

---

# DICE Embeddings

*Release 0.1.3.2*

**Caglar Demir**

**Jun 19, 2024**

## Contents:

<b>1</b>	<b>Dicee Manual</b>	<b>2</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	Installation from Source . . . . .	3
<b>3</b>	<b>Download Knowledge Graphs</b>	<b>3</b>
<b>4</b>	<b>Knowledge Graph Embedding Models</b>	<b>3</b>
<b>5</b>	<b>How to Train</b>	<b>3</b>
<b>6</b>	<b>Creating an Embedding Vector Database</b>	<b>5</b>
6.1	Learning Embeddings . . . . .	5
6.2	Loading Embeddings into Qdrant Vector Database . . . . .	6
6.3	Launching Webservice . . . . .	6
<b>7</b>	<b>Answering Complex Queries</b>	<b>6</b>
<b>8</b>	<b>Predicting Missing Links</b>	<b>8</b>
<b>9</b>	<b>Downloading Pretrained Models</b>	<b>8</b>
<b>10</b>	<b>How to Deploy</b>	<b>8</b>
<b>11</b>	<b>Docker</b>	<b>8</b>
<b>12</b>	<b>How to cite</b>	<b>9</b>
<b>13</b>	<b>dicee</b>	<b>10</b>
13.1	Subpackages . . . . .	10
13.2	Submodules . . . . .	128
13.3	Attributes . . . . .	204
13.4	Classes . . . . .	204
13.5	Functions . . . . .	205
13.6	Package Contents . . . . .	207
	<b>Python Module Index</b>	<b>244</b>

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

## 1 Dicee Manual

**Version:** dicee 0.1.3.2

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

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

**Contact:** [caglar.demir@upb.de](mailto:caglar.demir@upb.de)

**License:** OSI Approved :: MIT License

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

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

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

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

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

**Why PyTorch<sup>7</sup> & Co. ?** PyTorch is one of the most popular machine learning frameworks available at the time of writing. PytorchLightning facilitates scaling the training procedure of PyTorch without boilerplate. In our framework, we combine **PyTorch**<sup>8</sup> & **PytorchLightning**<sup>9</sup>. Users can choose the trainer class (e.g., DDP by Pytorch) to train large knowledge graph embedding models with billions of parameters. PytorchLightning allows us to use state-of-the-art model parallelism techniques (e.g. Fully Sharded Training, FairScale, or DeepSpeed) without extra effort. With our framework, practitioners can directly use PytorchLightning for model parallelism to train gigantic embedding models.

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

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

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

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

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

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

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

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

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

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

<sup>10</sup> <https://huggingface.co/gradio>

## 2 Installation

### 2.1 Installation from Source

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

or

```
pip install dicee
```

## 3 Download Knowledge Graphs

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

To test the Installation

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

## 4 Knowledge Graph Embedding Models

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

For more, please refer to examples.

## 5 How to Train

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

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

(continues on next page)

(continued from previous page)

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

where the data is in the following form

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

A KGE model can also be trained from the command line

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

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

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

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

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

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

Similarly, models can be easily trained with torchrun

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

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

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

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

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

where the data is in the following form

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

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

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

For more, please refer to examples.

## 6 Creating an Embedding Vector Database

### 6.1 Learning Embeddings

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

## 6.2 Loading Embeddings into Qdrant Vector Database

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

## 6.3 Launching Webservice

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

### Retrieve and Search

Get embedding of germany

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

Get most similar things to europe

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

## 7 Answering Complex Queries

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

(continues on next page)

(continued from previous page)

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

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

## 8 Predicting Missing Links

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

## 9 Downloading Pretrained Models

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

- For more please look at [dice-research.org/projects/DiceEmbeddings/](https://dice-research.org/projects/DiceEmbeddings/)<sup>11</sup>

## 10 How to Deploy

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

## 11 Docker

To build the Docker image:

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

To test the Docker image:

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

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



## 12 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
@inproceedings{demir2023litcqd,
  title={LitCQD: Multi-Hop Reasoning in Incomplete Knowledge Graphs with Numeric
  ↪Literals},
  author={Demir, Caglar and Wiebesiek, Michel and Lu, Renzhong and Ngonga Ngomo, Axel-
  ↪Cyrille and Heindorf, Stefan},
  booktitle={Joint European Conference on Machine Learning and Knowledge Discovery in
  ↪Databases},
  pages={617--633},
  year={2023},
  organization={Springer}
}

# DICE Embedding Framework
@article{demir2022hardware,
  title={Hardware-agnostic computation for large-scale knowledge graph embeddings},
  author={Demir, Caglar and Ngomo, Axel-Cyrille Ngonga},
  journal={Software Impacts},
  year={2022},
  publisher={Elsevier}
}

# KronE
@inproceedings{demir2022kronecker,
  title={Kronecker decomposition for knowledge graph embeddings},
  author={Demir, Caglar and Lienen, Julian and Ngonga Ngomo, Axel-Cyrille},
  booktitle={Proceedings of the 33rd ACM Conference on Hypertext and Social Media},
  pages={1--10},
  year={2022}
}

# QMult, OMult, ConvQ, ConvO
@InProceedings{pmlr-v157-demir21a,
  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},
```

(continues on next page)

(continued from previous page)

```
pdf = {https://proceedings.mlr.press/v157/demir21a/demir21a.pdf},
url = {https://proceedings.mlr.press/v157/demir21a.html},
}
# ConEx
@inproceedings{demir2021convolutional,
title={Convolutional Complex Knowledge Graph Embeddings},
author={Caglar Demir and Axel-Cyrille Ngonga Ngomo},
booktitle={Eighteenth Extended Semantic Web Conference - Research Track},
year={2021},
url={https://openreview.net/forum?id=6T45-4TFqaX}}
# Shallom
@inproceedings{demir2021shallow,
title={A shallow neural model for relation prediction},
author={Demir, Caglar and Moussallem, Diego and Ngomo, Axel-Cyrille Ngonga},
booktitle={2021 IEEE 15th International Conference on Semantic Computing (ICSC)},
pages={179--182},
year={2021},
organization={IEEE}
```

For any questions or wishes, please contact: [caglar.demir@upb.de](mailto:caglar.demir@upb.de)

## 13 dicee

### 13.1 Subpackages

**dicee.models**

**Submodules**

**dicee.models.base\_model**

**Classes**

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

### Module Contents

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

Bases: `lightning.LightningModule`

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

```

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

---

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

---

### Variables

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

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

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

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

---

**Note:** Some things to know:

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

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

**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.base_model.IdentityClass (args=None)`

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

#### Variables

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

`__call__ (x)`

`static forward (x)`

## dicee.models.clifford

### Classes

<i>BaseKGE</i>	Base class for all neural network modules.
<i>CMult</i>	Cl_(0,0) => Real Numbers
<i>Keci</i>	Base class for all neural network modules.
<i>KeciBase</i>	Without learning dimension scaling
<i>DeCaL</i>	Base class for all neural network modules.

### Module Contents

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

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

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

#### Variables

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

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

#### Parameters

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

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

byte pair encoded neural link predictors

### Parameters

-----

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

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

### Parameters

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

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

### Parameters

**x**

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

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

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

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

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

### Parameters

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

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

### Parameters

**x** (*B* × 2 × *x* × *T*)

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

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

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

Cl\_(0,0) => Real Numbers

Cl\_(0,1) =>

A multivector  $\mathbf{a} = a_0 + a_1 e_1$  A multivector  $\mathbf{b} = b_0 + b_1 e_1$

multiplication is isomorphic to the product of two complex numbers

$\mathbf{a} \text{ imes } \mathbf{b} = a_0 b_0 + a_0 b_1 e_1 + a_1 b_1 e_1 e_1$   
 $= (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) e_1$

Cl\_(2,0) =>

A multivector  $\mathbf{a} = a_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 e_2$  A multivector  $\mathbf{b} = b_0 + b_1 e_1 + b_2 e_2 + b_{12} e_1 e_2$

$\mathbf{a} \text{ imes } \mathbf{b} = a_0 b_0 + a_0 b_1 e_1 + a_0 b_2 e_2 + a_0 b_{12} e_1 e_2$

$$\bullet a_1 b_0 e_1 + a_1 b_1 e_1 e_1 ..$$

$Cl_{(0,2)} \Rightarrow$  Quaternions

**clifford\_mul** (*x: torch.FloatTensor, y: torch.FloatTensor, p: int, q: int*)  $\rightarrow$  tuple

Clifford multiplication  $Cl_{\{p,q\}}$  ( $\mathbb{R}$ )

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

eq  $j$

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

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

*p*: a non-negative integer  $p \geq 0$  *q*: a non-negative integer  $q \geq 0$

**score** (*head\_ent\_emb, rel\_ent\_emb, tail\_ent\_emb*)

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

Compute batch triple scores

### Parameter

*x*: torch.LongTensor with shape n by 3

**rtype**

torch.LongTensor with shape n

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

Compute batch KvsAll triple scores

### Parameter

*x*: torch.LongTensor with shape n by 3

**rtype**

torch.LongTensor with shape n

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)
```

(continues on next page)

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

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

---

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

---

## Variables

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

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

Compute  $\text{sigma}_{\{pp\}} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_{ir_k} - h_{kr_i}) e_{ie_k}$

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

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

    for k in range(i + 1, p):

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

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

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

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

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

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

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

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

    for k in range(j + 1, q):

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

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

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

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

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

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

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

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 (h0, hp, hq, r0, rp, rq)
    Multiplying a base vector with its scalar coefficient
clifford_multiplication (h0, hp, hq, r0, rp, rq)
    Compute our CL multiplication
    
$$h = h_0 + \sum_{i=1}^p h_i e_i + \sum_{j=p+1}^{p+q} h_j e_j$$


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


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

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

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

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

## Parameter

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

**rtype**

torch.FloatTensor with (n, **IE**) 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.**KeciBase** (*args*)

Bases: *Keci*

Without learning dimension scaling

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

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

### Parameter

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

#### rtype

torch.FloatTensor with (n) shape

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

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

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

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

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

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

and return:

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

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

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

For same bases vectors interaction we have

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

For different base vector interactions, we have

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



**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, **IE**) shape

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

Multiplying a base vector with its scalar coefficient

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

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

## Parameter

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

### returns

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

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

Compute .. math:

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

$\sigma_{\{pp\}}$  captures the interactions between along p bases For instance, let  $p \in e_1, e_2, e_3$ , 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

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

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

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

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

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

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

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

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

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

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

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

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

Compute

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

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

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

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

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

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

Compute

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

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

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

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

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

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

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

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

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

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

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

## dicee.models.complex

### Classes

<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.

### Module Contents

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

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

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

#### Variables

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

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

#### Parameters

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

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

byte pair encoded neural link predictors

### Parameters

-----

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

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

### Parameters

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

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

### Parameters

**x**

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

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

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

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

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

### Parameters

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

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

### Parameters

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

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

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

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

Convolutional ComplEx Knowledge Graph Embeddings

**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

**residual\_convolution** (*C\_1: Tuple[torch.Tensor, torch.Tensor],  
C\_2: Tuple[torch.Tensor, torch.Tensor]*)  $\rightarrow$  torch.FloatTensor

Compute residual score of two complex-valued embeddings. :param C\_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C\_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

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

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

**Parameters**

**x**

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

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

**Variables**

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

**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
```

