DeCaL method*), 176
`compute_sigma_pr()` (*dicee.models.clifford.DeCaL method*), 72
`compute_sigma_pr()` (*dicee.models.DeCaL method*), 132
`compute_sigma_qq()` (*dicee.DeCaL method*), 175
`compute_sigma_qq()` (*dicee.Keci method*), 171
`compute_sigma_qq()` (*dicee.models.clifford.DeCaL method*), 71
`compute_sigma_qq()` (*dicee.models.clifford.Keci method*), 66
`compute_sigma_qq()` (*dicee.models.DeCaL method*), 132
`compute_sigma_qq()` (*dicee.models.Keci method*), 127
`compute_sigma_qr()` (*dicee.DeCaL method*), 176
`compute_sigma_qr()` (*dicee.models.clifford.DeCaL method*), 72
`compute_sigma_qr()` (*dicee.models.DeCaL method*), 133
`compute_sigma_rr()` (*dicee.DeCaL method*), 176
`compute_sigma_rr()` (*dicee.models.clifford.DeCaL method*), 71
`compute_sigma_rr()` (*dicee.models.DeCaL method*), 132
`compute_sigmas_multivect()` (*dicee.DeCaL method*), 174
`compute_sigmas_multivect()` (*dicee.models.clifford.DeCaL method*), 70
`compute_sigmas_multivect()` (*dicee.models.DeCaL method*), 131
`compute_sigmas_single()` (*dicee.DeCaL method*), 174
`compute_sigmas_single()` (*dicee.models.clifford.DeCaL method*), 69
`compute_sigmas_single()` (*dicee.models.DeCaL method*), 130
`ConEx` (*class in dicee*), 182
`ConEx` (*class in dicee.models*), 112
`ConEx` (*class in dicee.models.complex*), 72
`config` (*dicee.BytE attribute*), 188
`config` (*dicee.models.transformers.BytE attribute*), 90
`config` (*dicee.models.transformers.GPT attribute*), 95
`configs` (*dicee.abstracts.BaseInteractiveKGE attribute*), 14
`configure_optimizers()` (*dicee.models.base_model.BaseKGELightning method*), 60
`configure_optimizers()` (*dicee.models.BaseKGELightning method*), 101
`configure_optimizers()` (*dicee.models.transformers.GPT method*), 95
`construct_batch_selected_cl_multivector()` (*dicee.Keci method*), 172
`construct_batch_selected_cl_multivector()` (*dicee.models.clifford.Keci method*), 67
`construct_batch_selected_cl_multivector()` (*dicee.models.Keci method*), 128
`construct_cl_multivector()` (*dicee.DeCaL method*), 175
`construct_cl_multivector()` (*dicee.Keci method*), 172
`construct_cl_multivector()` (*dicee.models.clifford.DeCaL method*), 70
`construct_cl_multivector()` (*dicee.models.clifford.Keci method*), 67
`construct_cl_multivector()` (*dicee.models.DeCaL method*), 131
`construct_cl_multivector()` (*dicee.models.Keci method*), 128
`construct_dataset()` (*in module dicee*), 200
`construct_dataset()` (*in module dicee.dataset_classes*), 29
`construct_ensemble` (*dicee.abstracts.BaseInteractiveKGE attribute*), 14
`construct_graph()` (*dicee.query_generator.QueryGenerator method*), 144
`construct_graph()` (*dicee.QueryGenerator method*), 212
`construct_input_and_output()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
`construct_multi_coeff()` (*dicee.LFMult method*), 185
`construct_multi_coeff()` (*dicee.models.function_space.LFMult method*), 79
`construct_multi_coeff()` (*dicee.models.LFMult method*), 141
`continual_learning` (*dicee.config.Namespace attribute*), 28
`continual_start()` (*dicee.DICE_Trainer method*), 194
`continual_start()` (*dicee.executer.ContinuousExecute method*), 44
`continual_start()` (*dicee.trainer.DICE_Trainer method*), 165

continual_start() (*dicee.trainer.dice_trainer.DICE_Trainer method*), 160
 continual_training_setup_executor() (*in module dicee*), 193
 continual_training_setup_executor() (*in module dicee.static_funcs*), 157
 ContinuousExecute (*class in dicee.executor*), 44
 conv2d (*dicee.AConEx attribute*), 179
 conv2d (*dicee.AConvO attribute*), 179
 conv2d (*dicee.AConvQ attribute*), 180
 conv2d (*dicee.ConEx attribute*), 182
 conv2d (*dicee.ConvO attribute*), 181
 conv2d (*dicee.ConvQ attribute*), 180
 conv2d (*dicee.models.AConEx attribute*), 113
 conv2d (*dicee.models.AConvO attribute*), 125
 conv2d (*dicee.models.AConvQ attribute*), 120
 conv2d (*dicee.models.complex.AConEx attribute*), 73
 conv2d (*dicee.models.complex.ConEx attribute*), 72
 conv2d (*dicee.models.ConEx attribute*), 112
 conv2d (*dicee.models.ConvO attribute*), 125
 conv2d (*dicee.models.ConvQ attribute*), 119
 conv2d (*dicee.models.octonion.AConvO attribute*), 82
 conv2d (*dicee.models.octonion.ConvO attribute*), 82
 conv2d (*dicee.models.quaternion.AConvQ attribute*), 87
 conv2d (*dicee.models.quaternion.ConvQ attribute*), 86
 ConvO (*class in dicee*), 181
 ConvO (*class in dicee.models*), 124
 ConvO (*class in dicee.models.octonion*), 81
 ConvQ (*class in dicee*), 180
 ConvQ (*class in dicee.models*), 119
 ConvQ (*class in dicee.models.quaternion*), 86
 create_constraints() (*in module dicee.read_preprocess_save_load_kg.util*), 149
 create_constraints() (*in module dicee.static_preprocess_funcs*), 159
 create_experiment_folder() (*in module dicee*), 193
 create_experiment_folder() (*in module dicee.static_funcs*), 157
 create_random_data() (*dicee.callbacks.PseudoLabellingCallback method*), 22
 create_recipriocal_triples() (*in module dicee*), 192
 create_recipriocal_triples() (*in module dicee.read_preprocess_save_load_kg.util*), 150
 create_recipriocal_triples() (*in module dicee.static_funcs*), 156
 create_vector_database() (*dicee.KGE method*), 195
 create_vector_database() (*dicee.knowledge_graph_embeddings.KGE method*), 47
 crop_block_size() (*dicee.models.transformers.GPT method*), 95
 ctx (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 164
 CVDataModule (*class in dicee*), 207
 CVDataModule (*class in dicee.dataset_classes*), 37

D

data_module (*dicee.callbacks.PseudoLabellingCallback attribute*), 22
 data_property_embeddings (*dicee.literal_classes.LiteralEmbeddings attribute*), 52
 data_property_to_idx (*dicee.literal_classes.LiteralDataset attribute*), 53
 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*), 150
 DeCaL (*class in dicee*), 173
 DeCaL (*class in dicee.models*), 129
 DeCaL (*class in dicee.models.clifford*), 68
 decide() (*dicee.callbacks.ASWA method*), 23
 degree (*dicee.LFMMult attribute*), 185
 degree (*dicee.models.function_space.LFMMult attribute*), 79
 degree (*dicee.models.LFMMult attribute*), 141
 denormalize() (*dicee.literal_classes.LiteralDataset static method*), 54
 deploy() (*dicee.KGE method*), 198
 deploy() (*dicee.knowledge_graph_embeddings.KGE method*), 50
 deploy_head_entity_prediction() (*in module dicee*), 193
 deploy_head_entity_prediction() (*in module dicee.static_funcs*), 157
 deploy_relation_prediction() (*in module dicee*), 193
 deploy_relation_prediction() (*in module dicee.static_funcs*), 157
 deploy_tail_entity_prediction() (*in module dicee*), 193
 deploy_tail_entity_prediction() (*in module dicee.static_funcs*), 157
 deploy_triple_prediction() (*in module dicee*), 193

- `deploy_triple_prediction()` (in module *dicee.static_funcs*), 157
- `describe()` (*dicee.knowledge_graph.KG* method), 46
- `description_of_input` (*dicee.knowledge_graph.KG* attribute), 46
- `device` (*dicee.literal_classes.LiteralEmbeddings* property), 53
- `DICE_Trainer` (class in *dicee*), 194
- `DICE_Trainer` (class in *dicee.trainer*), 165
- `DICE_Trainer` (class in *dicee.trainer.dice_trainer*), 160
- dicee*
 - module, 12
- dicee.__main__*
 - module, 12
- dicee.abstracts*
 - module, 12
- dicee.analyse_experiments*
 - module, 18
- dicee.callbacks*
 - module, 19
- dicee.config*
 - module, 26
- dicee.dataset_classes*
 - module, 29
- dicee.eval_static_funcs*
 - module, 40
- dicee.evaluator*
 - module, 42
- dicee.executer*
 - module, 43
- dicee.knowledge_graph*
 - module, 45
- dicee.knowledge_graph_embeddings*
 - module, 46
- dicee.literal_classes*
 - module, 51
- dicee.models*
 - module, 55
- dicee.models.adopt*
 - module, 55
- dicee.models.base_model*
 - module, 56
- dicee.models.clifford*
 - module, 65
- dicee.models.complex*
 - module, 72
- dicee.models.dualE*
 - module, 74
- dicee.models.ensemble*
 - module, 76
- dicee.models.function_space*
 - module, 76
- dicee.models.octonion*
 - module, 80
- dicee.models.pykeen_models*
 - module, 83
- dicee.models.quaternion*
 - module, 84
- dicee.models.real*
 - module, 87
- dicee.models.static_funcs*
 - module, 89
- dicee.models.transformers*
 - module, 89
- dicee.query_generator*
 - module, 143
- dicee.read_preprocess_save_load_kg*
 - module, 144
- dicee.read_preprocess_save_load_kg.preprocess*
 - module, 144
- dicee.read_preprocess_save_load_kg.read_from_disk*

- module, 145
- `dicee.read_preprocess_save_load_kg.save_load_disk`
 - module, 146
- `dicee.read_preprocess_save_load_kg.util`
 - module, 146
- `dicee.sanity_checkers`
 - module, 151
- `dicee.scripts`
 - module, 152
- `dicee.scripts.index_serve`
 - module, 152
- `dicee.scripts.run`
 - module, 154
- `dicee.static_funcs`
 - module, 155
- `dicee.static_funcs_training`
 - module, 158
- `dicee.static_preprocess_funcs`
 - module, 158
- `dicee.trainer`
 - module, 159
- `dicee.trainer.dice_trainer`
 - module, 159
- `dicee.trainer.model_parallelism`
 - module, 161
- `dicee.trainer.torch_trainer`
 - module, 162
- `dicee.trainer.torch_trainer_ddp`
 - module, 163
- `discrete_points` (*dicee.models.FMult2 attribute*), 140
- `discrete_points` (*dicee.models.function_space.FMult2 attribute*), 78
- `dist_func` (*dicee.models.Pyke attribute*), 109
- `dist_func` (*dicee.models.real.Pyke attribute*), 88
- `dist_func` (*dicee.Pyke attribute*), 169
- `DistMult` (*class in dicee*), 169
- `DistMult` (*class in dicee.models*), 108
- `DistMult` (*class in dicee.models.real*), 87
- `download_file()` (*in module dicee*), 194
- `download_file()` (*in module dicee.static_funcs*), 157
- `download_files_from_url()` (*in module dicee*), 194
- `download_files_from_url()` (*in module dicee.static_funcs*), 157
- `download_pretrained_model()` (*in module dicee*), 194
- `download_pretrained_model()` (*in module dicee.static_funcs*), 158
- `dropout` (*dicee.literal_classes.LiteralEmbeddings attribute*), 52
- `dropout` (*dicee.models.transformers.CausalSelfAttention attribute*), 92
- `dropout` (*dicee.models.transformers.GPTConfig attribute*), 94
- `dropout` (*dicee.models.transformers.MLP attribute*), 93
- `DualE` (*class in dicee*), 177
- `DualE` (*class in dicee.models*), 142
- `DualE` (*class in dicee.models.dualE*), 75
- `dummy_eval()` (*dicee.evaluator.Evaluator method*), 43
- `dummy_id` (*dicee.knowledge_graph.KG attribute*), 46
- `during_training` (*dicee.evaluator.Evaluator attribute*), 42

E

- `ee_vocab` (*dicee.evaluator.Evaluator attribute*), 42
- `efficient_zero_grad()` (*in module dicee.static_funcs_training*), 158
- `embedding_dim` (*dicee.analyse_experiments.Experiment attribute*), 18
- `embedding_dim` (*dicee.BaseKGE attribute*), 190
- `embedding_dim` (*dicee.config.Namespace attribute*), 26
- `embedding_dim` (*dicee.literal_classes.LiteralEmbeddings attribute*), 52
- `embedding_dim` (*dicee.models.base_model.BaseKGE attribute*), 62
- `embedding_dim` (*dicee.models.BaseKGE attribute*), 103, 107, 110, 115, 121, 133, 137
- `embedding_dims` (*dicee.literal_classes.LiteralEmbeddings attribute*), 52
- `enable_log` (*in module dicee.static_preprocess_funcs*), 159
- `enc` (*dicee.knowledge_graph.KG attribute*), 46
- `end()` (*dicee.executer.Execute method*), 44