## dicee.models.dualE

### Classes

<i>BaseKGE</i>	Base class for all neural network modules.
<i>DualE</i>	Dual Quaternion Knowledge Graph Embeddings ( <a href="https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657">https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657</a> )

### Module Contents

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

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

### Variables

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

**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.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>)

**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.function\_space

### Classes

<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:

### Module Contents

**class** dicee.models.function\_space.**BaseKGE** (*args: dict*)

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

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

#### Variables

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

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

#### Parameters

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

```

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

    Parameters
    -----

init_params_with_sanity_checking ()

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

    Parameters

    • x

    • y_idx

    • ordered_bpe_entities

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

    Parameters
    x

forward_k_vs_all (*args, **kwargs)

forward_k_vs_sample (*args, **kwargs)

get_triple_representation (idx_hrt)

get_head_relation_representation (indexed_triple)

get_sentence_representation (x: torch.LongTensor)

    Parameters

    • (b (x shape))

    • 3

    • t)

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

    Parameters

    x (B × 2 × T)

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

class dicee.models.function_space.FMult (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs

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

    chain_func (weights, x: torch.FloatTensor)

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

    Parameters

    x

```

```

class dicee.models.function_space.GFMult (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    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
    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
    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:
 $f(x) = \sum_{i=0}^{d-1} a_k x^{i\%d}$  and use the three differents scoring function as in the paper to evaluate the score.
    We also consider combining with Neural Networks.
    forward_triples (idx_triple)

```

#### Parameters

**x**

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

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

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

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

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

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

**tri\_score** ( $coeff_h$ ,  $coeff_r$ ,  $coeff_t$ )

this part implement the trilinear scoring techniques:

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

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

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

this part implement the vector triple product scoring techniques:

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

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

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

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

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

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

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

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

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

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

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

## dicее.models.octonion

### Classes

<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

### Functions

```
octonion_mul(*, O_1, O_2)

octonion_mul_norm(*, O_1, O_2)
```

### Module Contents

**class** dicее.models.octonion.**BaseKGE** (*args: dict*)

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

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

## Variables

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

**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.octonion.IdentityClass (args=None)

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

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

**static forward** (*x*)

dicee.models.octonion.octonion\_mul (\*, *O\_1*, *O\_2*)

dicee.models.octonion.octonion\_mul\_norm (\*, *O\_1*, *O\_2*)

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
```

(continues on next page)

(continued from previous page)

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

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

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

---

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

---

### Variables

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

**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  $\Rightarrow \text{shape}(\text{size of batch}, |\text{Entities}|)$

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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



---

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

---

### Variables

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

**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** `dicke.models.octonion.AConvO` (*args: dict*)

Bases: `dicke.models.base_model.BaseKGE`

Additive Convolutional Octonion Knowledge Graph Embeddings

**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)

## dicke.models.pykeen\_models

### Classes

<code>BaseKGE</code>	Base class for all neural network modules.
<code>PykeenKGE</code>	A class for using knowledge graph embedding models implemented in Pykeen

## Module Contents

**class** dicee.models.pykeen\_models.**BaseKGE** (*args: dict*)

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

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

### Parameters

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

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

byte pair encoded neural link predictors

### Parameters

-----

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

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

### Parameters

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

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

**Parameters**

**x**

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

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

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

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

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

**Parameters**

- (**b** (*x shape*)

- 3

- **t**)

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

**Parameters**

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

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

**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:

**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)

    @abstractmethod
    def forward_k_vs_sample(x: torch.LongTensor, target_entity_idx):

```

## dicee.models.quaternion

### Classes

<code>BaseKGE</code>	Base class for all neural network modules.
<code>IdentityClass</code>	Base class for all neural network modules.
<code>QMult</code>	Base class for all neural network modules.
<code>ConvQ</code>	Convolutional Quaternion Knowledge Graph Embeddings
<code>ACConvQ</code>	Additive Convolutional Quaternion Knowledge Graph Embeddings

### Functions

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

### Module Contents

```

dicee.models.quaternion.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.quaternion.BaseKGE(args: dict)

```

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

```

(continues on next page)

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

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

---

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

---

### Variables

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

**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* × 2 × *T*)

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

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

#### Variables

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

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

**static forward** (*x*)

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

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

```

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

---

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

---

### Variables

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

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

### Parameters

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

### Returns

Triple scores.

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

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

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

### Parameters

**x** – The vector.

### Returns

The normalized vector.

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

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

**Parameters**

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

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

**Parameters**

**x**

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

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

**class** dicee.models.quaternion.**ConvQ** (*args*)

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

Convolutional Quaternion Knowledge Graph Embeddings

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

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

**Parameters**

**x**

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

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

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

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

Additive Convolutional Quaternion Knowledge Graph Embeddings

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

### Module Contents

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

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

#### Variables

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

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

#### Parameters

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

**forward\_byte\_pair\_encoded\_triple** (*x*: *Tuple[torch.LongTensor, torch.LongTensor]*)  
byte pair encoded neural link predictors

**Parameters**  
-----

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

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

**Parameters**

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

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

**Parameters**  
**x**

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

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

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

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

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

**Parameters**

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

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

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

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

**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>

**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
    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)
    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
    forward_triples (x: torch.LongTensor)

    Parameters
    x

```

## `dicee.models.static_funcs`

### Functions

---

<code>quaternion_mul(→</code>	<code>Tuple[torch.Tensor,</code>	Perform quaternion multiplication
<code>torch.Tensor, ...)</code>		

---

## Module Contents

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

## dicee.models.transformers

### Classes

<i>BaseKGE</i>	Base class for all neural network modules.
<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.BaseKGE(args: dict)
```

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

```
import torch.nn as nn  
import torch.nn.functional as F  
  
class Model(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.conv1 = nn.Conv2d(1, 20, 5)  
        self.conv2 = nn.Conv2d(20, 20, 5)  
  
    def forward(self, x):  
        x = F.relu(self.conv1(x))  
        return F.relu(self.conv2(x))
```

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

---

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

---

## Variables

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

**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.transformers.Byte(*args, **kwargs)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

**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

**forward**(*input*)

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

**forward** (*x*)

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

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---



### Variables

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

**forward** (*x*)

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

**forward** (*x*)

**class** dicee.models.transformers.**GPTConfig**

**block\_size:** int = 1024

**vocab\_size:** int = 50304

**n\_layer:** int = 12

**n\_head:** int = 12

**n\_embd:** int = 768

**dropout:** float = 0.0

**bias:** bool = False

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

**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** (*fwd\_bwd\_per\_iter, dt*)

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

## Classes

<i>BaseKGELightning</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>IdentityClass</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>DistMult</i>	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
<i>TransE</i>	Translating Embeddings for Modeling
<i>Shallom</i>	A shallow neural model for relation prediction ( <a href="https://arxiv.org/abs/2101.09090">https://arxiv.org/abs/2101.09090</a> )
<i>Pyke</i>	A Physical Embedding Model for Knowledge Graphs
<i>BaseKGE</i>	Base class for all neural network modules.
<i>ConEx</i>	Convolutional ComplEx Knowledge Graph Embeddings
<i>AConEx</i>	Additive Convolutional ComplEx Knowledge Graph Embeddings
<i>Complex</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>IdentityClass</i>	Base class for all neural network modules.
<i>QMult</i>	Base class for all neural network modules.
<i>ConvQ</i>	Convolutional Quaternion Knowledge Graph Embeddings
<i>AConvQ</i>	Additive Convolutional Quaternion Knowledge Graph Embeddings
<i>BaseKGE</i>	Base class for all neural network modules.
<i>IdentityClass</i>	Base class for all neural network modules.
<i>OMult</i>	Base class for all neural network modules.
<i>ConvO</i>	Base class for all neural network modules.
<i>AConvO</i>	Additive Convolutional Octonion Knowledge Graph Embeddings
<i>Keci</i>	Base class for all neural network modules.
<i>KeciBase</i>	Without learning dimension scaling
<i>CMult</i>	$Cl_{(0,0)} \Rightarrow$ Real Numbers
<i>DeCaL</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>PykeenKGE</i>	A class for using knowledge graph embedding models implemented in Pykeen
<i>BaseKGE</i>	Base class for all neural network modules.
<i>FMult</i>	Learning Knowledge Neural Graphs
<i>GFMult</i>	Learning Knowledge Neural Graphs
<i>FMult2</i>	Learning Knowledge Neural Graphs
<i>LFMult1</i>	Embedding with trigonometric functions. We represent all entities and relations in the complex number space as:
<i>LFMult</i>	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
<i>DualE</i>	Dual Quaternion Knowledge Graph Embeddings ( <a href="https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657">https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657</a> )

## Functions

<code>quaternion_mul(→</code>	<code>Tuple[torch.Tensor,</code>	<code>Perform quaternion multiplication</code>
<code>torch.Tensor, ...)</code>		
<code>quaternion_mul_with_unit_norm(*,</code>	<code>Q_1,</code>	
<code>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.**BaseKGELightning** (\*args, \*\*kwargs)

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

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



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

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

---

**Note:** Some things to know:

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

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

## Variables

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

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

```

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

```

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

---

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

---

### Variables

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

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

**static forward** (*x*)

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

```

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

---

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

---

### Variables

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

**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*.**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>

**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>

**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>)

**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

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

**Parameters**

**x**

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

```

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

---

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

---

### Variables

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

**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 × 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

    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

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

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

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

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

    Parameters
        x

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

```

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

**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*

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

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

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.



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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

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

### Parameters

**x** (*B × 2 × T*)

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

byte pair encoded neural link predictors

### Parameters

-----

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

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

### Parameters

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

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

### Parameters

**x**

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

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

`get_triple_representation (idx_hrt)`

`get_head_relation_representation (indexed_triple)`

`get_sentence_representation (x: torch.LongTensor)`

#### Parameters

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

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

#### Parameters

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

#### Variables

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

`__call__ (x)`

`static forward (x)`

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

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

## Variables

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

**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

**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

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

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

#### Parameters

**x**

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

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

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

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

### Parameters

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

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

byte pair encoded neural link predictors

### Parameters

-----

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

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

### Parameters

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

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

**Parameters**

**x**

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

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

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

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

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

**Parameters**

- (**b** (*x* shape)

- 3

- **t**)

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

**Parameters**

**x** (*B* × 2 × *T*)

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

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

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

**static forward** (*x*)

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

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

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

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

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

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

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

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

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

### Variables

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

**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

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

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



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

#### Parameters

**x**

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

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

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

#### Variables

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

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

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

$\sigma_{pp}$  captures the interactions between along *p* bases For instance, let *p* *e*<sub>1</sub>, *e*<sub>2</sub>, *e*<sub>3</sub>, we compute interactions between *e*<sub>1</sub> *e*<sub>2</sub>, *e*<sub>1</sub> *e*<sub>3</sub>, and *e*<sub>2</sub> *e*<sub>3</sub> This can be implemented with a nested two for loops

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

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

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

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

Yet, this computation would be quite inefficient. Instead, we compute interactions along all *p*, e.g., *e*<sub>1</sub>*e*<sub>1</sub>, *e*<sub>1</sub>*e*<sub>2</sub>, *e*<sub>1</sub>*e*<sub>3</sub>,

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

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

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

Compute  $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_{j,r_k} - h_{k,r_j}) e_j e_k$  captures the interactions between along  $q$  bases. For instance, let  $q = e_1, e_2, e_3$ , we compute interactions between  $e_1e_2, e_1e_3$ , and  $e_2e_3$ . This can be implemented with a nested two for loops

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

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

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

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

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

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

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

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

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

print(sigma\_pq.shape)

**apply\_coefficients** ( $h_0, hp, hq, r_0, rp, rq$ )

Multiplying a base vector with its scalar coefficient

**clifford\_multiplication** ( $h_0, hp, hq, r_0, rp, rq$ )

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  $i$

$e_j$

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

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

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

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

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

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

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

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

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

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

## Parameter

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

### returns

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

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

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

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

Kvsall training

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

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

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

## Parameter

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

### rtype

torch.FloatTensor with (n, **IEI**) shape

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

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

## Parameter

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

**rtype**

torch.FloatTensor with (n) shape

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

Bases: *Keci*

Without learning dimension scaling

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

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

Cl\_(0,0) => Real Numbers

**Cl\_(0,1) =>**

A multivector  $\mathbf{a} = a_0 + a_1 e_1$  A multivector  $\mathbf{b} = b_0 + b_1 e_1$

multiplication is isomorphic to the product of two complex numbers

$$\mathbf{a} \mathbf{b} = a_0 b_0 + a_0 b_1 e_1 + a_1 b_1 e_1 e_1 \\ = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) e_1$$

**Cl\_(2,0) =>**

A multivector  $\mathbf{a} = a_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 e_2$  A multivector  $\mathbf{b} = b_0 + b_1 e_1 + b_2 e_2 + b_{12} e_1 e_2$

$$\mathbf{a} \mathbf{b} = a_0 b_0 + a_0 b_1 e_1 + a_0 b_2 e_2 + a_0 b_{12} e_1 e_2 \\ + a_1 b_0 e_1 + a_1 b_1 e_1 e_1 + \dots$$

Cl\_(0,2) => Quaternions

**clifford\_mul** (x: torch.FloatTensor, y: torch.FloatTensor, p: int, q: int) → tuple

Clifford multiplication  $Cl_{\{p,q\}}(\mathbb{R})$

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

$e_i e_j$

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

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

p: a non-negative integer  $p \geq 0$  q: a non-negative integer  $q \geq 0$

**score** (head\_ent\_emb, rel\_ent\_emb, tail\_ent\_emb)

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

Compute batch triple scores

## Parameter

x: torch.LongTensor with shape n by 3

**rtype**

torch.LongTensor with shape n

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

Compute batch KvsAll triple scores

## Parameter

x: torch.LongTensor with shape n by 3

**rtype**

torch.LongTensor with shape n

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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

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

---

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

---

## Variables

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

**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:

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

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

Compute

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

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

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

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

Compute

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

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

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

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

print(sigma\_pq.shape)

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

Compute

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

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

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

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

print(sigma\_pq.shape)

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

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

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

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

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

print(sigma\_pq.shape)

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

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



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

---

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

---

### Variables

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

**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.**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:

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

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

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

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

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

t = self.entity\_embeddings.weight.reshape(self.num\_entities, self.embedding\_dim, self.last\_dim)

**else:**

t = self.entity\_embeddings.weight

# (4) Call the score\_t from interactions to generate triple scores. return self.interaction.score\_t(h=h, r=r, all\_entities=t, slice\_size=1)

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

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

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

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

# (3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice\_size=None, slice\_dim=0)

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

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)
```

(continues on next page)

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

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

---

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

---

### Variables

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

**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*)  
 $\rightarrow$  Tuple[torch.FloatTensor, torch.FloatTensor]

**Parameters**

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

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

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

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

Learning Knowledge Neural Graphs

**compute\_func** (*weights*: *torch.FloatTensor*,  $x$ )  $\rightarrow$  torch.FloatTensor

**chain\_func** (*weights*,  $x$ : *torch.FloatTensor*)

**forward\_triples** (*idx\_triple*: *torch.Tensor*)  $\rightarrow$  torch.Tensor

**Parameters**

$\mathbf{x}$

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

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

Learning Knowledge Neural Graphs

**compute\_func** (*weights*: *torch.FloatTensor*,  $x$ )  $\rightarrow$  torch.FloatTensor

**chain\_func** (*weights*,  $x$ : *torch.FloatTensor*)

**forward\_triples** (*idx\_triple*: *torch.Tensor*)  $\rightarrow$  torch.Tensor

**Parameters**

$\mathbf{x}$

**class** dicee.models.**FMult2** (*args*)

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

Learning Knowledge Neural Graphs

**build\_func** (*Vec*)

**build\_chain\_funcs** (*list\_Vec*)

**compute\_func** ( $W$ ,  $b$ ,  $x$ )  $\rightarrow$  torch.FloatTensor

**function** (*list\_W*, *list\_b*)

**trapezoid** (*list\_W*, *list\_b*)

**forward\_triples** (*idx\_triple*: *torch.Tensor*)  $\rightarrow$  torch.Tensor

**Parameters**

$\mathbf{x}$

**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

**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:  $f(x) = \sum_{i=0}^{d-1} a_k x^{i\%d}$  and use the three different scoring function as in the paper to evaluate the score. We also consider combining with Neural Networks.

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

**Parameters**

**x**

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

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

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

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

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

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

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

this part implement the trilinear scoring techniques:

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

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

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

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

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

This function takes a matrix tensor of coefficients (*coeff*), a tensor vector of points *x* and range of integer  $[0, 1, \dots, d]$  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>)

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

Submodules

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

Classes

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

Functions

<i>timeit</i> (func)	
<i>index_triples_with_pandas</i> (→ das.core.frame.DataFrame)	pan- <b>param train_set</b> pandas dataframe
<i>dataset_sanity_checking</i> (→ None)	<b>param train_set</b>
<i>numpy_data_type_changer</i> (→ numpy.ndarray)	Detect most efficient data type for a given triples
<i>get_er_vocab</i> (data[, file_path])	
<i>get_re_vocab</i> (data[, file_path])	
<i>get_ee_vocab</i> (data[, file_path])	
<i>create_constraints</i> (triples[, file_path])	(1) Extract domains and ranges of relations
<i>apply_reciprical_or_noise</i> (add_reciprical, eval_model)	(1) Add reciprocal triples (2) Add noisy triples

## Module Contents

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

`dicee.read_preprocess_save_load_kg.preprocess.index_triples_with_pandas (train_set, entity_to_idx: dict, relation_to_idx: dict) → pandas.core.frame.DataFrame`

### Parameters

- **train\_set** – pandas dataframe
- **entity\_to\_idx** – a mapping from str to integer index
- **relation\_to\_idx** – a mapping from str to integer index
- **num\_core** – number of cores to be used

### Returns

indexed triples, i.e., pandas dataframe

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

### Parameters

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

### Returns

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

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

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

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

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

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

Preprocess the data in memory

**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 reciprocal 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)

**remove\_triples\_from\_train\_with\_condition()**

**dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk**

## Classes

---

<i>ReadFromDisk</i>
---------------------

Read the data from disk into memory
-------------------------------------

---

## Functions

---

<i>read_from_disk(data_path[, read_only_few, ...])</i>
--

<i>read_from_triple_store([endpoint])</i>
---

Read triples from triple store into pandas dataframe
--

---

## Module Contents

```
dicee.read_preprocess_save_load_kg.read_from_disk.read_from_disk(  
    data_path: str, read_only_few: int = None, sample_triples_ratio: float = None, backend=None)  
  
dicee.read_preprocess_save_load_kg.read_from_disk.read_from_triple_store(  
    endpoint: str = None)  
    Read triples from triple store into pandas dataframe  
  
class dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk(kg)  
    Read the data from disk into memory  
  
    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*

---

## Functions

---

*load\_pickle*(\*, file\_path)

*load\_numpy\_ndarray*(\*, file\_path)

*save\_pickle*(\*, data, file\_path)

*save\_numpy\_ndarray*(\*, data, file\_path)

---

## Module Contents

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

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