EnsembleKGE (*class in dicee*), 192
 EnsembleKGE (*class in dicee.models.ensemble*), 76
 ent2id (*dicee.query_generator.QueryGenerator attribute*), 143
 ent2id (*dicee.QueryGenerator attribute*), 211
 ent_in (*dicee.query_generator.QueryGenerator attribute*), 143
 ent_in (*dicee.QueryGenerator attribute*), 211
 ent_out (*dicee.query_generator.QueryGenerator attribute*), 143
 ent_out (*dicee.QueryGenerator attribute*), 211
 entities_str (*dicee.knowledge_graph.KG property*), 46
 entity_embeddings (*dicee.AConvQ attribute*), 180
 entity_embeddings (*dicee.ConvQ attribute*), 180
 entity_embeddings (*dicee.DeCaL attribute*), 174
 entity_embeddings (*dicee.DualE attribute*), 177
 entity_embeddings (*dicee.LFMult attribute*), 185
 entity_embeddings (*dicee.literal_classes.LiteralEmbeddings attribute*), 52
 entity_embeddings (*dicee.models.AConvQ attribute*), 120
 entity_embeddings (*dicee.models.clifford.DeCaL attribute*), 69
 entity_embeddings (*dicee.models.ConvQ attribute*), 119
 entity_embeddings (*dicee.models.DeCaL attribute*), 130
 entity_embeddings (*dicee.models.DualE attribute*), 142
 entity_embeddings (*dicee.models.dualE.DualE attribute*), 75
 entity_embeddings (*dicee.models.FMult attribute*), 139
 entity_embeddings (*dicee.models.FMult2 attribute*), 140
 entity_embeddings (*dicee.models.function_space.FMult attribute*), 77
 entity_embeddings (*dicee.models.function_space.FMult2 attribute*), 78
 entity_embeddings (*dicee.models.function_space.GFMult attribute*), 77
 entity_embeddings (*dicee.models.function_space.LFMult attribute*), 79
 entity_embeddings (*dicee.models.function_space.LFMult1 attribute*), 78
 entity_embeddings (*dicee.models.GFMult attribute*), 139
 entity_embeddings (*dicee.models.LFMult attribute*), 141
 entity_embeddings (*dicee.models.LFMult1 attribute*), 140
 entity_embeddings (*dicee.models.pykeen_models.PykeenKGE attribute*), 83
 entity_embeddings (*dicee.models.PykeenKGE attribute*), 135
 entity_embeddings (*dicee.models.quaternion.AConvQ attribute*), 86
 entity_embeddings (*dicee.models.quaternion.ConvQ attribute*), 86
 entity_embeddings (*dicee.PykeenKGE attribute*), 187
 entity_to_idx (*dicee.knowledge_graph.KG attribute*), 45
 entity_to_idx (*dicee.literal_classes.LiteralDataset attribute*), 53, 54
 entity_to_idx (*dicee.scripts.index_serve.NeuralSearcher attribute*), 153
 epoch_count (*dicee.abstracts.AbstractPPECallback attribute*), 17
 epoch_count (*dicee.callbacks.ASWA attribute*), 23
 epoch_counter (*dicee.callbacks.Eval attribute*), 24
 epoch_counter (*dicee.callbacks.KGESaveCallback attribute*), 21
 epoch_ratio (*dicee.callbacks.Eval attribute*), 24
 er_vocab (*dicee.evaluator.Evaluator attribute*), 42
 estimate_mfu () (*dicee.models.transformers.GPT method*), 95
 estimate_q () (*in module dicee.callbacks*), 22
 Eval (*class in dicee.callbacks*), 24
 eval () (*dicee.EnsembleKGE method*), 192
 eval () (*dicee.evaluator.Evaluator method*), 42
 eval () (*dicee.models.ensemble.EnsembleKGE method*), 76
 eval_lp_performance () (*dicee.KGE method*), 196
 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*), 43
 eval_with_vs_all () (*dicee.evaluator.Evaluator method*), 42
 evaluate () (*in module dicee*), 194
 evaluate () (*in module dicee.static_funcs*), 157
 evaluate_bpe_lp () (*in module dicee.static_funcs_training*), 158
 evaluate_link_prediction_performance () (*in module dicee.eval_static_funcs*), 41
 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_literal_prediction() (*dicее.KGE method*), 199
 evaluate_literal_prediction() (*dicее.knowledge_graph_embeddings.KGE method*), 51
 evaluate_lp() (*dicее.evaluator.Evaluator method*), 43
 evaluate_lp() (*in module dicее.static_funcs_training*), 158
 evaluate_lp_bpe_k_vs_all() (*dicее.evaluator.Evaluator method*), 43
 evaluate_lp_bpe_k_vs_all() (*in module dicее.eval_static_funcs*), 41
 evaluate_lp_k_vs_all() (*dicее.evaluator.Evaluator method*), 43
 evaluate_lp_with_byte() (*dicее.evaluator.Evaluator method*), 43
 Evaluator (*class in dicее.evaluator*), 42
 evaluator (*dicее.DICE_Trainer attribute*), 194
 evaluator (*dicее.executer.Execute attribute*), 43
 evaluator (*dicее.trainer.DICE_Trainer attribute*), 165
 evaluator (*dicее.trainer.dice_trainer.DICE_Trainer attribute*), 160
 every_x_epoch (*dicее.callbacks.KGESaveCallback attribute*), 21
 example_input_array (*dicее.EnsembleKGE property*), 192
 example_input_array (*dicее.models.ensemble.EnsembleKGE property*), 76
 Execute (*class in dicее.executer*), 43
 exists() (*dicее.knowledge_graph.KG method*), 46
 Experiment (*class in dicее.analyse_experiments*), 18
 explicit (*dicее.models.QMult attribute*), 118
 explicit (*dicее.models.quaternion.QMult attribute*), 85
 explicit (*dicее.QMult attribute*), 183
 exponential_function() (*in module dicее*), 193
 exponential_function() (*in module dicее.static_funcs*), 157
 extract_input_outputs() (*dicее.trainer.torch_trainer_ddp.NodeTrainer method*), 164
 extract_input_outputs() (*in module dicее.trainer.model_parallelism*), 162
 extract_input_outputs_set_device() (*dicее.trainer.torch_trainer.TorchTrainer method*), 163

F

f (*dicее.callbacks.KronE attribute*), 25
 fc (*dicее.literal_classes.LiteralEmbeddings attribute*), 52
 fc1 (*dicее.AConEx attribute*), 179
 fc1 (*dicее.AConvO attribute*), 179
 fc1 (*dicее.AConvQ attribute*), 180
 fc1 (*dicее.ConEx attribute*), 182
 fc1 (*dicее.ConvO attribute*), 181
 fc1 (*dicее.ConvQ attribute*), 180
 fc1 (*dicее.models.AConEx attribute*), 113
 fc1 (*dicее.models.AConvO attribute*), 125
 fc1 (*dicее.models.AConvQ attribute*), 120
 fc1 (*dicее.models.complex.AConEx attribute*), 73
 fc1 (*dicее.models.complex.ConEx attribute*), 72
 fc1 (*dicее.models.ConEx attribute*), 112
 fc1 (*dicее.models.ConvO attribute*), 125
 fc1 (*dicее.models.ConvQ attribute*), 119
 fc1 (*dicее.models.octonion.AConvO attribute*), 82
 fc1 (*dicее.models.octonion.ConvO attribute*), 82
 fc1 (*dicее.models.quaternion.AConvQ attribute*), 87
 fc1 (*dicее.models.quaternion.ConvQ attribute*), 86
 fc_num_input (*dicее.AConEx attribute*), 179
 fc_num_input (*dicее.AConvO attribute*), 179
 fc_num_input (*dicее.AConvQ attribute*), 180
 fc_num_input (*dicее.ConEx attribute*), 182
 fc_num_input (*dicее.ConvO attribute*), 181
 fc_num_input (*dicее.ConvQ attribute*), 180
 fc_num_input (*dicее.models.AConEx attribute*), 113
 fc_num_input (*dicее.models.AConvO attribute*), 125
 fc_num_input (*dicее.models.AConvQ attribute*), 120
 fc_num_input (*dicее.models.complex.AConEx attribute*), 73
 fc_num_input (*dicее.models.complex.ConEx attribute*), 72
 fc_num_input (*dicее.models.ConEx attribute*), 112
 fc_num_input (*dicее.models.ConvO attribute*), 125
 fc_num_input (*dicее.models.ConvQ attribute*), 119
 fc_num_input (*dicее.models.octonion.AConvO attribute*), 82
 fc_num_input (*dicее.models.octonion.ConvO attribute*), 82
 fc_num_input (*dicее.models.quaternion.AConvQ attribute*), 87
 fc_num_input (*dicее.models.quaternion.ConvQ attribute*), 86

`fc_out` (*dicee.literal_classes.LiteralEmbeddings* attribute), 52
`feature_map_dropout` (*dicee.AConEx* attribute), 179
`feature_map_dropout` (*dicee.AConvO* attribute), 179
`feature_map_dropout` (*dicee.AConvQ* attribute), 180
`feature_map_dropout` (*dicee.ConEx* attribute), 182
`feature_map_dropout` (*dicee.ConvO* attribute), 182
`feature_map_dropout` (*dicee.ConvQ* attribute), 180
`feature_map_dropout` (*dicee.models.AConEx* attribute), 113
`feature_map_dropout` (*dicee.models.AConvO* attribute), 126
`feature_map_dropout` (*dicee.models.AConvQ* attribute), 120
`feature_map_dropout` (*dicee.models.complex.AConEx* attribute), 73
`feature_map_dropout` (*dicee.models.complex.ConEx* attribute), 72
`feature_map_dropout` (*dicee.models.ConEx* attribute), 112
`feature_map_dropout` (*dicee.models.ConvO* attribute), 125
`feature_map_dropout` (*dicee.models.ConvQ* attribute), 119
`feature_map_dropout` (*dicee.models.octonion.AConvO* attribute), 82
`feature_map_dropout` (*dicee.models.octonion.ConvO* attribute), 82
`feature_map_dropout` (*dicee.models.quaternion.AConvQ* attribute), 87
`feature_map_dropout` (*dicee.models.quaternion.ConvQ* attribute), 86
`feature_map_dropout_rate` (*dicee.BaseKGE* attribute), 190
`feature_map_dropout_rate` (*dicee.config.Namespace* attribute), 28
`feature_map_dropout_rate` (*dicee.models.base_model.BaseKGE* attribute), 62
`feature_map_dropout_rate` (*dicee.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137
`fill_query()` (*dicee.query_generator.QueryGenerator* method), 144
`fill_query()` (*dicee.QueryGenerator* method), 212
`find_good_batch_size()` (in module *dicee.trainer.model_parallelism*), 162
`find_missing_triples()` (*dicee.KGE* method), 198
`find_missing_triples()` (*dicee.knowledge_graph_embeddings.KGE* method), 49
`fit()` (*dicee.trainer.model_parallelism.TensorParallel* method), 162
`fit()` (*dicee.trainer.torch_trainer_ddp.TorchDDPTrainer* method), 164
`fit()` (*dicee.trainer.torch_trainer.TorchTrainer* method), 163
`flash` (*dicee.models.transformers.CausalSelfAttention* attribute), 92
`FMult` (class in *dicee.models*), 139
`FMult` (class in *dicee.models.function_space*), 77
`FMult2` (class in *dicee.models*), 139
`FMult2` (class in *dicee.models.function_space*), 77
`form_of_labelling` (*dicee.DICE_Trainer* attribute), 194
`form_of_labelling` (*dicee.trainer.DICE_Trainer* attribute), 165
`form_of_labelling` (*dicee.trainer.dice_trainer.DICE_Trainer* attribute), 160
`forward()` (*dicee.BaseKGE* method), 191
`forward()` (*dicee.BytE* method), 188
`forward()` (*dicee.literal_classes.GatedLinearUnit* method), 51
`forward()` (*dicee.literal_classes.LiteralEmbeddings* method), 52
`forward()` (*dicee.models.base_model.BaseKGE* method), 63
`forward()` (*dicee.models.base_model.IdentityClass* static method), 65
`forward()` (*dicee.models.BaseKGE* method), 105, 108, 111, 116, 122, 135, 138
`forward()` (*dicee.models.IdentityClass* static method), 106, 117, 123
`forward()` (*dicee.models.transformers.Block* method), 94
`forward()` (*dicee.models.transformers.BytE* method), 90
`forward()` (*dicee.models.transformers.CausalSelfAttention* method), 92
`forward()` (*dicee.models.transformers.GPT* method), 95
`forward()` (*dicee.models.transformers.LayerNorm* method), 91
`forward()` (*dicee.models.transformers.MLP* method), 93
`forward_backward_update()` (*dicee.trainer.torch_trainer.TorchTrainer* method), 163
`forward_backward_update_loss()` (in module *dicee.trainer.model_parallelism*), 162
`forward_byte_pair_encoded_k_vs_all()` (*dicee.BaseKGE* method), 191
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.base_model.BaseKGE* method), 63
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.BaseKGE* method), 104, 107, 111, 116, 122, 134, 138
`forward_byte_pair_encoded_triple()` (*dicee.BaseKGE* method), 191
`forward_byte_pair_encoded_triple()` (*dicee.models.base_model.BaseKGE* method), 63
`forward_byte_pair_encoded_triple()` (*dicee.models.BaseKGE* method), 104, 108, 111, 116, 122, 134, 138
`forward_k_vs_all()` (*dicee.AConEx* method), 179
`forward_k_vs_all()` (*dicee.AConvO* method), 179
`forward_k_vs_all()` (*dicee.AConvQ* method), 180
`forward_k_vs_all()` (*dicee.BaseKGE* method), 191
`forward_k_vs_all()` (*dicee.ComplEx* method), 178
`forward_k_vs_all()` (*dicee.ConEx* method), 182
`forward_k_vs_all()` (*dicee.ConvO* method), 182