```
dicee.read_preprocess_save_load_kg.save_load_disk.save_pickle(* ,  
    data: object = None, file_path=str)
```

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

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

```
    save()
```

```
    load()
```

```
dicee.read_preprocess_save_load_kg.util
```

## Functions

<code>apply_reciprical_or_noise(add_reciprical, eval_model)</code>	(1) Add reciprocal triples (2) Add noisy triples
<code>timeit(func)</code>	
<code>read_with_polars(→ polars.DataFrame)</code> <code>read_with_pandas(data_path[, read_only_few, ...])</code>	Load and Preprocess via Polars
<code>read_from_disk(data_path[, read_only_few, ...])</code>	
<code>read_from_triple_store(endpoint)</code> <code>get_er_vocab(data[, file_path])</code>	Read triples from triple store into pandas dataframe
<code>get_re_vocab(data[, file_path])</code>	
<code>get_ee_vocab(data[, file_path])</code>	
<code>create_constraints(triples[, file_path])</code>	(1) Extract domains and ranges of relations
<code>load_with_pandas(→ None)</code> <code>save_numpy_ndarray(*, data, file_path)</code>	Deserialize data
<code>load_numpy_ndarray(*, file_path)</code>	
<code>save_pickle(*, data[, file_path])</code>	
<code>load_pickle(*[, file_path])</code>	
<code>create_recipriocal_triples(x)</code> <code>index_triples_with_pandas(→ pan- das.core.frame.DataFrame)</code>	Add inverse triples into dask dataframe <b>param train_set</b> pandas dataframe
<code>dataset_sanity_checking(→ None)</code>	<b>param train_set</b>

## Module Contents

`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) → 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)
```

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

```
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.index_triples_with_pandas(train_set,  
    entity_to_idx: dict, relation_to_idx: dict) → pandas.core.frame.DataFrame
```

#### Parameters

- **train\_set** – pandas dataframe
- **entity\_to\_idx** – a mapping from str to integer index
- **relation\_to\_idx** – a mapping from str to integer index
- **num\_core** – number of cores to be used

#### Returns

indexed triples, i.e., pandas dataframe

```
dicee.read_preprocess_save_load_kg.util.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

**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)

```

    remove_triples_from_train_with_condition()
class dicee.read_preprocess_save_load_kg.LoadSaveToDisk(kg)
    save()
    load()
class dicee.read_preprocess_save_load_kg.ReadFromDisk(kg)
    Read the data from disk into memory
    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.scripts`

### Submodules

#### `dicee.scripts.index`

### Functions

---

```
get_default_arguments()
```

```
main()
```

---

### Module Contents

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

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

## dicee.scripts.run

### Classes

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

### Functions

<i>get_default_arguments</i> ([description])	Extends pytorch_lightning Trainer's arguments with ours
<i>main</i> ()	

### Module Contents

**class** `dicee.scripts.run.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

**read\_or\_load\_kg** ()

**read\_preprocess\_index\_serialize\_data** () → None

Read & Preprocess & Index & Serialize Input Data

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

#### Parameter

**rtype**

None

**load\_indexed\_data** () → None

Load the indexed data from disk into memory



## Parameter

**rtype**  
None

**save\_trained\_model** () → None

Save a knowledge graph embedding model

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

## Parameter

**rtype**  
None

**end** (*form\_of\_labelling: str*) → dict

End training

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

## Parameter

**rtype**  
A dict containing information about the training and/or evaluation

**write\_report** () → None

Report training related information in a report.json file

**start** () → dict

Start training

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

## Parameter

**rtype**  
A dict containing information about the training and/or evaluation

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

Extends pytorch\_lightning Trainer's arguments with ours

`dicee.scripts.run.main()`

## dicee.scripts.serve

## Attributes

*app*

*neural\_searcher*

## Classes

*KGE*

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

*NeuralSearcher*

## Functions

`get_default_arguments()`

`root()`

`search_embeddings(q)`

`retrieve_embeddings(q)`

`main()`

## Module Contents

**class** `dicee.scripts.serve.KGE` (*path=None, url=None, construct\_ensemble=False, model\_name=None, apply\_semantic\_constraint=False*)

Bases: `dicee.abstracts.BaseInteractiveKGE`

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

**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=""*)

**\_\_str\_\_** ()  
Return `str(self)`.

**eval\_lp\_performance** (*dataset=List[Tuple[str, str, str]], filtered=True*)

**predict\_missing\_head\_entity** (*relation: List[str] | str, tail\_entity: List[str] | str, within=None*)  
→ `Tuple`

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

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

### Parameter

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

String representation of selected relations.

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

String representation of selected entities.

**k**: `int`

Highest ranked k entities.

### Returns: Tuple

Highest K scores and entities

**predict\_missing\_relations** (*head\_entity: List[str] | str, tail\_entity: List[str] | str, within=None*)  
→ Tuple

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

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

### Parameter

head\_entity: List[str]

String representation of selected entities.

tail\_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

### Returns: Tuple

Highest K scores and entities

**predict\_missing\_tail\_entity** (*head\_entity: List[str] | str, relation: List[str] | str, within: List[str] = None*) → torch.FloatTensor

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

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

### Parameter

head\_entity: List[str]

String representation of selected entities.

tail\_entity: List[str]

String representation of selected entities.

### Returns: Tuple

scores

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

### Parameters

- **logits**
- **h**

- **r**
- **t**
- **within**

**predict\_topk** (\*, *h*: List[str] = None, *r*: List[str] = None, *t*: List[str] = None, *topk*: int = 10, *within*: List[str] = None)

Predict missing item in a given 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.

k: int

Highest ranked k item.

### Returns: Tuple

Highest K scores and items

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

Predict triple score

### Parameter

head\_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail\_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

## Returns: Tuple

pytorch tensor of triple score

**t\_norm** (*tens\_1*: torch.Tensor, *tens\_2*: torch.Tensor, *tnorm*: str = 'min') → torch.Tensor

**tensor\_t\_norm** (*subquery\_scores*: torch.FloatTensor, *tnorm*: str = 'min') → torch.FloatTensor

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

**t\_conorm** (*tens\_1*: torch.Tensor, *tens\_2*: torch.Tensor, *tconorm*: str = 'min') → torch.Tensor

**negnorm** (*tens\_1*: torch.Tensor, *lambda\_*: float, *neg\_norm*: str = 'standard') → torch.Tensor

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

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

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

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

Find an answer set for EPFO queries including negation and disjunction

## Parameter

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

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

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

*tnorm*: str The t-norm operator.

*neg\_norm*: str The negation norm.

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

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

### returns

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

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

Find missing triples

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

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

Return (e,r,x)

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

confidence: float

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

topk: int

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

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.

`dicee.scripts.serve.app`

`dicee.scripts.serve.neural_searcher = None`

`dicee.scripts.serve.get_default_arguments()`

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

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

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

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

**get** (*entity: str*)

**search** (*entity: str*)

`dicee.scripts.serve.main()`

**dicee.trainer**

**Submodules**

**dicee.trainer.dice\_trainer**

## Classes

<i>BaseKGE</i>	Base class for all neural network modules.
<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>PrintCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>AccumulateEpochLossCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>Perturb</i>	A callback for a three-Level Perturbation
<i>TorchTrainer</i>	TorchTrainer for using single GPU or multi CPUs on a single node
<i>TorchDDPTrainer</i>	A Trainer based on torch.nn.parallel.DistributedDataParallel
<i>KG</i>	Knowledge Graph
<i>DICE_Trainer</i>	DICE_Trainer implement

## Functions

<i>select_model</i> (args[, is_continual_training, storage_path])	
<i>construct_dataset</i> (→ torch.utils.data.Dataset)	
<i>reload_dataset</i> (path, form_of_labelling, ...)	Reload the files from disk to construct the Pytorch dataset
<i>timeit</i> (func)	
<i>initialize_trainer</i> (args, callbacks)	
<i>get_callbacks</i> (args)	

## Module Contents

**class** dicee.trainer.dice\_trainer.**BaseKGE** (args: dict)

Bases: BaseKGELightning

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
```

(continues on next page)



```

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

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

```

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

---

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

---

### Variables

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

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

### Parameters

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

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

byte pair encoded neural link predictors

### Parameters

-----

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

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

### Parameters

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

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

### Parameters

**x**

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

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

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

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

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

### Parameters

- (**b** (*x shape*))

- 3
- t)

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

#### Parameters

**x** (*B* × 2 × *T*)

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

`dicee.trainer.dice_trainer.select_model` (*args*: dict, *is\_continual\_training*: bool = None, *storage\_path*: str = None)

**class** `dicee.trainer.dice_trainer.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.

**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.trainer.dice\_trainer.**Eval** (*path, epoch\_ratio: int = None*)

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

## Parameter

**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.trainer.dice\_trainer.**KronE**

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

## Parameter

**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.trainer.dice\_trainer.**PrintCallback**

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

## Parameter

**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.trainer.dice\_trainer.**AccumulateEpochLossCallback** (*path: str*)

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

## Parameter

**on\_fit\_end** (*trainer, model*) → None

Store epoch loss

## Parameter

trainer:

model:

**rtype**

None

**class** `dicee.trainer.dice_trainer.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:

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

Called when the train batch begins.

`dicee.trainer.dice_trainer.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`

`dicee.trainer.dice_trainer.reload_dataset` (*path: str, form\_of\_labelling, scoring\_technique, neg\_ratio, label\_smoothing\_rate*)

Reload the files from disk to construct the Pytorch dataset

**class** `dicee.trainer.dice_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

**fit** (*\*args, train\_data loaders, \*\*kwargs*) → None

Training starts

Arguments

**kwargs: Tuple**

empty dictionary

**Return type**

batch loss (float)

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

Compute forward, loss, backward, and parameter update

Arguments

**Return type**

batch loss (float)

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

Construct inputs and outputs from a batch of inputs with outputs From a batch of inputs and put

Arguments

**Return type**

(tuple) mini-batch on select device

**class** `dicee.trainer.dice_trainer.TorchDDPTrainer` (*args, callbacks*)Bases: `dicee.abstracts.AbstractTrainer`

A Trainer based on torch.nn.parallel.DistributedDataParallel

Arguments

**entity\_idxs**

mapping.

**relation\_idxs**

mapping.

**form**

?

**store**

?

**label\_smoothing\_rate**

Using hard targets (0,1) drives weights to infinity. An outlier produces enormous gradients.

**Return type**

torch.utils.data.Dataset

**fit** (*\*args, \*\*kwargs*)

Train model

`dicee.trainer.dice_trainer.timeit` (*func*)

**class** `dicee.trainer.dice_trainer.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\_reciprical: 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*)

Knowledge Graph

```

    property entities_str: List
    property relations_str: List

    func_triple_to_bpe_representation (triple: List[str])

dicee.trainer.dice_trainer.initialize_trainer (args, callbacks)

dicee.trainer.dice_trainer.get_callbacks (args)

class dicee.trainer.dice_trainer.DICE_Trainer (args, is_continual_training, storage_path,
    evaluator=None)

DICE_Trainer implement
    1- Pytorch Lightning trainer (https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html)
    2- Multi-GPU Trainer(https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html)
    3- CPU Trainer

    args
    is_continual_training:bool
    storage_path:str
    evaluator:
    report:dict

continual_start ()

    (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
    Initialize Trainer from input arguments

initialize_or_load_model ()

initialize_dataloader (dataset: torch.utils.data.Dataset) → torch.utils.data.DataLoader

initialize_dataset (dataset: dicee.knowledge\_graph.KG, form_of_labelling)
    → torch.utils.data.Dataset

start (knowledge_graph: dicee.knowledge\_graph.KG) → Tuple[dicee.models.base_model.BaseKGE, str]
    Train selected model via the selected training strategy

```



**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

## **dicee.trainer.torch\_trainer**

### **Classes**

<i>AbstractTrainer</i>	Abstract class for Trainer class for knowledge graph embedding models
<i>TorchTrainer</i>	TorchTrainer for using single GPU or multi CPUs on a single node

### **Module Contents**

**class** `dicee.trainer.torch_trainer.AbstractTrainer` (*args, callbacks*)

Abstract class for Trainer class for knowledge graph embedding models

#### **Parameter**

**args**

[str] ?

**callbacks: list**

?

**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.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

**fit** (*\*args, train\_data loaders, \*\*kwargs*) → None

Training starts

Arguments

**kwargs:Tuple**

empty dictionary

**Return type**

batch loss (float)

**forward\_backward\_update** (*x\_batch: torch.Tensor, y\_batch: torch.Tensor*) → torch.Tensor

Compute forward, loss, backward, and parameter update

Arguments

**Return type**

batch loss (float)

**extract\_input\_outputs\_set\_device** (*batch: list*) → Tuple

Construct inputs and outputs from a batch of inputs with outputs From a batch of inputs and put

Arguments

**Return type**

(tuple) mini-batch on select device

**dicee.trainer.torch\_trainer\_ddp**

## Classes

<code>AbstractTrainer</code>	Abstract class for Trainer class for knowledge graph embedding models
<code>TorchDDPTrainer</code>	A Trainer based on torch.nn.parallel.DistributedDataParallel
<code>NodeTrainer</code>	
<code>DDPTrainer</code>	

## Functions

<code>efficient_zero_grad(model)</code>
<code>print_peak_memory(prefix, device)</code>

## Module Contents

**class** dicee.trainer.torch\_trainer\_ddp.**AbstractTrainer** (*args, callbacks*)

Abstract class for Trainer class for knowledge graph embedding models

### Parameter

**args**

[str] ?

**callbacks:** list

?

**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

`dicee.trainer.torch_trainer_ddp.efficient_zero_grad(model)`

`dicee.trainer.torch_trainer_ddp.print_peak_memory(prefix, device)`

**class** `dicee.trainer.torch_trainer_ddp.TorchDDPTrainer` (args, callbacks)

Bases: `dicee.abstracts.AbstractTrainer`

A Trainer based on `torch.nn.parallel.DistributedDataParallel`

Arguments

**entity\_idx**s  
mapping.

**relation\_idx**s  
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, optimizer: torch.optim.Optimizer, callbacks,  
num\_epochs: int)

**extract\_input\_outputs** (z: list)

**train** ()  
Training loop for DDP

**class** dicee.trainer.torch\_trainer\_ddp.**DDPTrainer** (model: torch.nn.Module,  
train\_dataset\_loader: torch.utils.data.DataLoader, optimizer: torch.optim.Optimizer, gpu\_id: int,  
callbacks, num\_epochs)

**extract\_input\_outputs** (z: list)

**train** ()

## Classes

<i>DICE_Trainer</i>
---------------------

DICE_Trainer implement
------------------------

## Package Contents

**class** dicee.trainer.**DICE\_Trainer** (args, is\_continual\_training, storage\_path, evaluator=None)

**DICE\_Trainer implement**

1- Pytorch Lightning trainer (<https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html>)  
2- Multi-GPU Trainer(<https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>)  
3- CPU Trainer

args

is\_continual\_training:bool

storage\_path:str

```

evaluator:
report:dict
continual_start ()

(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

Initialize Trainer from input arguments

**initialize\_or\_load\_model** ()

**initialize\_dataloader** (*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

**initialize\_dataset** (*dataset: [dicee.knowledge\\_graph.KG](#), form\_of\_labelling*)  
→ torch.utils.data.Dataset

**start** (*knowledge\_graph: [dicee.knowledge\\_graph.KG](#)*) → Tuple[*[dicee.models.base\\_model.BaseKGE](#)*, str]

Train selected model via the selected training strategy

**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

## 13.2 Submodules

### `dicee.abstracts`

#### Classes

<code>AbstractTrainer</code>	Abstract class for Trainer class for knowledge graph embedding models
<code>BaseInteractiveKGE</code>	Abstract/base class for using knowledge graph embedding models interactively.
<code>AbstractCallback</code>	Abstract class for Callback class for knowledge graph embedding models
<code>AbstractPPECallback</code>	Abstract class for Callback class for knowledge graph embedding models

#### Functions

<code>load_model_ensemble(...)</code>	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
<code>load_model(→ Tuple[object, Tuple[dict, dict]])</code>	Load weights and initialize pytorch module from namespace arguments
<code>save_checkpoint_model(→ None)</code>	Store Pytorch model into disk
<code>load_json(→ dict)</code>	
<code>download_pretrained_model(→ str)</code>	

#### Module Contents

`dicee.abstracts.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.abstracts.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.abstracts.save_checkpoint_model` (*model, path: str*) → `None`  
Store Pytorch model into disk

`dicee.abstracts.load_json` (*p: str*) → `dict`



`dicee.abstracts.download_pretrained_model (url: str) → str`

**class** `dicee.abstracts.AbstractTrainer (args, callbacks)`

Abstract class for Trainer class for knowledge graph embedding models

### Parameter

**args**

[str] ?

**callbacks: list**

?

**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

**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**

**get\_domain\_of\_relation** (*rel: str*) → List[str]

**get\_range\_of\_relation** (*rel: str*) → List[str]

**set\_model\_train\_mode** () → None  
Setting the model into training mode

### Parameter

**set\_model\_eval\_mode** () → None  
Setting the model into eval mode

### Parameter

**property name**

**sample\_entity** (*n: int*) → List[str]

**sample\_relation** (*n: int*) → List[str]

**is\_seen** (*entity: str = None, relation: str = None*) → bool

**save** () → None

**get\_entity\_index** (*x: str*)

**get\_relation\_index** (*x: str*)

**index\_triple** (*head\_entity: List[str], relation: List[str], tail\_entity: List[str]*)  
→ Tuple[torch.LongTensor, torch.LongTensor, torch.LongTensor]  
Index Triple

### Parameter

head\_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail\_entity: List[str]

String representation of selected entities.

### Returns: Tuple

pytorch tensor of triple score

**add\_new\_entity\_embeddings** (*entity\_name: str = None, embeddings: torch.FloatTensor = None*)

**get\_entity\_embeddings** (*items: List[str]*)

Return embedding of an entity given its string representation

## Parameter

**items:**  
entities

**get\_relation\_embeddings** (*items: List[str]*)  
Return embedding of a relation given its string representation

## Parameter

**items:**  
relations

**construct\_input\_and\_output** (*head\_entity: List[str], relation: List[str], tail\_entity: List[str], labels*)  
Construct a data point :param head\_entity: :param relation: :param tail\_entity: :param labels: :return:  
**parameters** ()

**class** dicee.abstracts.**AbstractCallback**  
Bases: abc.ABC, lightning.pytorch.callbacks.Callback  
Abstract class for Callback class for knowledge graph embedding models

## Parameter

**on\_init\_start** (*\*args, \*\*kwargs*)

## Parameter

trainer:  
model:  
**rtype**  
None

**on\_init\_end** (*\*args, \*\*kwargs*)  
Call at the beginning of the training.

## Parameter

trainer:  
model:  
**rtype**  
None

**on\_fit\_start** (*trainer, model*)  
Call at the beginning of the training.

### Parameter

trainer:

model:

**rtype**

None

**on\_train\_epoch\_end** (*trainer, model*)

Call at the end of each epoch during training.

### Parameter

trainer:

model:

**rtype**

None

**on\_train\_batch\_end** (*\*args, \*\*kwargs*)

Call at the end of each mini-batch during the training.

### Parameter

trainer:

model:

**rtype**

None

**on\_fit\_end** (*\*args, \*\*kwargs*)

Call at the end of the training.

### Parameter

trainer:

model:

**rtype**

None

**class** dicee.abstracts.**AbstractPPECallback** (*num\_epochs, path, epoch\_to\_start, last\_percent\_to\_consider*)

Bases: *AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

## Parameter

**on\_fit\_start** (*trainer, model*)

Call at the beginning of the training.

## Parameter

trainer:

model:

**rtype**

None

**on\_fit\_end** (*trainer, model*)

Call at the end of the training.