`forward_k_vs_all()` (*dicee.ConvQ method*), 181
`forward_k_vs_all()` (*dicee.DeCaL method*), 175
`forward_k_vs_all()` (*dicee.DistMult method*), 170
`forward_k_vs_all()` (*dicee.DualE method*), 177
`forward_k_vs_all()` (*dicee.Keci method*), 172
`forward_k_vs_all()` (*dicee.models.AConEx method*), 113
`forward_k_vs_all()` (*dicee.models.AConvO method*), 126
`forward_k_vs_all()` (*dicee.models.AConvQ method*), 120
`forward_k_vs_all()` (*dicee.models.base_model.BaseKGE method*), 64
`forward_k_vs_all()` (*dicee.models.BaseKGE method*), 105, 108, 112, 116, 122, 135, 138
`forward_k_vs_all()` (*dicee.models.clifford.DeCaL method*), 70
`forward_k_vs_all()` (*dicee.models.clifford.Keci method*), 67
`forward_k_vs_all()` (*dicee.models.ComplEx method*), 114
`forward_k_vs_all()` (*dicee.models.complex.AConEx method*), 73
`forward_k_vs_all()` (*dicee.models.complex.ComplEx method*), 74
`forward_k_vs_all()` (*dicee.models.complex.ConEx method*), 73
`forward_k_vs_all()` (*dicee.models.ConEx method*), 112
`forward_k_vs_all()` (*dicee.models.ConvO method*), 125
`forward_k_vs_all()` (*dicee.models.ConvQ method*), 119
`forward_k_vs_all()` (*dicee.models.DeCaL method*), 131
`forward_k_vs_all()` (*dicee.models.DistMult method*), 109
`forward_k_vs_all()` (*dicee.models.DualE method*), 142
`forward_k_vs_all()` (*dicee.models.dualE.DualE method*), 75
`forward_k_vs_all()` (*dicee.models.Keci method*), 128
`forward_k_vs_all()` (*dicee.models.octonion.AConvO method*), 83
`forward_k_vs_all()` (*dicee.models.octonion.ConvO method*), 82
`forward_k_vs_all()` (*dicee.models.octonion.OMult method*), 81
`forward_k_vs_all()` (*dicee.models.OMult method*), 124
`forward_k_vs_all()` (*dicee.models.pykeen_models.PykeenKGE method*), 53
`forward_k_vs_all()` (*dicee.models.PykeenKGE method*), 136
`forward_k_vs_all()` (*dicee.models.QMult method*), 119
`forward_k_vs_all()` (*dicee.models.quaternion.AConvQ method*), 87
`forward_k_vs_all()` (*dicee.models.quaternion.ConvQ method*), 86
`forward_k_vs_all()` (*dicee.models.quaternion.QMult method*), 86
`forward_k_vs_all()` (*dicee.models.real.DistMult method*), 87
`forward_k_vs_all()` (*dicee.models.real.Shallom method*), 88
`forward_k_vs_all()` (*dicee.models.real.TransE method*), 88
`forward_k_vs_all()` (*dicee.models.Shallom method*), 109
`forward_k_vs_all()` (*dicee.models.TransE method*), 109
`forward_k_vs_all()` (*dicee.OMult method*), 185
`forward_k_vs_all()` (*dicee.PykeenKGE method*), 187
`forward_k_vs_all()` (*dicee.QMult method*), 184
`forward_k_vs_all()` (*dicee.Shallom method*), 185
`forward_k_vs_all()` (*dicee.TransE method*), 173
`forward_k_vs_sample()` (*dicee.AConEx method*), 179
`forward_k_vs_sample()` (*dicee.BaseKGE method*), 191
`forward_k_vs_sample()` (*dicee.ComplEx method*), 178
`forward_k_vs_sample()` (*dicee.ConEx method*), 182
`forward_k_vs_sample()` (*dicee.DistMult method*), 170
`forward_k_vs_sample()` (*dicee.Keci method*), 172
`forward_k_vs_sample()` (*dicee.models.AConEx method*), 113
`forward_k_vs_sample()` (*dicee.models.base_model.BaseKGE method*), 64
`forward_k_vs_sample()` (*dicee.models.BaseKGE method*), 105, 108, 112, 116, 122, 135, 138
`forward_k_vs_sample()` (*dicee.models.clifford.Keci method*), 68
`forward_k_vs_sample()` (*dicee.models.ComplEx method*), 114
`forward_k_vs_sample()` (*dicee.models.complex.AConEx method*), 73
`forward_k_vs_sample()` (*dicee.models.complex.ComplEx method*), 74
`forward_k_vs_sample()` (*dicee.models.complex.ConEx method*), 73
`forward_k_vs_sample()` (*dicee.models.ConEx method*), 112
`forward_k_vs_sample()` (*dicee.models.DistMult method*), 109
`forward_k_vs_sample()` (*dicee.models.Keci method*), 129
`forward_k_vs_sample()` (*dicee.models.pykeen_models.PykeenKGE method*), 84
`forward_k_vs_sample()` (*dicee.models.PykeenKGE method*), 136
`forward_k_vs_sample()` (*dicee.models.QMult method*), 119
`forward_k_vs_sample()` (*dicee.models.quaternion.QMult method*), 86
`forward_k_vs_sample()` (*dicee.models.real.DistMult method*), 88
`forward_k_vs_sample()` (*dicee.PykeenKGE method*), 187
`forward_k_vs_sample()` (*dicee.QMult method*), 184

`forward_k_vs_with_explicit()` (*dicee.Keci method*), 172
`forward_k_vs_with_explicit()` (*dicee.models.clifford.Keci method*), 67
`forward_k_vs_with_explicit()` (*dicee.models.Keci method*), 128
`forward_triples()` (*dicee.AConEx method*), 179
`forward_triples()` (*dicee.AConvO method*), 179
`forward_triples()` (*dicee.AConvQ method*), 180
`forward_triples()` (*dicee.BaseKGE method*), 191
`forward_triples()` (*dicee.ConEx method*), 182
`forward_triples()` (*dicee.ConvO method*), 182
`forward_triples()` (*dicee.ConvQ method*), 181
`forward_triples()` (*dicee.DeCaL method*), 174
`forward_triples()` (*dicee.DualE method*), 177
`forward_triples()` (*dicee.Keci method*), 173
`forward_triples()` (*dicee.LFMult method*), 185
`forward_triples()` (*dicee.models.AConEx method*), 113
`forward_triples()` (*dicee.models.AConvO method*), 126
`forward_triples()` (*dicee.models.AConvQ method*), 120
`forward_triples()` (*dicee.models.base_model.BaseKGE method*), 63
`forward_triples()` (*dicee.models.BaseKGE method*), 105, 108, 111, 116, 122, 135, 138
`forward_triples()` (*dicee.models.clifford.DeCaL method*), 69
`forward_triples()` (*dicee.models.clifford.Keci method*), 68
`forward_triples()` (*dicee.models.complex.AConEx method*), 73
`forward_triples()` (*dicee.models.complex.ConEx method*), 73
`forward_triples()` (*dicee.models.ConEx method*), 112
`forward_triples()` (*dicee.models.ConvO method*), 125
`forward_triples()` (*dicee.models.ConvQ method*), 119
`forward_triples()` (*dicee.models.DeCaL method*), 130
`forward_triples()` (*dicee.models.DualE method*), 142
`forward_triples()` (*dicee.models.dualE.DualE method*), 75
`forward_triples()` (*dicee.models.FMult method*), 139
`forward_triples()` (*dicee.models.FMult2 method*), 140
`forward_triples()` (*dicee.models.function_space.FMult method*), 77
`forward_triples()` (*dicee.models.function_space.FMult2 method*), 78
`forward_triples()` (*dicee.models.function_space.GFMult method*), 77
`forward_triples()` (*dicee.models.function_space.LFMult method*), 79
`forward_triples()` (*dicee.models.function_space.LFMult1 method*), 78
`forward_triples()` (*dicee.models.GFMult method*), 139
`forward_triples()` (*dicee.models.Keci method*), 129
`forward_triples()` (*dicee.models.LFMult method*), 141
`forward_triples()` (*dicee.models.LFMult1 method*), 140
`forward_triples()` (*dicee.models.octonion.AConvO method*), 83
`forward_triples()` (*dicee.models.octonion.ConvO method*), 82
`forward_triples()` (*dicee.models.Pyke method*), 109
`forward_triples()` (*dicee.models.pykeen_models.PykeenKGE method*), 84
`forward_triples()` (*dicee.models.PykeenKGE method*), 136
`forward_triples()` (*dicee.models.quaternion.AConvQ method*), 87
`forward_triples()` (*dicee.models.quaternion.ConvQ method*), 86
`forward_triples()` (*dicee.models.real.Pyke method*), 88
`forward_triples()` (*dicee.models.real.Shallom method*), 88
`forward_triples()` (*dicee.models.Shallom method*), 109
`forward_triples()` (*dicee.Pyke method*), 169
`forward_triples()` (*dicee.PykeenKGE method*), 187
`forward_triples()` (*dicee.Shallom method*), 185
`freeze_entity_embeddings` (*dicee.literal_classes.LiteralEmbeddings attribute*), 52
`frequency` (*dicee.callbacks.Perturb attribute*), 26
`from_pretrained()` (*dicee.models.transformers.GPT class method*), 95
`from_pretrained_model_write_embeddings_into_csv()` (*in module dicee*), 194
`from_pretrained_model_write_embeddings_into_csv()` (*in module dicee.static_funcs*), 158
`full_storage_path` (*dicee.analyse_experiments.Experiment attribute*), 18
`func_triple_to_bpe_representation` (*dicee.evaluator.Evaluator attribute*), 42
`func_triple_to_bpe_representation()` (*dicee.knowledge_graph.KG method*), 46
`function()` (*dicee.models.FMult2 method*), 140
`function()` (*dicee.models.function_space.FMult2 method*), 78

G

`gamma` (*dicee.models.FMult attribute*), 139
`gamma` (*dicee.models.function_space.FMult attribute*), 77

gate_residual (*dicee.literal_classes.GatedLinearUnit* attribute), 51
 gate_residual (*dicee.literal_classes.LiteralEmbeddings* attribute), 52
 GatedLinearUnit (*class in dicee.literal_classes*), 51
 gelu (*dicee.models.transformers.MLP* attribute), 93
 gen_test (*dicee.query_generator.QueryGenerator* attribute), 143
 gen_test (*dicee.QueryGenerator* attribute), 211
 gen_valid (*dicee.query_generator.QueryGenerator* attribute), 143
 gen_valid (*dicee.QueryGenerator* attribute), 211
 generate () (*dicee.BytE* method), 188
 generate () (*dicee.KGE* method), 196
 generate () (*dicee.knowledge_graph_embeddings.KGE* method), 47
 generate () (*dicee.models.transformers.BytE* method), 90
 generate_queries () (*dicee.query_generator.QueryGenerator* method), 144
 generate_queries () (*dicee.QueryGenerator* method), 212
 get_aswa_state_dict () (*dicee.callbacks.ASWA* method), 23
 get_bpe_head_and_relation_representation () (*dicee.BaseKGE* method), 192
 get_bpe_head_and_relation_representation () (*dicee.models.base_model.BaseKGE* method), 64
 get_bpe_head_and_relation_representation () (*dicee.models.BaseKGE* method), 105, 108, 112, 116, 122, 135, 138
 get_bpe_token_representation () (*dicee.abstracts.BaseInteractiveKGE* method), 14
 get_callbacks () (*in module dicee.trainer.dice_trainer*), 160
 get_default_arguments () (*in module dicee.analyse_experiments*), 18
 get_default_arguments () (*in module dicee.scripts.index_serve*), 153
 get_default_arguments () (*in module dicee.scripts.run*), 155
 get_ee_vocab () (*in module dicee*), 192
 get_ee_vocab () (*in module dicee.read_preprocess_save_load_kg.util*), 149
 get_ee_vocab () (*in module dicee.static_funcs*), 156
 get_ee_vocab () (*in module dicee.static_preprocess_funcs*), 159
 get_embeddings () (*dicee.BaseKGE* method), 192
 get_embeddings () (*dicee.EnsembleKGE* method), 192
 get_embeddings () (*dicee.models.base_model.BaseKGE* method), 64
 get_embeddings () (*dicee.models.BaseKGE* method), 105, 108, 112, 116, 122, 135, 138
 get_embeddings () (*dicee.models.ensemble.EnsembleKGE* method), 76
 get_embeddings () (*dicee.models.real.Shallom* method), 88
 get_embeddings () (*dicee.models.Shallom* method), 109
 get_embeddings () (*dicee.Shallom* method), 185
 get_entity_embeddings () (*dicee.abstracts.BaseInteractiveKGE* method), 15
 get_entity_index () (*dicee.abstracts.BaseInteractiveKGE* method), 15
 get_er_vocab () (*in module dicee*), 192
 get_er_vocab () (*in module dicee.read_preprocess_save_load_kg.util*), 149
 get_er_vocab () (*in module dicee.static_funcs*), 156
 get_er_vocab () (*in module dicee.static_preprocess_funcs*), 159
 get_eval_report () (*dicee.abstracts.BaseInteractiveKGE* method), 14
 get_head_relation_representation () (*dicee.BaseKGE* method), 191
 get_head_relation_representation () (*dicee.models.base_model.BaseKGE* method), 64
 get_head_relation_representation () (*dicee.models.BaseKGE* method), 105, 108, 112, 116, 122, 135, 138
 get_kronecker_triple_representation () (*dicee.callbacks.KronE* method), 25
 get_num_params () (*dicee.models.transformers.GPT* method), 95
 get_padded_bpe_triple_representation () (*dicee.abstracts.BaseInteractiveKGE* method), 14
 get_queries () (*dicee.query_generator.QueryGenerator* method), 144
 get_queries () (*dicee.QueryGenerator* method), 212
 get_re_vocab () (*in module dicee*), 192
 get_re_vocab () (*in module dicee.read_preprocess_save_load_kg.util*), 149
 get_re_vocab () (*in module dicee.static_funcs*), 156
 get_re_vocab () (*in module dicee.static_preprocess_funcs*), 159
 get_relation_embeddings () (*dicee.abstracts.BaseInteractiveKGE* method), 15
 get_relation_index () (*dicee.abstracts.BaseInteractiveKGE* method), 15
 get_sentence_representation () (*dicee.BaseKGE* method), 191
 get_sentence_representation () (*dicee.models.base_model.BaseKGE* method), 64
 get_sentence_representation () (*dicee.models.BaseKGE* method), 105, 108, 112, 116, 122, 135, 138
 get_transductive_entity_embeddings () (*dicee.KGE* method), 195
 get_transductive_entity_embeddings () (*dicee.knowledge_graph_embeddings.KGE* method), 47
 get_triple_representation () (*dicee.BaseKGE* method), 191
 get_triple_representation () (*dicee.models.base_model.BaseKGE* method), 64
 get_triple_representation () (*dicee.models.BaseKGE* method), 105, 108, 112, 116, 122, 135, 138
 GFMult (*class in dicee.models*), 139
 GFMult (*class in dicee.models.function_space*), 77
 global_rank (*dicee.abstracts.AbstractTrainer* attribute), 12
 global_rank (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 164