## Parameter

trainer:

model:

**rtype**

None

**store\_ensemble** (*param\_ensemble*) → None

## dicee.analyse\_experiments

This script should be moved to dicee/scripts

## Classes

---

*Experiment*

---

## Functions

---

*get\_default\_arguments()*

*analyse(args)*

---

## Module Contents

`dicee.analyse_experiments.get_default_arguments()`

**class** `dicee.analyse_experiments.Experiment`

`save_experiment(x)`

`to_df()`

`dicee.analyse_experiments.analyse(args)`

## dicee.callbacks

### Classes

<i>AbstractCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>AccumulateEpochLossCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>PrintCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>KGESaveCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>PseudoLabellingCallback</i>	Abstract class for Callback class for knowledge graph embedding models
<i>ASWA</i>	Adaptive stochastic weight averaging
<i>Eval</i>	Abstract class for Callback class for knowledge graph embedding models
<i>KronE</i>	Abstract class for Callback class for knowledge graph embedding models
<i>Perturb</i>	A callback for a three-Level Perturbation

### Functions

<i>save_checkpoint_model</i> ( $\rightarrow$ None)	Store Pytorch model into disk
<i>save_pickle</i> (*[, data, file_path])	
<i>estimate_q</i> (eps)	estimate rate of convergence q from sequence esp
<i>compute_convergence</i> (seq, i)	

## Module Contents

`dicee.callbacks.save_checkpoint_model(model, path: str) → None`

Store Pytorch model into disk

`dicee.callbacks.save_pickle(*, data: object = None, file_path=str)`

**class** `dicee.callbacks.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.callbacks.**AccumulateEpochLossCallback** (path: str)

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

### Parameter

**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

**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

**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

**create\_random\_data** ()

**on\_epoch\_end** (*trainer, model*)

`dicee.callbacks.estimate_q` (*eps*)

estimate rate of convergence q from sequence esp

`dicee.callbacks.compute_convergence` (*seq, i*)

**class** `dicee.callbacks.ASWA` (*num\_epochs, path*)

Bases: `dicee.abstracts.AbstractCallback`

Adaptive stochastic weight averaging ASWE keeps track of the validation performance and update s the ensemble model accordingly.

**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

**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

**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:

**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

**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

**num\_core: int = 0**

Number of CPUs to be used in the mini-batch loading process

**random\_seed: int = 0**

Random Seed

**sample\_triples\_ratio: float = None**

Read some triples that are uniformly at random sampled. Ratio being between 0 and 1

**read\_only\_few: int = None**

Read only first few triples



### **pykeen\_model\_kwargs**

Additional keyword arguments for pykeen models

**kernel\_size: int = 3**

Size of a square kernel in a convolution operation

**num\_of\_output\_channels: int = 32**

Number of slices in the generated feature map by convolution.

**p: int = 0**

P parameter of Clifford Embeddings

**q: int = 1**

Q parameter of Clifford Embeddings

**input\_dropout\_rate: float = 0.0**

Dropout rate on embeddings of input triples

**hidden\_dropout\_rate: float = 0.0**

Dropout rate on hidden representations of input triples

**feature\_map\_dropout\_rate: float = 0.0**

Dropout rate on a feature map generated by a convolution operation

**byte\_pair\_encoding: bool = False**

Byte pair encoding

**Type**

WIP

**adaptive\_swa: bool = False**

Adaptive stochastic weight averaging

**swa: bool = False**

Stochastic weight averaging

**block\_size: int = None**

block size of LLM

**continual\_learning = None**

Path of a pretrained model size of LLM

**\_\_iter\_\_()**

### **dicee.dataset\_classes**

## Classes

<i>BPE_NegativeSamplingDataset</i>	An abstract class representing a Dataset.
<i>MultiLabelDataset</i>	An abstract class representing a Dataset.
<i>MultiClassClassificationDataset</i>	Dataset for the 1vsALL training strategy
<i>OnevsAllDataset</i>	Dataset for the 1vsALL training strategy
<i>KvsAll</i>	Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.
<i>AllvsAll</i>	Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.
<i>KvsSampleDataset</i>	KvsSample a Dataset:
<i>NegSampleDataset</i>	An abstract class representing a Dataset.
<i>TriplePredictionDataset</i>	Triple Dataset
<i>CVDDataModule</i>	Create a Dataset for cross validation

## Functions

<i>mapping_from_first_two_cols_to_third</i> (tr	
<i>timeit</i> (func)	
<i>load_pickle</i> ([file_path])	
<i>reload_dataset</i> (path, form_of_labelling, ...)	Reload the files from disk to construct the Pytorch dataset
<i>construct_dataset</i> (→ torch.utils.data.Dataset)	

## Module Contents

dicee.dataset\_classes.**mapping\_from\_first\_two\_cols\_to\_third** (train\_set\_idx)

dicee.dataset\_classes.**timeit** (func)

dicee.dataset\_classes.**load\_pickle** (file\_path=str)

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.

---

`__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.

---

`__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\_idxs** – mapping.
- **relation\_idxs** – mapping.
- **form** – ?
- **num\_workers** – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

#### Return type

`torch.utils.data.Dataset`

`__len__()`

`__getitem__(idx)`

**class** dicee.dataset\_classes.**OnevsAllDataset** (*train\_set\_idx: numpy.ndarray, entity\_idx*s)

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

`__len__()`

`__getitem__(idx)`

**class** dicee.dataset\_classes.**KvsAll** (*train\_set\_idx: numpy.ndarray, entity\_idx*s, *relation\_idx*s, *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  $[0, 1]^{|IE|}$  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\_idx**s

[dictionary] string representation of an entity to its integer id

**relation\_idx**s

[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])
```

`__len__()`

`__getitem__ (idx)`

**class** dicee.dataset\_classes.**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.  $(h \ r \ E_i)$  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\_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])
```

`__len__ ()`

`__getitem__ (idx)`

**class** dicee.dataset\_classes.**KvsSampleDataset** (*train\_set: numpy.ndarray, num\_entities, num\_relations, neg\_sample\_ratio: int = None, label\_smoothing\_rate: float = 0.0*)

Bases: torch.utils.data.Dataset

**KvsSample a Dataset:**

$D := \{(x, y)_i\}_i^N$ , where

.  $x: (h, r)$  is a unique  $h$  in  $E$  and a relation  $r$  in  $R$  and .  $y$  in  $[0, 1]^{|E|}$  is a binary label.

orall  $y_i = 1$  s.t.  $(h \ r \ E_i)$  in KG

**At each mini-batch construction, we subsample(y), hence n**

**$|new\_y| \ll |E|$**  new\_y contains all 1's if  $\text{sum}(y) < \text{neg\_sample\_ratio}$  new\_y contains

**train\_set\_idx**

Indexed triples for the training.

**entity\_idxxs**

mapping.

**relation\_idx**  
mapping.

**form**  
?

**store**  
?

**label\_smoothing\_rate**  
?

torch.utils.data.Dataset

**\_\_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.

---

**\_\_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:

or all (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**s  
mapping.

**form**  
?

**store**  
?

label\_smoothing\_rate

collate\_fn: batch:List[torch.IntTensor] Returns —— torch.utils.data.Dataset

**\_\_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\_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.

<b>Warning:</b> do not assign state in <code>prepare_data</code>
--

- `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

### Classes

<i>KGE</i>	Knowledge Graph Embedding Class for interactive usage of pre-trained models
------------	---

### Functions

<i>evaluate_link_prediction_performance(→ Dict)</i>	<b>param model</b>
<i>evaluate_link_prediction_performance_w</i>	
<i>evaluate_link_prediction_performance_w</i>	
<i>evaluate_link_prediction_performance_w</i> ...)	<b>param model</b>
<i>evaluate_lp_bpe_k_vs_all(model, triples[, er_vocab, ...])</i>	

## Module Contents

**class** `dicee.eval_static_funcs.KGE` (*path=None, url=None, construct\_ensemble=False, model\_name=None, apply\_semantic\_constraint=False*)

Bases: `dicee.abstracts.BaseInteractiveKGE`

Knowledge Graph Embedding Class for interactive usage of pre-trained models

**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=""*)

**\_\_str\_\_** ()

Return str(self).

**eval\_lp\_performance** (*dataset=List[Tuple[str, str, str]], filtered=True*)

**predict\_missing\_head\_entity** (*relation: List[str] | str, tail\_entity: List[str] | str, within=None*)  
→ `Tuple`

Given a relation and a tail entity, return top k ranked head entity.

$\operatorname{argmax}_{\{e \in E\}} f(e, r, t)$ , where  $r \in R, t \in E$ .

### Parameter

*relation*: `Union[List[str], str]`

String representation of selected relations.

*tail\_entity*: `Union[List[str], str]`

String representation of selected entities.

*k*: `int`

Highest ranked k entities.

### Returns: Tuple

Highest K scores and entities

**predict\_missing\_relations** (*head\_entity: List[str] | str, tail\_entity: List[str] | str, within=None*)  
→ `Tuple`

Given a head entity and a tail entity, return top k ranked relations.

$\operatorname{argmax}_{\{r \in R\}} f(h, r, t)$ , where  $h, t \in E$ .

## Parameter

head\_entity: List[str]

String representation of selected entities.

tail\_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

## Returns: Tuple

Highest K scores and entities

**predict\_missing\_tail\_entity** (*head\_entity: List[str] | str, relation: List[str] | str, within: List[str] = None*) → torch.FloatTensor

Given a head entity and a relation, return top k ranked entities

$\text{argmax}_{\{e \in E\}} f(h, r, e)$ , where  $h \in E$  and  $r \in R$ .

## Parameter

head\_entity: List[str]

String representation of selected entities.

tail\_entity: List[str]

String representation of selected entities.

## Returns: Tuple

scores

**predict** (\*, *h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, within=None, logits=True*) → torch.FloatTensor

### Parameters

- **logits**
- **h**
- **r**
- **t**
- **within**

**predict\_topk** (\*, *h: List[str] = None, r: List[str] = None, t: List[str] = None, topk: int = 10, within: List[str] = None*)

Predict missing item in a given 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.

k: int

Highest ranked k item.

### Returns: Tuple

Highest K scores and items

**triple\_score** (*h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, logits=False*)  
→ torch.FloatTensor

Predict triple score

### Parameter

head\_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail\_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

### Returns: Tuple

pytorch tensor of triple score

**t\_norm** (*tens\_1: torch.Tensor, tens\_2: torch.Tensor, tnorm: str = 'min'*) → torch.Tensor

**tensor\_t\_norm** (*subquery\_scores: torch.FloatTensor, tnorm: str = 'min'*) → torch.FloatTensor

Compute T-norm over  $[0,1]^{n \times d}$  where n denotes the number of hops and d denotes number of entities

**t\_conorm** (*tens\_1: torch.Tensor, tens\_2: torch.Tensor, tconorm: str = 'min'*) → torch.Tensor

**negnorm** (*tens\_1: torch.Tensor, lambda\_: float, neg\_norm: str = 'standard'*) → torch.Tensor

```

return_multi_hop_query_results (aggregated_query_for_all_entities, k: int, only_scores)

single_hop_query_answering (query: tuple, only_scores: bool = True, k: int = None)

answer_multi_hop_query (query_type: str = None,
    query: Tuple[str | Tuple[str, str], Ellipsis] = None,
    queries: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, tnorm: str = 'prod',
    neg_norm: str = 'standard', lambda_: float = 0.0, k: int = 10, only_scores=False)
    → List[Tuple[str, torch.Tensor]]

# @TODO: Refactoring is needed # @TODO: Score computation for each query type should be done in a
static function

Find an answer set for EPFO queries including negation and disjunction

```

## Parameter

*query\_type*: str The type of the query, e.g., “2p”.

*query*: Union[str, Tuple[str, Tuple[str, str]]] The query itself, either a string or a nested tuple.

*queries*: List of Tuple[Union[str, Tuple[str, str]], ...]

*tnorm*: str The t-norm operator.

*neg\_norm*: str The negation norm.

**lambda\_**: float lambda parameter for sugeno and yager negation norms

*k*: int The top-k substitutions for intermediate variables.

### returns

- *List[Tuple[str, torch.Tensor]]*
- *Entities and corresponding scores sorted in the descening order of scores*

```

find_missing_triples (confidence: float, entities: List[str] = None, relations: List[str] = None,
    topk: int = 10, at_most: int = sys.maxsize) → Set

```

Find missing triples

Iterative over a set of entities E and a set of relation R :

forall e in E and forall r in R f(e,r,x)

Return (e,r,x)

until G and f(e,r,x) > confidence

confidence: float

A threshold for an output of a sigmoid function given a triple.

topk: int

Highest ranked k item to select triples with f(e,r,x) > confidence .

at\_most: int

Stop after finding at\_most missing triples

{(e,r,x) | f(e,r,x) > confidence and (e,r,x)}

until G

```

deploy (share: bool = False, top_k: int = 10)

train_triples (h: List[str], r: List[str], t: List[str], labels: List[float], iteration=2, optimizer=None)

train_k_vs_all (h, r, iteration=1, lr=0.001)
    Train k vs all :param head_entity: :param relation: :param iteration: :param lr: :return:
train (kg, lr=0.1, epoch=10, batch_size=32, neg_sample_ratio=10, num_workers=1) → None
    Retrained a pretrain model on an input KG via negative sampling.

dicee.eval_static_funcs.evaluate_link_prediction_performance (
    model: dicee.knowledge_graph_embeddings.KGE, triples, er_vocab: Dict[Tuple, List],
    re_vocab: Dict[Tuple, List]) → Dict

Parameters
    • model
    • triples
    • er_vocab
    • re_vocab

dicee.eval_static_funcs.evaluate_link_prediction_performance_with_reciprocals (
    model: dicee.knowledge_graph_embeddings.KGE, triples, er_vocab: Dict[Tuple, List])

dicee.eval_static_funcs.evaluate_link_prediction_performance_with_bpe_reciprocals (
    model: dicee.knowledge_graph_embeddings.KGE, within_entities: List[str], triples: List[List[str]],
    er_vocab: Dict[Tuple, List])

dicee.eval_static_funcs.evaluate_link_prediction_performance_with_bpe (
    model: dicee.knowledge_graph_embeddings.KGE, within_entities: List[str], triples: List[Tuple[str]],
    er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List])

Parameters
    • model
    • triples
    • within_entities
    • er_vocab
    • re_vocab

dicee.eval_static_funcs.evaluate_lp_bpe_k_vs_all (model, triples: List[List[str]],
    er_vocab=None, batch_size=None, func_triple_to_bpe_representation: Callable = None,
    str_to_bpe_entity_to_idx=None)

```

## dicee.evaluator

### Classes

<i>KG</i>	Knowledge Graph
<i>Evaluator</i>	Evaluator class to evaluate KGE models in various downstream tasks

### Functions

<i>evaluate_lp</i> (model, triple_idx, num_entities, er_vocab, ...)	Evaluate model in a standard link prediction task
<i>evaluate_bpe_lp</i> (model, triple_idx, ..., info)	

### Module Contents

`dicee.evaluator.evaluate_lp(model, triple_idx, num_entities, er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List], info='Eval Starts')`

Evaluate model in a standard link prediction task

for each triple the rank is computed by taking the mean of the filtered missing head entity rank and the filtered missing tail entity rank :param model: :param triple\_idx: :param info: :return:

`dicee.evaluator.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')`

**class** `dicee.evaluator.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_reciprical: 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)`

Knowledge Graph

**property** `entities_str: List`

**property** `relations_str: List`

**func** `triple_to_bpe_representation(triple: List[str])`

**class** `dicee.evaluator.Evaluator(args, is_continual_training=None)`

Evaluator class to evaluate KGE models in various downstream tasks

Arguments

**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>KG</i>	Knowledge Graph
<i>Evaluator</i>	Evaluator class to evaluate KGE models in various downstream tasks
<i>DICE_Trainer</i>	DICE_Trainer implement
<i>Execute</i>	A class for Training, Retraining and Evaluation a model.
<i>ContinuousExecute</i>	A subclass of Execute Class for retraining

## Functions

<code>preprocesses_input_args(args)</code> <code>timeit(func)</code>	Sanity Checking in input arguments
<code>continual_training_setup_executor(→ None)</code> <code>read_or_load_kg(args, cls)</code>	<code>storage_path:str</code> A path leading to a parent directory, where a subdirectory containing KGE related data
<code>load_json(→ dict)</code>	
<code>store(→ None)</code>	Store trained_model model and save embeddings into csv file.

## Module Contents

```
class dicee.executer.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_reciprical: 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)
```

Knowledge Graph

**property entities\_str: List**

**property relations\_str: List**

**func triple\_to\_bpe\_representation** (*triple: List[str]*)

```
class dicee.executer.Evaluator (args, is_continual_training=None)
```

Evaluator class to evaluate KGE models in various downstream tasks

Arguments

**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.preprocesses_input_args` (args)

Sanity Checking in input arguments

**class** `dicee.executer.DICE_Trainer` (args, is\_continual\_training, storage\_path, evaluator=None)

#### DICE\_Trainer implement

- 1- Pytorch Lightning trainer (<https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html>)
- 2- Multi-GPU Trainer(<https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>)
- 3- CPU Trainer

args

is\_continual\_training:bool

storage\_path:str

evaluator:

report:dict

**continual\_start** ()

- (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

Initialize Trainer from input arguments

**initialize\_or\_load\_model** ()

**initialize\_dataloader** (*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

**initialize\_dataset** (*dataset: dicee.knowledge\_graph.KG, form\_of\_labelling*)  
→ torch.utils.data.Dataset

**start** (*knowledge\_graph: dicee.knowledge\_graph.KG*) → Tuple[dicee.models.base\_model.BaseKGE, str]

Train selected model via the selected training strategy

**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

`dicee.executer.timeit` (*func*)

`dicee.executer.continual_training_setup_executor` (*executor*) → None

`storage_path`:str A path leading to a parent directory, where a subdirectory containing KGE related data

`full_storage_path`:str A path leading to a subdirectory containing KGE related data

`dicee.executer.read_or_load_kg` (*args, cls*)

`dicee.executer.load_json` (*p: str*) → dict

`dicee.executer.store` (*trainer, trained\_model, model\_name: str = 'model',  
full\_storage\_path: str = None, save\_embeddings\_as\_csv=False*) → None

Store trained\_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param full\_storage\_path: path to save parameters. :param model\_name: string representation of the name of the model. :param trained\_model: an instance of BaseKGE see core.models.base\_model . :param save\_embeddings\_as\_csv: for easy access of embeddings. :return:

**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

**read\_or\_load\_kg** ()

**read\_preprocess\_index\_serialize\_data** () → None

Read & Preprocess & Index & Serialize Input Data

- (1) Read or load the data from disk into memory.
- (2) Store the statistics of the data.

### Parameter

**rtype**

None

**load\_indexed\_data** () → None

Load the indexed data from disk into memory

### Parameter

**rtype**

None

**save\_trained\_model** () → None

Save a knowledge graph embedding model

- (1) Send model to eval mode and cpu.
- (2) Store the memory footprint of the model.
- (3) Save the model into disk.
- (4) Update the stats of KG again ?

### Parameter

**rtype**

None

**end** (*form\_of\_labelling: str*) → dict

End training

- (1) Store trained model.
- (2) Report runtimes.
- (3) Eval model if required.

## Parameter

### **rtype**

A dict containing information about the training and/or evaluation

**write\_report** () → None

Report training related information in a report.json file

**start** () → dict

Start training

# (1) Loading the Data # (2) Create an evaluator object. # (3) Create a trainer object. # (4) Start the training

## Parameter

### **rtype**

A dict containing information about the training and/or evaluation

**class** dicee.executer.**ContinuousExecute** (*args*)

Bases: *Execute*

A subclass of Execute Class for retraining

(1) Loading & Preprocessing & Serializing input data.

(2) Training & Validation & Testing

(3) Storing all necessary info

During the continual learning we can only modify \* **num\_epochs** \* parameter. Trained model stored in the same folder as the seed model for the training. Trained model is noted with the current time.

**continual\_start** () → dict

Start Continual Training

(1) Initialize training.

(2) Start continual training.

(3) Save trained model.