GPT (class in *dicее.models.transformers*), 94
 GPTConfig (class in *dicее.models.transformers*), 94
 gpus (*dicее.config.Namespace* attribute), 27
 gradient_accumulation_steps (*dicее.config.Namespace* attribute), 27
 ground_queries () (*dicее.query_generator.QueryGenerator* method), 144
 ground_queries () (*dicее.QueryGenerator* method), 212

H

hidden_dim (*dicее.literal_classes.LiteralEmbeddings* attribute), 52
 hidden_dropout (*dicее.BaseKGE* attribute), 191
 hidden_dropout (*dicее.models.base_model.BaseKGE* attribute), 63
 hidden_dropout (*dicее.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137
 hidden_dropout_rate (*dicее.BaseKGE* attribute), 190
 hidden_dropout_rate (*dicее.config.Namespace* attribute), 28
 hidden_dropout_rate (*dicее.models.base_model.BaseKGE* attribute), 62
 hidden_dropout_rate (*dicее.models.BaseKGE* attribute), 104, 107, 110, 115, 121, 134, 137
 hidden_normalizer (*dicее.BaseKGE* attribute), 191
 hidden_normalizer (*dicее.models.base_model.BaseKGE* attribute), 63
 hidden_normalizer (*dicее.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137

I

IdentityClass (class in *dicее.models*), 105, 117, 122
 IdentityClass (class in *dicее.models.base_model*), 64
 idx_entity_to_bpe_shaped (*dicее.knowledge_graph.KG* attribute), 46
 index () (in module *dicее.scripts.index_serve*), 153
 index_triple () (*dicее.abstracts.BaseInteractiveKGE* method), 15
 init_data_loader () (*dicее.DICE_Trainer* method), 195
 init_data_loader () (*dicее.trainer.DICE_Trainer* method), 166
 init_data_loader () (*dicее.trainer.dice_trainer.DICE_Trainer* method), 161
 init_dataset () (*dicее.DICE_Trainer* method), 195
 init_dataset () (*dicее.trainer.DICE_Trainer* method), 166
 init_dataset () (*dicее.trainer.dice_trainer.DICE_Trainer* method), 161
 init_param (*dicее.config.Namespace* attribute), 27
 init_params_with_sanity_checking () (*dicее.BaseKGE* method), 191
 init_params_with_sanity_checking () (*dicее.models.base_model.BaseKGE* method), 63
 init_params_with_sanity_checking () (*dicее.models.BaseKGE* method), 104, 108, 111, 116, 122, 134, 138
 initial_eval_setting (*dicее.callbacks.ASWA* attribute), 23
 initialize_or_load_model () (*dicее.DICE_Trainer* method), 195
 initialize_or_load_model () (*dicее.trainer.DICE_Trainer* method), 166
 initialize_or_load_model () (*dicее.trainer.dice_trainer.DICE_Trainer* method), 161
 initialize_trainer () (*dicее.DICE_Trainer* method), 195
 initialize_trainer () (*dicее.trainer.DICE_Trainer* method), 165
 initialize_trainer () (*dicее.trainer.dice_trainer.DICE_Trainer* method), 161
 initialize_trainer () (in module *dicее.trainer.dice_trainer*), 160
 input_dp_ent_real (*dicее.BaseKGE* attribute), 191
 input_dp_ent_real (*dicее.models.base_model.BaseKGE* attribute), 63
 input_dp_ent_real (*dicее.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137
 input_dp_rel_real (*dicее.BaseKGE* attribute), 191
 input_dp_rel_real (*dicее.models.base_model.BaseKGE* attribute), 63
 input_dp_rel_real (*dicее.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137
 input_dropout_rate (*dicее.BaseKGE* attribute), 190
 input_dropout_rate (*dicее.config.Namespace* attribute), 28
 input_dropout_rate (*dicее.models.base_model.BaseKGE* attribute), 62
 input_dropout_rate (*dicее.models.BaseKGE* attribute), 104, 107, 110, 115, 121, 134, 137
 InteractiveQueryDecomposition (class in *dicее.abstracts*), 15
 initialize_model () (in module *dicее*), 193
 initialize_model () (in module *dicее.static_funcs*), 157
 is_continual_training (*dicее.DICE_Trainer* attribute), 194
 is_continual_training (*dicее.evaluator.Evaluator* attribute), 42
 is_continual_training (*dicее.executer.Execute* attribute), 43
 is_continual_training (*dicее.trainer.DICE_Trainer* attribute), 165
 is_continual_training (*dicее.trainer.dice_trainer.DICE_Trainer* attribute), 160
 is_global_zero (*dicее.abstracts.AbstractTrainer* attribute), 12
 is_seen () (*dicее.abstracts.BaseInteractiveKGE* method), 14
 is_sparql_endpoint_alive () (in module *dicее.sanity_checkers*), 151

K

`k` (*dicee.models.FMult* attribute), 139
`k` (*dicee.models.FMult2* attribute), 140
`k` (*dicee.models.function_space.FMult* attribute), 77
`k` (*dicee.models.function_space.FMult2* attribute), 78
`k` (*dicee.models.function_space.GFMult* attribute), 77
`k` (*dicee.models.GFMult* attribute), 139
`k_fold_cross_validation()` (*dicee.DICE_Trainer* method), 195
`k_fold_cross_validation()` (*dicee.trainer.DICE_Trainer* method), 166
`k_fold_cross_validation()` (*dicee.trainer.dice_trainer.DICE_Trainer* method), 161
`k_vs_all_score()` (*dicee.ComplEx* static method), 178
`k_vs_all_score()` (*dicee.DistMult* method), 169
`k_vs_all_score()` (*dicee.Keci* method), 172
`k_vs_all_score()` (*dicee.models.clifford.Keci* method), 67
`k_vs_all_score()` (*dicee.models.ComplEx* static method), 114
`k_vs_all_score()` (*dicee.models.complex.ComplEx* static method), 74
`k_vs_all_score()` (*dicee.models.DistMult* method), 108
`k_vs_all_score()` (*dicee.models.Keci* method), 128
`k_vs_all_score()` (*dicee.models.octonion.OMult* method), 81
`k_vs_all_score()` (*dicee.models.OMult* method), 124
`k_vs_all_score()` (*dicee.models.QMult* method), 119
`k_vs_all_score()` (*dicee.models.quaternion.QMult* method), 85
`k_vs_all_score()` (*dicee.models.real.DistMult* method), 87
`k_vs_all_score()` (*dicee.OMult* method), 185
`k_vs_all_score()` (*dicee.QMult* method), 184
Keci (class in *dicee*), 170
Keci (class in *dicee.models*), 126
Keci (class in *dicee.models.clifford*), 65
`kernel_size` (*dicee.BaseKGE* attribute), 190
`kernel_size` (*dicee.config.Namespace* attribute), 28
`kernel_size` (*dicee.models.base_model.BaseKGE* attribute), 62
`kernel_size` (*dicee.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137
KG (class in *dicee.knowledge_graph*), 45
`kg` (*dicee.callbacks.PseudoLabellingCallback* attribute), 22
`kg` (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk* attribute), 151
`kg` (*dicee.read_preprocess_save_load_kg.PreprocessKG* attribute), 150
`kg` (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* attribute), 144
`kg` (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk* attribute), 145
`kg` (*dicee.read_preprocess_save_load_kg.ReadFromDisk* attribute), 151
`kg` (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk* attribute), 146
KGE (class in *dicee*), 195
KGE (class in *dicee.knowledge_graph_embeddings*), 47
KGESaveCallback (class in *dicee.callbacks*), 21
`knowledge_graph` (*dicee.executer.Execute* attribute), 43
KronE (class in *dicee.callbacks*), 25
KvsAll (class in *dicee*), 202
KvsAll (class in *dicee.dataset_classes*), 31
`kvsall_score()` (*dicee.DualE* method), 177
`kvsall_score()` (*dicee.models.DualE* method), 142
`kvsall_score()` (*dicee.models.dualE.DualE* method), 75
KvsSampleDataset (class in *dicee*), 205
KvsSampleDataset (class in *dicee.dataset_classes*), 35

L

`label_smoothing_rate` (*dicee.AllvsAll* attribute), 204
`label_smoothing_rate` (*dicee.config.Namespace* attribute), 27
`label_smoothing_rate` (*dicee.dataset_classes.AllvsAll* attribute), 33
`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), 34
`label_smoothing_rate` (*dicee.dataset_classes.TriplePredictionDataset* attribute), 36
`label_smoothing_rate` (*dicee.KvsAll* attribute), 203
`label_smoothing_rate` (*dicee.KvsSampleDataset* attribute), 206
`label_smoothing_rate` (*dicee.OnevsSample* attribute), 204, 205
`label_smoothing_rate` (*dicee.TriplePredictionDataset* attribute), 207
`layer_norm` (*dicee.literal_classes.LiteralEmbeddings* attribute), 52
LayerNorm (class in *dicee.models.transformers*), 91

`learning_rate` (*dicee.BaseKGE attribute*), 190
`learning_rate` (*dicee.models.base_model.BaseKGE attribute*), 62
`learning_rate` (*dicee.models.BaseKGE attribute*), 104, 107, 110, 115, 121, 134, 137
`length` (*dicee.dataset_classes.NegSampleDataset attribute*), 36
`length` (*dicee.dataset_classes.TriplePredictionDataset attribute*), 37
`length` (*dicee.NegSampleDataset attribute*), 206
`length` (*dicee.TriplePredictionDataset attribute*), 207
`level` (*dicee.callbacks.Perturb attribute*), 25
`LFMult` (*class in dicee*), 185
`LFMult` (*class in dicee.models*), 140
`LFMult` (*class in dicee.models.function_space*), 78
`LFMult1` (*class in dicee.models*), 140
`LFMult1` (*class in dicee.models.function_space*), 78
`linear()` (*dicee.LFMult method*), 186
`linear()` (*dicee.models.function_space.LFMult method*), 79
`linear()` (*dicee.models.LFMult method*), 141
`list2tuple()` (*dicee.query_generator.QueryGenerator method*), 143
`list2tuple()` (*dicee.QueryGenerator method*), 211
`LiteralDataset` (*class in dicee.literal_classes*), 53
`LiteralEmbeddings` (*class in dicee.literal_classes*), 51
`lm_head` (*dicee.BytE attribute*), 188
`lm_head` (*dicee.models.transformers.BytE attribute*), 90
`lm_head` (*dicee.models.transformers.GPT attribute*), 95
`ln_1` (*dicee.models.transformers.Block attribute*), 94
`ln_2` (*dicee.models.transformers.Block attribute*), 94
`load()` (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk method*), 151
`load()` (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk method*), 146
`load_and_validate_literal_data()` (*dicee.literal_classes.LiteralDataset static method*), 54
`load_json()` (*in module dicee*), 193
`load_json()` (*in module dicee.static_funcs*), 157
`load_model()` (*in module dicee*), 193
`load_model()` (*in module dicee.static_funcs*), 156
`load_model_ensemble()` (*in module dicee*), 193
`load_model_ensemble()` (*in module dicee.static_funcs*), 156
`load_numpy()` (*in module dicee*), 194
`load_numpy()` (*in module dicee.static_funcs*), 157
`load_numpy_ndarray()` (*in module dicee.read_preprocess_save_load_kg.util*), 149
`load_pickle()` (*in module dicee*), 192
`load_pickle()` (*in module dicee.read_preprocess_save_load_kg.util*), 150
`load_pickle()` (*in module dicee.static_funcs*), 156
`load_queries()` (*dicee.query_generator.QueryGenerator method*), 144
`load_queries()` (*dicee.QueryGenerator method*), 212
`load_queries_and_answers()` (*dicee.query_generator.QueryGenerator static method*), 144
`load_queries_and_answers()` (*dicee.QueryGenerator static method*), 212
`load_term_mapping()` (*in module dicee*), 193, 200
`load_term_mapping()` (*in module dicee.static_funcs*), 156
`load_term_mapping()` (*in module dicee.trainer.dice_trainer*), 160
`load_with_pandas()` (*in module dicee.read_preprocess_save_load_kg.util*), 149
`loader_backend` (*dicee.literal_classes.LiteralDataset attribute*), 54
`LoadSaveToDisk` (*class in dicee.read_preprocess_save_load_kg*), 151
`LoadSaveToDisk` (*class in dicee.read_preprocess_save_load_kg.save_load_disk*), 146
`local_rank` (*dicee.abstracts.AbstractTrainer attribute*), 12
`local_rank` (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 164
`loss` (*dicee.BaseKGE attribute*), 190
`loss` (*dicee.models.base_model.BaseKGE attribute*), 63
`loss` (*dicee.models.BaseKGE attribute*), 104, 107, 111, 115, 121, 134, 137
`loss_func` (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 164
`loss_function` (*dicee.trainer.torch_trainer.TorchTrainer attribute*), 162
`loss_function()` (*dicee.BytE method*), 188
`loss_function()` (*dicee.models.base_model.BaseKGE Lightning method*), 58
`loss_function()` (*dicee.models.BaseKGE Lightning method*), 99
`loss_function()` (*dicee.models.transformers.BytE method*), 90
`loss_history` (*dicee.BaseKGE attribute*), 191
`loss_history` (*dicee.models.base_model.BaseKGE attribute*), 63
`loss_history` (*dicee.models.BaseKGE attribute*), 104, 107, 111, 116, 121, 134, 138
`loss_history` (*dicee.models.pykeen_models.PykeenKGE attribute*), 83
`loss_history` (*dicee.models.PykeenKGE attribute*), 135
`loss_history` (*dicee.PykeenKGE attribute*), 187

loss_history (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 164
 lr (dicee.analyse_experiments.Experiment attribute), 18
 lr (dicee.config.Namespace attribute), 26

M

m (dicee.LFMult attribute), 185
 m (dicee.models.function_space.LFMult attribute), 79
 m (dicee.models.LFMult attribute), 141
 main() (in module dicee.scripts.index_serve), 154
 main() (in module dicee.scripts.run), 155
 make_iterable_verbose() (in module dicee.static_funcs_training), 158
 make_iterable_verbose() (in module dicee.trainer.torch_trainer_ddp), 164
 mapping_from_first_two_cols_to_third() (in module dicee), 200
 mapping_from_first_two_cols_to_third() (in module dicee.static_preprocess_funcs), 159
 margin (dicee.models.Pyke attribute), 109
 margin (dicee.models.real.Pyke attribute), 88
 margin (dicee.models.real.TransE attribute), 88
 margin (dicee.models.TransE attribute), 109
 margin (dicee.Pyke attribute), 169
 margin (dicee.TransE attribute), 173
 max_ans_num (dicee.query_generator.QueryGenerator attribute), 143
 max_ans_num (dicee.QueryGenerator attribute), 211
 max_epochs (dicee.callbacks.KGESaveCallback attribute), 21
 max_length_subword_tokens (dicee.BaseKGE attribute), 191
 max_length_subword_tokens (dicee.knowledge_graph.KG attribute), 46
 max_length_subword_tokens (dicee.models.base_model.BaseKGE attribute), 63
 max_length_subword_tokens (dicee.models.BaseKGE attribute), 104, 107, 111, 116, 122, 134, 138
 max_num_of_classes (dicee.dataset_classes.KvsSampleDataset attribute), 35
 max_num_of_classes (dicee.KvsSampleDataset attribute), 206
 mem_of_model() (dicee.EnsembleKGE method), 192
 mem_of_model() (dicee.models.base_model.BaseKGE Lightning method), 57
 mem_of_model() (dicee.models.BaseKGE Lightning method), 98
 mem_of_model() (dicee.models.ensemble.EnsembleKGE method), 76
 method (dicee.callbacks.Perturb attribute), 25
 MLP (class in dicee.models.transformers), 92
 mlp (dicee.models.transformers.Block attribute), 94
 mode (dicee.query_generator.QueryGenerator attribute), 143
 mode (dicee.QueryGenerator attribute), 211
 model (dicee.config.Namespace attribute), 26
 model (dicee.models.pykeen_models.PykeenKGE attribute), 83
 model (dicee.models.PykeenKGE attribute), 135
 model (dicee.PykeenKGE attribute), 187
 model (dicee.trainer.torch_trainer_ddp.NodeTrainer attribute), 164
 model (dicee.trainer.torch_trainer.TorchTrainer attribute), 163
 model_kwargs (dicee.models.pykeen_models.PykeenKGE attribute), 83
 model_kwargs (dicee.models.PykeenKGE attribute), 135
 model_kwargs (dicee.PykeenKGE attribute), 186
 model_name (dicee.analyse_experiments.Experiment attribute), 18
 module
 dicee, 12
 dicee.__main__, 12
 dicee.abstracts, 12
 dicee.analyse_experiments, 18
 dicee.callbacks, 19
 dicee.config, 26
 dicee.dataset_classes, 29
 dicee.eval_static_funcs, 40
 dicee.evaluator, 42
 dicee.executer, 43
 dicee.knowledge_graph, 45
 dicee.knowledge_graph_embeddings, 46
 dicee.literal_classes, 51
 dicee.models, 55
 dicee.models.adopt, 55
 dicee.models.base_model, 56
 dicee.models.clifford, 65
 dicee.models.complex, 72

- dicee.models.dualE, 74
- dicee.models.ensemble, 76
- dicee.models.function_space, 76
- dicee.models.octonion, 80
- dicee.models.pykeen_models, 83
- dicee.models.quaternion, 84
- dicee.models.real, 87
- dicee.models.static_funcs, 89
- dicee.models.transformers, 89
- dicee.query_generator, 143
- dicee.read_preprocess_save_load_kg, 144
- dicee.read_preprocess_save_load_kg.preprocess, 144
- dicee.read_preprocess_save_load_kg.read_from_disk, 145
- dicee.read_preprocess_save_load_kg.save_load_disk, 146
- dicee.read_preprocess_save_load_kg.util, 146
- dicee.sanity_checkers, 151
- dicee.scripts, 152
- dicee.scripts.index_serve, 152
- dicee.scripts.run, 154
- dicee.static_funcs, 155
- dicee.static_funcs_training, 158
- dicee.static_preprocess_funcs, 158
- dicee.trainer, 159
- dicee.trainer.dice_trainer, 159
- dicee.trainer.model_parallelism, 161
- dicee.trainer.torch_trainer, 162
- dicee.trainer.torch_trainer_ddp, 163
- modules() (dicee.EnsembleKGE method), 192
- modules() (dicee.models.ensemble.EnsembleKGE method), 76
- MultiClassClassificationDataset (class in dicee), 201
- MultiClassClassificationDataset (class in dicee.dataset_classes), 30
- MultiLabelDataset (class in dicee), 201
- MultiLabelDataset (class in dicee.dataset_classes), 30

N

- n (dicee.models.FMult2 attribute), 140
- n (dicee.models.function_space.FMult2 attribute), 78
- n_embd (dicee.models.transformers.CausalSelfAttention attribute), 92
- n_embd (dicee.models.transformers.GPTConfig attribute), 94
- n_head (dicee.models.transformers.CausalSelfAttention attribute), 92
- n_head (dicee.models.transformers.GPTConfig attribute), 94
- n_layer (dicee.models.transformers.GPTConfig attribute), 94
- n_layers (dicee.models.FMult2 attribute), 140
- n_layers (dicee.models.function_space.FMult2 attribute), 78
- name (dicee.abstracts.BaseInteractiveKGE property), 14
- name (dicee.AConEx attribute), 178
- name (dicee.AConvO attribute), 179
- name (dicee.AConvQ attribute), 180
- name (dicee.BytE attribute), 188
- name (dicee.CKeci attribute), 170
- name (dicee.ComplEx attribute), 178
- name (dicee.ConEx attribute), 182
- name (dicee.ConvO attribute), 181
- name (dicee.ConvQ attribute), 180
- name (dicee.DeCaL attribute), 174
- name (dicee.DistMult attribute), 169
- name (dicee.DualE attribute), 177
- name (dicee.EnsembleKGE attribute), 192
- name (dicee.Keci attribute), 170
- name (dicee.LFMult attribute), 185
- name (dicee.models.AConEx attribute), 113
- name (dicee.models.AConvO attribute), 125
- name (dicee.models.AConvQ attribute), 119
- name (dicee.models.CKeci attribute), 129
- name (dicee.models.clifford.CKeci attribute), 68
- name (dicee.models.clifford.DeCaL attribute), 69
- name (dicee.models.clifford.Keci attribute), 66

name (*dicdee.models.ComplEx* attribute), 114
 name (*dicdee.models.complex.AConEx* attribute), 73
 name (*dicdee.models.complex.ComplEx* attribute), 74
 name (*dicdee.models.complex.ConEx* attribute), 72
 name (*dicdee.models.ConEx* attribute), 112
 name (*dicdee.models.ConvO* attribute), 125
 name (*dicdee.models.ConvQ* attribute), 119
 name (*dicdee.models.DeCaL* attribute), 130
 name (*dicdee.models.DistMult* attribute), 108
 name (*dicdee.models.DualE* attribute), 142
 name (*dicdee.models.dualE.DualE* attribute), 75
 name (*dicdee.models.ensemble.EnsembleKGE* attribute), 76
 name (*dicdee.models.FMult* attribute), 139
 name (*dicdee.models.FMult2* attribute), 140
 name (*dicdee.models.function_space.FMult* attribute), 77
 name (*dicdee.models.function_space.FMult2* attribute), 78
 name (*dicdee.models.function_space.GFMult* attribute), 77
 name (*dicdee.models.function_space.LFMult* attribute), 79
 name (*dicdee.models.function_space.LFMult1* attribute), 78
 name (*dicdee.models.GFMult* attribute), 139
 name (*dicdee.models.Keci* attribute), 126
 name (*dicdee.models.LFMult* attribute), 141
 name (*dicdee.models.LFMult1* attribute), 140
 name (*dicdee.models.octonion.AConvO* attribute), 82
 name (*dicdee.models.octonion.ConvO* attribute), 82
 name (*dicdee.models.octonion.OMult* attribute), 81
 name (*dicdee.models.OMult* attribute), 124
 name (*dicdee.models.Pyke* attribute), 109
 name (*dicdee.models.pykeen_models.PykeenKGE* attribute), 83
 name (*dicdee.models.PykeenKGE* attribute), 135
 name (*dicdee.models.QMult* attribute), 118
 name (*dicdee.models.quaternion.AConvQ* attribute), 86
 name (*dicdee.models.quaternion.ConvQ* attribute), 86
 name (*dicdee.models.quaternion.QMult* attribute), 85
 name (*dicdee.models.real.DistMult* attribute), 87
 name (*dicdee.models.real.Pyke* attribute), 88
 name (*dicdee.models.real.Shallom* attribute), 88
 name (*dicdee.models.real.TransE* attribute), 88
 name (*dicdee.models.Shallom* attribute), 109
 name (*dicdee.models.TransE* attribute), 109
 name (*dicdee.models.transformers.Byte* attribute), 90
 name (*dicdee.OMult* attribute), 185
 name (*dicdee.Pyke* attribute), 169
 name (*dicdee.PykeenKGE* attribute), 186
 name (*dicdee.QMult* attribute), 183
 name (*dicdee.Shallom* attribute), 185
 name (*dicdee.TransE* attribute), 173
 named_children() (*dicdee.EnsembleKGE* method), 192
 named_children() (*dicdee.models.ensemble.EnsembleKGE* method), 76
 Namespace (class in *dicdee.config*), 26
 neg_ratio (*dicdee.BPE_NegativeSamplingDataset* attribute), 200
 neg_ratio (*dicdee.config.Namespace* attribute), 27
 neg_ratio (*dicdee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 30
 neg_ratio (*dicdee.dataset_classes.KvsSampleDataset* attribute), 35
 neg_ratio (*dicdee.KvsSampleDataset* attribute), 206
 neg_sample_ratio (*dicdee.CVDDataModule* attribute), 208
 neg_sample_ratio (*dicdee.dataset_classes.CVDDataModule* attribute), 37
 neg_sample_ratio (*dicdee.dataset_classes.NegSampleDataset* attribute), 36
 neg_sample_ratio (*dicdee.dataset_classes.OnevsSample* attribute), 34
 neg_sample_ratio (*dicdee.dataset_classes.TriplePredictionDataset* attribute), 37
 neg_sample_ratio (*dicdee.NegSampleDataset* attribute), 206
 neg_sample_ratio (*dicdee.OnevsSample* attribute), 204, 205
 neg_sample_ratio (*dicdee.TriplePredictionDataset* attribute), 207
 negnorm() (*dicdee.abstracts.InteractiveQueryDecomposition* method), 16
 NegSampleDataset (class in *dicdee*), 206
 NegSampleDataset (class in *dicdee.dataset_classes*), 35
 neural_searcher (in module *dicdee.scripts.index_serve*), 153
 NeuralSearcher (class in *dicdee.scripts.index_serve*), 153

NodeTrainer (class in *dicee.trainer.torch_trainer_ddp*), 164
norm_fc1 (*dicee.AConEx* attribute), 179
norm_fc1 (*dicee.AConvO* attribute), 179
norm_fc1 (*dicee.ConEx* attribute), 182
norm_fc1 (*dicee.ConvO* attribute), 182
norm_fc1 (*dicee.models.AConEx* attribute), 113
norm_fc1 (*dicee.models.AConvO* attribute), 126
norm_fc1 (*dicee.models.complex.AConEx* attribute), 73
norm_fc1 (*dicee.models.complex.ConEx* attribute), 72
norm_fc1 (*dicee.models.ConEx* attribute), 112
norm_fc1 (*dicee.models.ConvO* attribute), 125
norm_fc1 (*dicee.models.octonion.AConvO* attribute), 82
norm_fc1 (*dicee.models.octonion.ConvO* attribute), 82
normalization (*dicee.analyse_experiments.Experiment* attribute), 19
normalization (*dicee.config.Namespace* attribute), 27
normalization (*dicee.literal_classes.LiteralDataset* attribute), 53
normalization_params (*dicee.literal_classes.LiteralDataset* attribute), 53, 54
normalization_type (*dicee.literal_classes.LiteralDataset* attribute), 54
normalize_head_entity_embeddings (*dicee.BaseKGE* attribute), 191
normalize_head_entity_embeddings (*dicee.models.base_model.BaseKGE* attribute), 63
normalize_head_entity_embeddings (*dicee.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137
normalize_relation_embeddings (*dicee.BaseKGE* attribute), 191
normalize_relation_embeddings (*dicee.models.base_model.BaseKGE* attribute), 63
normalize_relation_embeddings (*dicee.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137
normalize_tail_entity_embeddings (*dicee.BaseKGE* attribute), 191
normalize_tail_entity_embeddings (*dicee.models.base_model.BaseKGE* attribute), 63
normalize_tail_entity_embeddings (*dicee.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137
normalizer_class (*dicee.BaseKGE* attribute), 190
normalizer_class (*dicee.models.base_model.BaseKGE* attribute), 63
normalizer_class (*dicee.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137
num_bpe_entities (*dicee.BPE_NegativeSamplingDataset* attribute), 200
num_bpe_entities (*dicee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 30
num_bpe_entities (*dicee.knowledge_graph.KG* attribute), 46
num_core (*dicee.config.Namespace* attribute), 27
num_data_properties (*dicee.literal_classes.LiteralDataset* attribute), 54
num_datapoints (*dicee.BPE_NegativeSamplingDataset* attribute), 200
num_datapoints (*dicee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 30
num_datapoints (*dicee.dataset_classes.MultiLabelDataset* attribute), 30
num_datapoints (*dicee.MultiLabelDataset* attribute), 201
num_ent (*dicee.DualE* attribute), 177
num_ent (*dicee.models.DualE* attribute), 142
num_ent (*dicee.models.dualE.DualE* attribute), 75
num_entities (*dicee.BaseKGE* attribute), 190
num_entities (*dicee.CVDataModule* attribute), 208
num_entities (*dicee.dataset_classes.CVDataModule* attribute), 37
num_entities (*dicee.dataset_classes.KvsSampleDataset* attribute), 35
num_entities (*dicee.dataset_classes.NegSampleDataset* attribute), 36
num_entities (*dicee.dataset_classes.OnevsSample* attribute), 33, 34
num_entities (*dicee.dataset_classes.TriplePredictionDataset* attribute), 37
num_entities (*dicee.evaluator.Evaluator* attribute), 42
num_entities (*dicee.knowledge_graph.KG* attribute), 45
num_entities (*dicee.KvsSampleDataset* attribute), 206
num_entities (*dicee.literal_classes.LiteralDataset* attribute), 53, 54
num_entities (*dicee.models.base_model.BaseKGE* attribute), 62
num_entities (*dicee.models.BaseKGE* attribute), 103, 107, 110, 115, 121, 133, 137
num_entities (*dicee.NegSampleDataset* attribute), 206
num_entities (*dicee.OnevsSample* attribute), 204, 205
num_entities (*dicee.TriplePredictionDataset* attribute), 207
num_epochs (*dicee.abstracts.AbstractPPECallback* attribute), 17
num_epochs (*dicee.analyse_experiments.Experiment* attribute), 18
num_epochs (*dicee.callbacks.ASWA* attribute), 23
num_epochs (*dicee.config.Namespace* attribute), 26
num_epochs (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 164
num_folds_for_cv (*dicee.config.Namespace* attribute), 27
num_of_data_points (*dicee.dataset_classes.MultiClassClassificationDataset* attribute), 31
num_of_data_points (*dicee.MultiClassClassificationDataset* attribute), 202
num_of_data_properties (*dicee.literal_classes.LiteralEmbeddings* attribute), 52
num_of_epochs (*dicee.callbacks.PseudoLabellingCallback* attribute), 22

- `num_of_output_channels` (*dicee.BaseKGE attribute*), 190
- `num_of_output_channels` (*dicee.config.Namespace attribute*), 28
- `num_of_output_channels` (*dicee.models.base_model.BaseKGE attribute*), 63
- `num_of_output_channels` (*dicee.models.BaseKGE attribute*), 104, 107, 111, 115, 121, 134, 137
- `num_params` (*dicee.analyse_experiments.Experiment attribute*), 18
- `num_relations` (*dicee.BaseKGE attribute*), 190
- `num_relations` (*dicee.CVDDataModule attribute*), 208
- `num_relations` (*dicee.dataset_classes.CVDDataModule attribute*), 37
- `num_relations` (*dicee.dataset_classes.NegSampleDataset attribute*), 36
- `num_relations` (*dicee.dataset_classes.OnevsSample attribute*), 34
- `num_relations` (*dicee.dataset_classes.TriplePredictionDataset attribute*), 37
- `num_relations` (*dicee.evaluator.Evaluator attribute*), 42
- `num_relations` (*dicee.knowledge_graph.KG attribute*), 45
- `num_relations` (*dicee.models.base_model.BaseKGE attribute*), 62
- `num_relations` (*dicee.models.BaseKGE attribute*), 103, 107, 110, 115, 121, 133, 137
- `num_relations` (*dicee.NegSampleDataset attribute*), 206
- `num_relations` (*dicee.OnevsSample attribute*), 204, 205
- `num_relations` (*dicee.TriplePredictionDataset attribute*), 207
- `num_sample` (*dicee.models.FMult attribute*), 139
- `num_sample` (*dicee.models.function_space.FMult attribute*), 77
- `num_sample` (*dicee.models.function_space.GFMult attribute*), 77
- `num_sample` (*dicee.models.GFMult attribute*), 139
- `num_tokens` (*dicee.BaseKGE attribute*), 190
- `num_tokens` (*dicee.knowledge_graph.KG attribute*), 46
- `num_tokens` (*dicee.models.base_model.BaseKGE attribute*), 62
- `num_tokens` (*dicee.models.BaseKGE attribute*), 103, 107, 110, 115, 121, 134, 137
- `num_workers` (*dicee.CVDDataModule attribute*), 208
- `num_workers` (*dicee.dataset_classes.CVDDataModule attribute*), 37
- `numpy_data_type_changer()` (in module *dicee*), 193
- `numpy_data_type_changer()` (in module *dicee.static_funcs*), 157

O

- `octonion_mul()` (in module *dicee.models*), 123
- `octonion_mul()` (in module *dicee.models.octonion*), 80
- `octonion_mul_norm()` (in module *dicee.models*), 123
- `octonion_mul_norm()` (in module *dicee.models.octonion*), 80
- `octonion_normalizer()` (*dicee.AConvO static method*), 179
- `octonion_normalizer()` (*dicee.ConvO static method*), 182
- `octonion_normalizer()` (*dicee.models.AConvO static method*), 126
- `octonion_normalizer()` (*dicee.models.ConvO static method*), 125
- `octonion_normalizer()` (*dicee.models.octonion.AConvO static method*), 82
- `octonion_normalizer()` (*dicee.models.octonion.ConvO static method*), 82
- `octonion_normalizer()` (*dicee.models.octonion.OMult static method*), 81
- `octonion_normalizer()` (*dicee.models.OMult static method*), 124
- `octonion_normalizer()` (*dicee.OMult static method*), 185
- `OMult` (class in *dicee*), 184
- `OMult` (class in *dicee.models*), 123
- `OMult` (class in *dicee.models.octonion*), 80
- `on_epoch_end()` (*dicee.callbacks.KGESaveCallback method*), 22
- `on_epoch_end()` (*dicee.callbacks.PseudoLabellingCallback method*), 22
- `on_fit_end()` (*dicee.abstracts.AbstractCallback method*), 17
- `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*), 13
- `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*), 16

- `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*), 21
- `on_train_batch_start()` (*dicee.callbacks.Perturb method*), 26
- `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*), 22
- `on_train_epoch_end()` (*dicee.callbacks.PrintCallback method*), 21
- `on_train_epoch_end()` (*dicee.models.base_model.BaseKGELightning method*), 58
- `on_train_epoch_end()` (*dicee.models.BaseKGELightning method*), 99
- `OnevsAllDataset` (*class in dicee*), 202
- `OnevsAllDataset` (*class in dicee.dataset_classes*), 31
- `OnevsSample` (*class in dicee*), 204
- `OnevsSample` (*class in dicee.dataset_classes*), 33
- `optim` (*dicee.config.Namespace attribute*), 26
- `optimizer` (*dicee.trainer.torch_trainer_ddp.NodeTrainer attribute*), 164
- `optimizer` (*dicee.trainer.torch_trainer.TorchTrainer attribute*), 162
- `optimizer_name` (*dicee.BaseKGE attribute*), 190
- `optimizer_name` (*dicee.models.base_model.BaseKGE attribute*), 62
- `optimizer_name` (*dicee.models.BaseKGE attribute*), 104, 107, 110, 115, 121, 134, 137
- `ordered_bpe_entities` (*dicee.BPE_NegativeSamplingDataset attribute*), 200
- `ordered_bpe_entities` (*dicee.dataset_classes.BPE_NegativeSamplingDataset attribute*), 30
- `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*), 174
- `p` (*dicee.Keci attribute*), 170
- `p` (*dicee.models.clifford.DeCaL attribute*), 69
- `p` (*dicee.models.clifford.Keci attribute*), 66
- `p` (*dicee.models.DeCaL attribute*), 130
- `p` (*dicee.models.Keci attribute*), 127
- `padding` (*dicee.knowledge_graph.KG attribute*), 46
- `pandas_dataframe_indexer()` (*in module dicee.read_preprocess_save_load_kg.util*), 148
- `param_init` (*dicee.BaseKGE attribute*), 191
- `param_init` (*dicee.models.base_model.BaseKGE attribute*), 63
- `param_init` (*dicee.models.BaseKGE attribute*), 104, 107, 111, 115, 121, 134, 137
- `parameters()` (*dicee.abstracts.BaseInteractiveKGE method*), 15
- `parameters()` (*dicee.EnsembleKGE method*), 192
- `parameters()` (*dicee.models.ensemble.EnsembleKGE method*), 76
- `path` (*dicee.abstracts.AbstractPPECallback attribute*), 17
- `path` (*dicee.callbacks.AccumulateEpochLossCallback attribute*), 20
- `path` (*dicee.callbacks.ASWA attribute*), 23
- `path` (*dicee.callbacks.Eval attribute*), 24
- `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*), 147
- `poly_NN()` (*dicee.LFMMult method*), 185
- `poly_NN()` (*dicee.models.function_space.LFMMult method*), 79
- `poly_NN()` (*dicee.models.LFMMult method*), 141
- `polynomial()` (*dicee.LFMMult method*), 186
- `polynomial()` (*dicee.models.function_space.LFMMult method*), 79
- `polynomial()` (*dicee.models.LFMMult method*), 141
- `pop()` (*dicee.LFMMult method*), 186
- `pop()` (*dicee.models.function_space.LFMMult method*), 80
- `pop()` (*dicee.models.LFMMult method*), 142

pq (*dicee.analyse_experiments.Experiment* attribute), 18
 predict () (*dicee.KGE* method), 197
 predict () (*dicee.knowledge_graph_embeddings.KGE* method), 48
 predict_data_loader () (*dicee.models.base_model.BaseKGELightning* method), 59
 predict_data_loader () (*dicee.models.BaseKGELightning* method), 101
 predict_literals () (*dicee.KGE* method), 199
 predict_literals () (*dicee.knowledge_graph_embeddings.KGE* method), 50
 predict_missing_head_entity () (*dicee.KGE* method), 196
 predict_missing_head_entity () (*dicee.knowledge_graph_embeddings.KGE* method), 47
 predict_missing_relations () (*dicee.KGE* method), 196
 predict_missing_relations () (*dicee.knowledge_graph_embeddings.KGE* method), 47
 predict_missing_tail_entity () (*dicee.KGE* method), 196
 predict_missing_tail_entity () (*dicee.knowledge_graph_embeddings.KGE* method), 48
 predict_topk () (*dicee.KGE* method), 197
 predict_topk () (*dicee.knowledge_graph_embeddings.KGE* method), 48
 prepare_data () (*dicee.CVDataModule* method), 210
 prepare_data () (*dicee.dataset_classes.CVDataModule* method), 39
 preprocess_with_byte_pair_encoding () (*dicee.read_preprocess_save_load_kg.PreprocessKG* method), 150
 preprocess_with_byte_pair_encoding () (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 145
 preprocess_with_byte_pair_encoding_with_padding () (*dicee.read_preprocess_save_load_kg.PreprocessKG* method), 150
 preprocess_with_byte_pair_encoding_with_padding () (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 145
 preprocess_with_pandas () (*dicee.read_preprocess_save_load_kg.PreprocessKG* method), 150
 preprocess_with_pandas () (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 145
 preprocess_with_polars () (*dicee.read_preprocess_save_load_kg.PreprocessKG* method), 151
 preprocess_with_polars () (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 145
 preprocesses_input_args () (in module *dicee.static_preprocess_funcs*), 159
 PreprocessKG (class in *dicee.read_preprocess_save_load_kg*), 150
 PreprocessKG (class in *dicee.read_preprocess_save_load_kg.preprocess*), 144
 PrintCallback (class in *dicee.callbacks*), 20
 process (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 163
 proj (*dicee.literal_classes.GatedLinearUnit* attribute), 51
 PseudoLabellingCallback (class in *dicee.callbacks*), 22
 Pyke (class in *dicee*), 169
 Pyke (class in *dicee.models*), 109
 Pyke (class in *dicee.models.real*), 88
 pykeen_model_kwargs (*dicee.config.Namespace* attribute), 28
 PykeenKGE (class in *dicee*), 186
 PykeenKGE (class in *dicee.models*), 135
 PykeenKGE (class in *dicee.models.pykeen_models*), 83

Q

q (*dicee.config.Namespace* attribute), 28
 q (*dicee.DeCaL* attribute), 174
 q (*dicee.Keci* attribute), 170
 q (*dicee.models.clifford.DeCaL* attribute), 69
 q (*dicee.models.clifford.Keci* attribute), 66
 q (*dicee.models.DeCaL* attribute), 130
 q (*dicee.models.Keci* attribute), 127
 qdrant_client (*dicee.scripts.index_serve.NeuralSearcher* attribute), 153
 QMult (class in *dicee*), 182
 QMult (class in *dicee.models*), 117
 QMult (class in *dicee.models.quaternion*), 84
 quaternion_mul () (in module *dicee.models*), 114
 quaternion_mul () (in module *dicee.models.static_funcs*), 89
 quaternion_mul_with_unit_norm () (in module *dicee.models*), 117
 quaternion_mul_with_unit_norm () (in module *dicee.models.quaternion*), 84
 quaternion_multiplication_followed_by_inner_product () (*dicee.models.QMult* method), 118
 quaternion_multiplication_followed_by_inner_product () (*dicee.models.quaternion.QMult* method), 85
 quaternion_multiplication_followed_by_inner_product () (*dicee.QMult* method), 183
 quaternion_normalizer () (*dicee.models.QMult* static method), 118
 quaternion_normalizer () (*dicee.models.quaternion.QMult* static method), 85
 quaternion_normalizer () (*dicee.QMult* static method), 183
 queries (*dicee.scripts.index_serve.StringListRequest* attribute), 154
 query_name_to_struct (*dicee.query_generator.QueryGenerator* attribute), 143
 query_name_to_struct (*dicee.QueryGenerator* attribute), 211
 QueryGenerator (class in *dicee*), 211
 QueryGenerator (class in *dicee.query_generator*), 143

R

- `r` (*dicee.DeCaL* attribute), 174
- `r` (*dicee.Keci* attribute), 170
- `r` (*dicee.models.clifford.DeCaL* attribute), 69
- `r` (*dicee.models.clifford.Keci* attribute), 66
- `r` (*dicee.models.DeCaL* attribute), 130
- `r` (*dicee.models.Keci* attribute), 127
- `random_prediction()` (in module *dicee*), 193
- `random_prediction()` (in module *dicee.static_funcs*), 157
- `random_seed` (*dicee.config.Namespace* attribute), 27
- `ratio` (*dicee.callbacks.Perturb* attribute), 25
- `re` (*dicee.DeCaL* attribute), 174
- `re` (*dicee.models.clifford.DeCaL* attribute), 69
- `re` (*dicee.models.DeCaL* attribute), 130
- `re_vocab` (*dicee.evaluator.Evaluator* attribute), 42
- `read_from_disk()` (in module *dicee.read_preprocess_save_load_kg.util*), 149
- `read_from_triple_store()` (in module *dicee.read_preprocess_save_load_kg.util*), 149
- `read_only_few` (*dicee.config.Namespace* attribute), 28
- `read_only_few` (*dicee.knowledge_graph.KG* attribute), 45
- `read_or_load_kg()` (in module *dicee*), 193
- `read_or_load_kg()` (in module *dicee.static_funcs*), 157
- `read_with_pandas()` (in module *dicee.read_preprocess_save_load_kg.util*), 149
- `read_with_polars()` (in module *dicee.read_preprocess_save_load_kg.util*), 149
- `ReadFromDisk` (class in *dicee.read_preprocess_save_load_kg*), 151
- `ReadFromDisk` (class in *dicee.read_preprocess_save_load_kg.read_from_disk*), 145
- `reducer` (*dicee.scripts.index_serve.StringListRequest* attribute), 154
- `rel2id` (*dicee.query_generator.QueryGenerator* attribute), 143
- `rel2id` (*dicee.QueryGenerator* attribute), 211
- `relation_embeddings` (*dicee.AConvQ* attribute), 180
- `relation_embeddings` (*dicee.ConvQ* attribute), 180
- `relation_embeddings` (*dicee.DeCaL* attribute), 174
- `relation_embeddings` (*dicee.DualE* attribute), 177
- `relation_embeddings` (*dicee.LFMult* attribute), 185
- `relation_embeddings` (*dicee.models.AConvQ* attribute), 120
- `relation_embeddings` (*dicee.models.clifford.DeCaL* attribute), 69
- `relation_embeddings` (*dicee.models.ConvQ* attribute), 119
- `relation_embeddings` (*dicee.models.DeCaL* attribute), 130
- `relation_embeddings` (*dicee.models.DualE* attribute), 142
- `relation_embeddings` (*dicee.models.dualE.DualE* attribute), 75
- `relation_embeddings` (*dicee.models.FMult* attribute), 139
- `relation_embeddings` (*dicee.models.FMult2* attribute), 140
- `relation_embeddings` (*dicee.models.function_space.FMult* attribute), 77
- `relation_embeddings` (*dicee.models.function_space.FMult2* attribute), 78
- `relation_embeddings` (*dicee.models.function_space.GFMult* attribute), 77
- `relation_embeddings` (*dicee.models.function_space.LFMult* attribute), 79
- `relation_embeddings` (*dicee.models.function_space.LFMult1* attribute), 78
- `relation_embeddings` (*dicee.models.GFMult* attribute), 139
- `relation_embeddings` (*dicee.models.LFMult* attribute), 141
- `relation_embeddings` (*dicee.models.LFMult1* attribute), 140
- `relation_embeddings` (*dicee.models.pykeen_models.PykeenKGE* attribute), 83
- `relation_embeddings` (*dicee.models.PykeenKGE* attribute), 136
- `relation_embeddings` (*dicee.models.quaternion.AConvQ* attribute), 87
- `relation_embeddings` (*dicee.models.quaternion.ConvQ* attribute), 86
- `relation_embeddings` (*dicee.PykeenKGE* attribute), 187
- `relation_to_idx` (*dicee.knowledge_graph.KG* attribute), 46
- `relations_str` (*dicee.knowledge_graph.KG* property), 46
- `reload_dataset()` (in module *dicee*), 200
- `reload_dataset()` (in module *dicee.dataset_classes*), 29
- `report` (*dicee.DICE_Trainer* attribute), 194
- `report` (*dicee.evaluator.Evaluator* attribute), 42
- `report` (*dicee.executer.Execute* attribute), 43
- `report` (*dicee.trainer.DICE_Trainer* attribute), 165
- `report` (*dicee.trainer.dice_trainer.DICE_Trainer* attribute), 160
- `reports` (*dicee.callbacks.Eval* attribute), 24
- `requires_grad_for_interactions` (*dicee.CKeci* attribute), 170
- `requires_grad_for_interactions` (*dicee.Keci* attribute), 170
- `requires_grad_for_interactions` (*dicee.models.CKeci* attribute), 129
- `requires_grad_for_interactions` (*dicee.models.clifford.CKeci* attribute), 68

requires_grad_for_interactions (*dicee.models.clifford.Keci* attribute), 66
 requires_grad_for_interactions (*dicee.models.Keci* attribute), 127
 resid_dropout (*dicee.models.transformers.CausalSelfAttention* attribute), 92
 residual (*dicee.literal_classes.LiteralEmbeddings* attribute), 52
 residual_convolution() (*dicee.AConEx* method), 179
 residual_convolution() (*dicee.AConvO* method), 179
 residual_convolution() (*dicee.AConvQ* method), 180
 residual_convolution() (*dicee.ConEx* method), 182
 residual_convolution() (*dicee.ConvO* method), 182
 residual_convolution() (*dicee.ConvQ* method), 181
 residual_convolution() (*dicee.models.AConEx* method), 113
 residual_convolution() (*dicee.models.AConvO* method), 126
 residual_convolution() (*dicee.models.AConvQ* method), 120
 residual_convolution() (*dicee.models.complex.AConEx* method), 73
 residual_convolution() (*dicee.models.complex.ConEx* method), 72
 residual_convolution() (*dicee.models.ConEx* method), 112
 residual_convolution() (*dicee.models.ConvO* method), 125
 residual_convolution() (*dicee.models.ConvQ* method), 119
 residual_convolution() (*dicee.models.octonion.AConvO* method), 83
 residual_convolution() (*dicee.models.octonion.ConvO* method), 82
 residual_convolution() (*dicee.models.quaternion.AConvQ* method), 87
 residual_convolution() (*dicee.models.quaternion.ConvQ* method), 86
 retrieve_embedding() (*dicee.scripts.index_serve.NeuralSearcher* method), 153
 retrieve_embeddings() (in module *dicee.scripts.index_serve*), 153
 return_multi_hop_query_results() (*dicee.KGE* method), 198
 return_multi_hop_query_results() (*dicee.knowledge_graph_embeddings.KGE* method), 49
 root() (in module *dicee.scripts.index_serve*), 153
 roots (*dicee.models.FMult* attribute), 139
 roots (*dicee.models.function_space.FMult* attribute), 77
 roots (*dicee.models.function_space.GFMult* attribute), 77
 roots (*dicee.models.GFMult* attribute), 139
 runtime (*dicee.analyse_experiments.Experiment* attribute), 19

S

sample_counter (*dicee.abstracts.AbstractPPECallback* attribute), 17
 sample_entity() (*dicee.abstracts.BaseInteractiveKGE* method), 14
 sample_relation() (*dicee.abstracts.BaseInteractiveKGE* method), 14
 sample_triples_ratio (*dicee.config.Namespace* attribute), 28
 sample_triples_ratio (*dicee.knowledge_graph.KG* attribute), 45
 sampling_ratio (*dicee.literal_classes.LiteralDataset* attribute), 53, 54
 sanity_checking_with_arguments() (in module *dicee.sanity_checkers*), 152
 save() (*dicee.abstracts.BaseInteractiveKGE* method), 14
 save() (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk* method), 151
 save() (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk* method), 146
 save_checkpoint() (*dicee.abstracts.AbstractTrainer* static method), 13
 save_checkpoint_model() (in module *dicee*), 193
 save_checkpoint_model() (in module *dicee.static_funcs*), 157
 save_embeddings() (in module *dicee*), 193
 save_embeddings() (in module *dicee.static_funcs*), 157
 save_embeddings_as_csv (*dicee.config.Namespace* attribute), 26
 save_experiment() (*dicee.analyse_experiments.Experiment* method), 19
 save_model_at_every_epoch (*dicee.config.Namespace* attribute), 27
 save_numpy_ndarray() (in module *dicee*), 193
 save_numpy_ndarray() (in module *dicee.read_preprocess_save_load_kg.util*), 149
 save_numpy_ndarray() (in module *dicee.static_funcs*), 157
 save_pickle() (in module *dicee*), 192
 save_pickle() (in module *dicee.read_preprocess_save_load_kg.util*), 150
 save_pickle() (in module *dicee.static_funcs*), 156
 save_queries() (*dicee.query_generator.QueryGenerator* method), 144
 save_queries() (*dicee.QueryGenerator* method), 212
 save_queries_and_answers() (*dicee.query_generator.QueryGenerator* static method), 144
 save_queries_and_answers() (*dicee.QueryGenerator* static method), 212
 save_trained_model() (*dicee.executer.Execute* method), 44
 scalar_batch_NN() (*dicee.LFMult* method), 186
 scalar_batch_NN() (*dicee.models.function_space.LFMult* method), 79
 scalar_batch_NN() (*dicee.models.LFMult* method), 141
 scaler (*dicee.callbacks.Perturb* attribute), 25

scaler (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 164
score() (*dicee.ComplEx* static method), 178
score() (*dicee.DistMult* method), 170
score() (*dicee.Keci* method), 173
score() (*dicee.models.clifford.Keci* method), 68
score() (*dicee.models.ComplEx* static method), 114
score() (*dicee.models.complex.ComplEx* static method), 74
score() (*dicee.models.DistMult* method), 109
score() (*dicee.models.Keci* method), 129
score() (*dicee.models.octonion.OMult* method), 81
score() (*dicee.models.OMult* method), 124
score() (*dicee.models.QMult* method), 118
score() (*dicee.models.quaternion.QMult* method), 85
score() (*dicee.models.real.DistMult* method), 88
score() (*dicee.models.real.TransE* method), 88
score() (*dicee.models.TransE* method), 109
score() (*dicee.OMult* method), 185
score() (*dicee.QMult* method), 184
score() (*dicee.TransE* method), 173
score_func (*dicee.models.FMult2* attribute), 140
score_func (*dicee.models.function_space.FMult2* attribute), 78
scoring_technique (*dicee.analyse_experiments.Experiment* attribute), 19
scoring_technique (*dicee.config.Namespace* attribute), 27
search() (*dicee.scripts.index_serve.NeuralSearcher* method), 153
search_embeddings() (in module *dicee.scripts.index_serve*), 153
search_embeddings_batch() (in module *dicee.scripts.index_serve*), 154
seed (*dicee.query_generator.QueryGenerator* attribute), 143
seed (*dicee.QueryGenerator* attribute), 211
select_model() (in module *dicee*), 193
select_model() (in module *dicee.static_funcs*), 156
selected_optimizer (*dicee.BaseKGE* attribute), 190
selected_optimizer (*dicee.models.base_model.BaseKGE* attribute), 63
selected_optimizer (*dicee.models.BaseKGE* attribute), 104, 107, 111, 115, 121, 134, 137
separator (*dicee.config.Namespace* attribute), 27
separator (*dicee.knowledge_graph.KG* attribute), 46
sequential_vocabulary_construction() (*dicee.read_preprocess_save_load_kg.PreprocessKG* method), 151
sequential_vocabulary_construction() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 145
serve() (in module *dicee.scripts.index_serve*), 154
set_global_seed() (*dicee.query_generator.QueryGenerator* method), 143
set_global_seed() (*dicee.QueryGenerator* method), 212
set_model_eval_mode() (*dicee.abstracts.BaseInteractiveKGE* method), 14
set_model_train_mode() (*dicee.abstracts.BaseInteractiveKGE* method), 14
setup() (*dicee.CVDataModule* method), 209
setup() (*dicee.dataset_classes.CVDataModule* method), 38
setup_executor() (*dicee.executor.Execute* method), 44
Shallom (class in *dicee*), 185
Shallom (class in *dicee.models*), 109
Shallom (class in *dicee.models.real*), 88
shallom (*dicee.models.real.Shallom* attribute), 88
shallom (*dicee.models.Shallom* attribute), 109
shallom (*dicee.Shallom* attribute), 185
single_hop_query_answering() (*dicee.KGE* method), 198
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), 195
start() (*dicee.executor.Execute* method), 44
start() (*dicee.read_preprocess_save_load_kg.PreprocessKG* method), 150
start() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG* method), 144
start() (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk* method), 145
start() (*dicee.read_preprocess_save_load_kg.ReadFromDisk* method), 151
start() (*dicee.trainer.DICE_Trainer* method), 166
start() (*dicee.trainer.dice_trainer.DICE_Trainer* method), 161
start_time (*dicee.callbacks.PrintCallback* attribute), 20
start_time (*dicee.executor.Execute* attribute), 43
step() (*dicee.EnsembleKGE* method), 192
step() (*dicee.models.ADOPT* method), 97
step() (*dicee.models.adopt.ADOPT* method), 55

step() (*dicee.models.ensemble.EnsembleKGE method*), 76
 storage_path (*dicee.config.Namespace attribute*), 26
 storage_path (*dicee.DICE_Trainer attribute*), 194
 storage_path (*dicee.trainer.DICE_Trainer attribute*), 165
 storage_path (*dicee.trainer.dice_trainer.DICE_Trainer attribute*), 160
 store() (*in module dicee*), 193
 store() (*in module dicee.static_funcs*), 157
 store_ensemble() (*dicee.abstracts.AbstractPPECallback method*), 18
 strategy (*dicee.abstracts.AbstractTrainer attribute*), 12
 StringListRequest (*class in dicee.scripts.index_serve*), 153
 swa (*dicee.config.Namespace attribute*), 28

T

T() (*dicee.DualE method*), 177
 T() (*dicee.models.DualE method*), 143
 T() (*dicee.models.dualE.DualE method*), 75
 t_conorm() (*dicee.abstracts.InteractiveQueryDecomposition method*), 15
 t_norm() (*dicee.abstracts.InteractiveQueryDecomposition method*), 15
 target_dim (*dicee.AllvsAll attribute*), 204
 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*), 201
 target_dim (*dicee.OnevsAllDataset attribute*), 202
 temperature (*dicee.BytE attribute*), 188
 temperature (*dicee.models.transformers.BytE attribute*), 90
 tensor_t_norm() (*dicee.abstracts.InteractiveQueryDecomposition method*), 15
 TensorParallel (*class in dicee.trainer.model_parallelism*), 162
 test_data_loader() (*dicee.models.base_model.BaseKGELightning method*), 58
 test_data_loader() (*dicee.models.base_model.BaseKGELightning method*), 99
 test_epoch_end() (*dicee.models.base_model.BaseKGELightning method*), 58
 test_epoch_end() (*dicee.models.BaseKGELightning method*), 99
 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*), 143
 test_path (*dicee.QueryGenerator attribute*), 211
 timeit() (*in module dicee*), 192, 200
 timeit() (*in module dicee.read_preprocess_save_load_kg.util*), 149
 timeit() (*in module dicee.static_funcs*), 156
 timeit() (*in module dicee.static_preprocess_funcs*), 159
 to() (*dicee.EnsembleKGE method*), 192
 to() (*dicee.KGE method*), 195
 to() (*dicee.knowledge_graph_embeddings.KGE method*), 47
 to() (*dicee.models.ensemble.EnsembleKGE method*), 76
 to_df() (*dicee.analyse_experiments.Experiment method*), 19
 topk (*dicee.BytE attribute*), 188
 topk (*dicee.models.transformers.BytE attribute*), 90
 topk (*dicee.scripts.index_serve.NeuralSearcher attribute*), 153
 torch_ordered_shaped_bpe_entities (*dicee.dataset_classes.MultiLabelDataset attribute*), 30
 torch_ordered_shaped_bpe_entities (*dicee.MultiLabelDataset attribute*), 201
 TorchDDPTrainer (*class in dicee.trainer.torch_trainer_ddp*), 164
 TorchTrainer (*class in dicee.trainer.torch_trainer*), 162
 train() (*dicee.KGE method*), 199
 train() (*dicee.knowledge_graph_embeddings.KGE method*), 50
 train() (*dicee.trainer.torch_trainer_ddp.NodeTrainer method*), 165
 train_data (*dicee.AllvsAll attribute*), 203
 train_data (*dicee.dataset_classes.AllvsAll attribute*), 33
 train_data (*dicee.dataset_classes.KvsAll attribute*), 32
 train_data (*dicee.dataset_classes.KvsSampleDataset attribute*), 35
 train_data (*dicee.dataset_classes.MultiClassClassificationDataset attribute*), 31
 train_data (*dicee.dataset_classes.OnevsAllDataset attribute*), 31
 train_data (*dicee.dataset_classes.OnevsSample attribute*), 33, 34
 train_data (*dicee.KvsAll attribute*), 203
 train_data (*dicee.KvsSampleDataset attribute*), 206

train_data (*dicee.MultiClassClassificationDataset* attribute), 201
 train_data (*dicee.OnevsAllDataset* attribute), 202
 train_data (*dicee.OnevsSample* attribute), 204, 205
 train_dataloader() (*dicee.CVDataModule* method), 208
 train_dataloader() (*dicee.dataset_classes.CVDataModule* method), 37
 train_dataloader() (*dicee.models.base_model.BaseKGELightning* method), 60
 train_dataloader() (*dicee.models.BaseKGELightning* method), 101
 train_data_loaders (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 163
 train_dataset_loader (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 164
 train_file_path (*dicee.literal_classes.LiteralDataset* attribute), 53, 54
 train_h1 (*dicee.analyse_experiments.Experiment* attribute), 18
 train_h3 (*dicee.analyse_experiments.Experiment* attribute), 19
 train_h10 (*dicee.analyse_experiments.Experiment* attribute), 19
 train_indices_target (*dicee.dataset_classes.MultiLabelDataset* attribute), 30
 train_indices_target (*dicee.MultiLabelDataset* attribute), 201
 train_k_vs_all() (*dicee.KGE* method), 199
 train_k_vs_all() (*dicee.knowledge_graph_embeddings.KGE* method), 50
 train_literals() (*dicee.KGE* method), 199
 train_literals() (*dicee.knowledge_graph_embeddings.KGE* method), 50
 train_mode (*dicee.EnsembleKGE* attribute), 192
 train_mode (*dicee.models.ensemble.EnsembleKGE* attribute), 76
 train_mrr (*dicee.analyse_experiments.Experiment* attribute), 18
 train_path (*dicee.query_generator.QueryGenerator* attribute), 143
 train_path (*dicee.QueryGenerator* attribute), 211
 train_set (*dicee.BPE_NegativeSamplingDataset* attribute), 200
 train_set (*dicee.dataset_classes.BPE_NegativeSamplingDataset* attribute), 30
 train_set (*dicee.dataset_classes.MultiLabelDataset* attribute), 30
 train_set (*dicee.dataset_classes.NegSampleDataset* attribute), 36
 train_set (*dicee.dataset_classes.TriplePredictionDataset* attribute), 37
 train_set (*dicee.MultiLabelDataset* attribute), 201
 train_set (*dicee.NegSampleDataset* attribute), 206
 train_set (*dicee.TriplePredictionDataset* attribute), 207
 train_set_idx (*dicee.CVDataModule* attribute), 208
 train_set_idx (*dicee.dataset_classes.CVDataModule* attribute), 37
 train_set_target (*dicee.knowledge_graph.KG* attribute), 46
 train_target (*dicee.AllvsAll* attribute), 203
 train_target (*dicee.dataset_classes.AllvsAll* attribute), 33
 train_target (*dicee.dataset_classes.KvsAll* attribute), 32
 train_target (*dicee.dataset_classes.KvsSampleDataset* attribute), 35
 train_target (*dicee.KvsAll* attribute), 203
 train_target (*dicee.KvsSampleDataset* attribute), 206
 train_target_indices (*dicee.knowledge_graph.KG* attribute), 46
 train_triples() (*dicee.KGE* method), 199
 train_triples() (*dicee.knowledge_graph_embeddings.KGE* method), 50
 trained_model (*dicee.executer.Execute* attribute), 43
 trainer (*dicee.config.Namespace* attribute), 27
 trainer (*dicee.DICE_Trainer* attribute), 194
 trainer (*dicee.executer.Execute* attribute), 43
 trainer (*dicee.trainer.DICE_Trainer* attribute), 165
 trainer (*dicee.trainer.dice_trainer.DICE_Trainer* attribute), 160
 trainer (*dicee.trainer.torch_trainer_ddp.NodeTrainer* attribute), 164
 training_step (*dicee.trainer.torch_trainer.TorchTrainer* attribute), 163
 training_step() (*dicee.BytE* method), 188
 training_step() (*dicee.models.base_model.BaseKGELightning* method), 57
 training_step() (*dicee.models.BaseKGELightning* method), 98
 training_step() (*dicee.models.transformers.BytE* method), 90
 training_step_outputs (*dicee.models.base_model.BaseKGELightning* attribute), 56
 training_step_outputs (*dicee.models.BaseKGELightning* attribute), 98
 training_technique (*dicee.knowledge_graph.KG* attribute), 46
 TransE (*class in dicee*), 173
 TransE (*class in dicee.models*), 109
 TransE (*class in dicee.models.real*), 88
 transfer_batch_to_device() (*dicee.CVDataModule* method), 209
 transfer_batch_to_device() (*dicee.dataset_classes.CVDataModule* method), 38
 transformer (*dicee.BytE* attribute), 188
 transformer (*dicee.models.transformers.BytE* attribute), 90
 transformer (*dicee.models.transformers.GPT* attribute), 95
 trapezoid() (*dicee.models.FMult2* method), 140

trapezoid() (*dicee.models.function_space.FMult2 method*), 78
 tri_score() (*dicee.LFMult method*), 186
 tri_score() (*dicee.models.function_space.LFMult method*), 79
 tri_score() (*dicee.models.function_space.LFMult1 method*), 78
 tri_score() (*dicee.models.LFMult method*), 141
 tri_score() (*dicee.models.LFMult1 method*), 140
 triple_score() (*dicee.KGE method*), 197
 triple_score() (*dicee.knowledge_graph_embeddings.KGE method*), 48
 TriplePredictionDataset (*class in dicee*), 207
 TriplePredictionDataset (*class in dicee.dataset_classes*), 36
 tuple2list() (*dicee.query_generator.QueryGenerator method*), 143
 tuple2list() (*dicee.QueryGenerator method*), 211

U

unlabelled_size (*dicee.callbacks.PseudoLabellingCallback attribute*), 22
 unmap() (*dicee.query_generator.QueryGenerator method*), 144
 unmap() (*dicee.QueryGenerator method*), 212
 unmap_query() (*dicee.query_generator.QueryGenerator method*), 144
 unmap_query() (*dicee.QueryGenerator method*), 212

V

val_aswa (*dicee.callbacks.ASWA attribute*), 23
 val_data_loader() (*dicee.models.base_model.BaseKGELighting method*), 59
 val_data_loader() (*dicee.models.BaseKGELighting method*), 100
 val_h1 (*dicee.analyse_experiments.Experiment attribute*), 19
 val_h3 (*dicee.analyse_experiments.Experiment attribute*), 19
 val_h10 (*dicee.analyse_experiments.Experiment attribute*), 19
 val_mrr (*dicee.analyse_experiments.Experiment attribute*), 19
 val_path (*dicee.query_generator.QueryGenerator attribute*), 143
 val_path (*dicee.QueryGenerator attribute*), 211
 validate_knowledge_graph() (*in module dicee.sanity_checkers*), 151
 vocab_preparation() (*dicee.evaluator.Evaluator method*), 42
 vocab_size (*dicee.models.transformers.GPTConfig attribute*), 94
 vocab_to_parquet() (*in module dicee*), 193
 vocab_to_parquet() (*in module dicee.static_funcs*), 157
 vtp_score() (*dicee.LFMult method*), 186
 vtp_score() (*dicee.models.function_space.LFMult method*), 79
 vtp_score() (*dicee.models.function_space.LFMult1 method*), 78
 vtp_score() (*dicee.models.LFMult method*), 141
 vtp_score() (*dicee.models.LFMult1 method*), 140

W

weight (*dicee.models.transformers.LayerNorm attribute*), 91
 weight_decay (*dicee.BaseKGE attribute*), 190
 weight_decay (*dicee.config.Namespace attribute*), 27
 weight_decay (*dicee.models.base_model.BaseKGE attribute*), 63
 weight_decay (*dicee.models.BaseKGE attribute*), 104, 107, 111, 115, 121, 134, 137
 weights (*dicee.models.FMult attribute*), 139
 weights (*dicee.models.function_space.FMult attribute*), 77
 weights (*dicee.models.function_space.GFMult attribute*), 77
 weights (*dicee.models.GFMult attribute*), 139
 write_csv_from_model_parallel() (*in module dicee*), 194
 write_csv_from_model_parallel() (*in module dicee.static_funcs*), 158
 write_links() (*dicee.query_generator.QueryGenerator method*), 144
 write_links() (*dicee.QueryGenerator method*), 212
 write_report() (*dicee.executer.Execute method*), 44

X

x_values (*dicee.LFMult attribute*), 185
 x_values (*dicee.models.function_space.LFMult attribute*), 79
 x_values (*dicee.models.LFMult attribute*), 141