## Parameter

### **rtype**

A dict containing information about the training and/or evaluation

## dicee.knowledge\_graph

### Classes

<i>ReadFromDisk</i>	Read the data from disk into memory
<i>PreprocessKG</i>	Preprocess the data in memory
<i>LoadSaveToDisk</i>	
<i>KG</i>	Knowledge Graph

### Module Contents

**class** dicee.knowledge\_graph.**ReadFromDisk**(*kg*)

Read the data from disk into memory

**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** ()

**class** dicee.knowledge\_graph.**PreprocessKG**(*kg*)

Preprocess the data in memory

**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 reciprocal 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)

**remove\_triples\_from\_train\_with\_condition**()

**class** dicee.knowledge\_graph.**LoadSaveToDisk**(kg)

**save**()

**load**()

**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\_reciprical: 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)

Knowledge Graph

**property** entities\_str: List

**property** relations\_str: List

**func** triple\_to\_bpe\_representation(triple: List[str])

## **dicee.knowledge\_graph\_embeddings**

### Classes

<i>BaseInteractiveKGE</i>	Abstract/base class for using knowledge graph embedding models interactively.
<i>TriplePredictionDataset</i>	Triple Dataset
<i>KGE</i>	Knowledge Graph Embedding Class for interactive usage of pre-trained models



## Functions

```
random_prediction(pre_trained_kge)

deploy_triple_prediction(pre_trained_kge,
str_subject, ...)
deploy_tail_entity_prediction(pre_trained_]
...)
deploy_relation_prediction(pre_trained_kge,
...)
deploy_head_entity_prediction(pre_trained_]
...)
load_pickle([file_path])

evaluate_lp(model,    triple_idx,    num_entities,    Evaluate model in a standard link prediction task
er_vocab, ...)
```

## Module Contents

**class** `dicee.knowledge_graph_embeddings.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

**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**

**get\_domain\_of\_relation** (*rel: str*) → List[str]

**get\_range\_of\_relation** (*rel: str*) → List[str]

**set\_model\_train\_mode** () → None

Setting the model into training mode

### Parameter

**set\_model\_eval\_mode** () → None

Setting the model into eval mode

### Parameter

**property name**

**sample\_entity** (*n: int*) → List[str]

**sample\_relation** (*n: int*) → List[str]

**is\_seen** (*entity: str = None, relation: str = None*) → bool

**save** () → None

**get\_entity\_index** (*x: str*)

**get\_relation\_index** (*x: str*)

**index\_triple** (*head\_entity: List[str], relation: List[str], tail\_entity: List[str]*)  
→ Tuple[torch.LongTensor, torch.LongTensor, torch.LongTensor]

Index Triple

### Parameter

head\_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail\_entity: List[str]

String representation of selected entities.

### Returns: Tuple

pytorch tensor of triple score

**add\_new\_entity\_embeddings** (*entity\_name: str = None, embeddings: torch.FloatTensor = None*)

**get\_entity\_embeddings** (*items: List[str]*)

Return embedding of an entity given its string representation

## Parameter

### items:

entities

**get\_relation\_embeddings** (*items: List[str]*)

Return embedding of a relation given its string representation

## Parameter

### items:

relations

**construct\_input\_and\_output** (*head\_entity: List[str], relation: List[str], tail\_entity: List[str], labels*)

Construct a data point :param head\_entity: :param relation: :param tail\_entity: :param labels: :return:

**parameters** ()

```
class dicee.knowledge_graph_embeddings.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

**\_\_len\_\_** ()

**\_\_getitem\_\_** (idx)

```

collate_fn (batch: List[torch.Tensor])

dicee.knowledge_graph_embeddings.random_prediction (pre_trained_kge)

dicee.knowledge_graph_embeddings.deploy_triple_prediction (pre_trained_kge,
    str_subject, str_predicate, str_object)

dicee.knowledge_graph_embeddings.deploy_tail_entity_prediction (pre_trained_kge,
    str_subject, str_predicate, top_k)

dicee.knowledge_graph_embeddings.deploy_relation_prediction (pre_trained_kge,
    str_subject, str_object, top_k)

dicee.knowledge_graph_embeddings.deploy_head_entity_prediction (pre_trained_kge,
    str_object, str_predicate, top_k)

dicee.knowledge_graph_embeddings.load_pickle (file_path=str)

dicee.knowledge_graph_embeddings.evaluate_lp (model, triple_idx, num_entities,
    er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List], info='Eval Starts')
    Evaluate model in a standard link prediction task

    for each triple the rank is computed by taking the mean of the filtered missing head entity rank and the filtered
    missing tail entity rank :param model: :param triple_idx: :param info: :return:

class dicee.knowledge_graph_embeddings.KGE (path=None, url=None,
    construct_ensemble=False, model_name=None, apply_semantic_constraint=False)
    Bases: dicee.abstracts.BaseInteractiveKGE

    Knowledge Graph Embedding Class for interactive usage of pre-trained models

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="")

__str__ ()
    Return str(self).

eval_lp_performance (dataset=List[Tuple[str, str, str]], filtered=True)

predict_missing_head_entity (relation: List[str] | str, tail_entity: List[str] | str, within=None)
    → Tuple
    Given a relation and a tail entity, return top k ranked head entity.

     $\operatorname{argmax}_{\{e \in E\}} f(e, r, t)$ , where  $r \in R$ ,  $t \in E$ .

```

## Parameter

relation: Union[List[str], str]

String representation of selected relations.

tail\_entity: Union[List[str], str]

String representation of selected entities.

k: int

Highest ranked k entities.

## Returns: Tuple

Highest K scores and entities

**predict\_missing\_relations** (*head\_entity: List[str] | str, tail\_entity: List[str] | str, within=None*)  
→ Tuple

Given a head entity and a tail entity, return top k ranked relations.

$\text{argmax}_{\{r \in R\}} f(h,r,t)$ , where h, t in E.

## Parameter

head\_entity: List[str]

String representation of selected entities.

tail\_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

## Returns: Tuple

Highest K scores and entities

**predict\_missing\_tail\_entity** (*head\_entity: List[str] | str, relation: List[str] | str, within: List[str] = None*) → torch.FloatTensor

Given a head entity and a relation, return top k ranked entities

$\text{argmax}_{\{e \in E\}} f(h,r,e)$ , where h in E and r in R.

## Parameter

head\_entity: List[str]

String representation of selected entities.

tail\_entity: List[str]

String representation of selected entities.

## Returns: Tuple

scores

**predict** (\*, h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, within=None, logits=True) → torch.FloatTensor

### Parameters

- **logits**
- **h**
- **r**
- **t**
- **within**

**predict\_topk** (\*, h: List[str] = None, r: List[str] = None, t: List[str] = None, topk: int = 10, within: List[str] = None)

Predict missing item in a given 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.

k: int

Highest ranked k item.

## Returns: Tuple

Highest K scores and items

**triple\_score** (*h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, logits=False*)  
→ torch.FloatTensor

Predict triple score

## Parameter

head\_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail\_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

## Returns: Tuple

pytorch tensor of triple score

**t\_norm** (*tens\_1: torch.Tensor, tens\_2: torch.Tensor, tnorm: str = 'min'*) → torch.Tensor

**tensor\_t\_norm** (*subquery\_scores: torch.FloatTensor, tnorm: str = 'min'*) → torch.FloatTensor

Compute T-norm over  $[0,1]^{n \times d}$  where n denotes the number of hops and d denotes number of entities

**t\_conorm** (*tens\_1: torch.Tensor, tens\_2: torch.Tensor, tconorm: str = 'min'*) → torch.Tensor

**negnorm** (*tens\_1: torch.Tensor, lambda\_: float, neg\_norm: str = 'standard'*) → torch.Tensor

**return\_multi\_hop\_query\_results** (*aggregated\_query\_for\_all\_entities, k: int, only\_scores*)

**single\_hop\_query\_answering** (*query: tuple, only\_scores: bool = True, k: int = None*)

**answer\_multi\_hop\_query** (*query\_type: str = None,*  
*query: Tuple[str | Tuple[str, str], Ellipsis] = None,*  
*queries: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, tnorm: str = 'prod',*  
*neg\_norm: str = 'standard', lambda\_: float = 0.0, k: int = 10, only\_scores=False*)  
→ List[Tuple[str, torch.Tensor]]

# @TODO: Refactoring is needed # @TODO: Score computation for each query type should be done in a static function

Find an answer set for EPFO queries including negation and disjunction

## Parameter

query\_type: str The type of the query, e.g., “2p”.

query: Union[str, Tuple[str, Tuple[str, str]]] The query itself, either a string or a nested tuple.

queries: List of Tuple[Union[str, Tuple[str, str]], ...]

tnorm: str The t-norm operator.

neg\_norm: str The negation norm.

**lambda\_**: float lambda parameter for sugeno and yager negation norms

k: int The top-k substitutions for intermediate variables.

### returns

- *List[Tuple[str, torch.Tensor]]*
- *Entities and corresponding scores sorted in the descening order of scores*

**find\_missing\_triples** (*confidence: float, entities: List[str] = None, relations: List[str] = None, topk: int = 10, at\_most: int = sys.maxsize*) → Set

Find missing triples

Iterative over a set of entities E and a set of relation R :

orall e in E and orall r in R f(e,r,x)

Return (e,r,x)

otin G and f(e,r,x) > confidence

confidence: float

A threshold for an output of a sigmoid function given a triple.

topk: int

Highest ranked k item to select triples with f(e,r,x) > confidence .

at\_most: int

Stop after finding at\_most missing triples

{(e,r,x) | f(e,r,x) > confidence land (e,r,x)}

otin G

**deploy** (*share: bool = False, top\_k: int = 10*)

**train\_triples** (*h: List[str], r: List[str], t: List[str], labels: List[float], iteration=2, optimizer=None*)

**train\_k\_vs\_all** (*h, r, iteration=1, lr=0.001*)

Train k vs all :param head\_entity: :param relation: :param iteration: :param lr: :return:

**train** (*kg, lr=0.1, epoch=10, batch\_size=32, neg\_sample\_ratio=10, num\_workers=1*) → None

Retrained a pretrain model on an input KG via negative sampling.



## dicee.query\_generator

### Classes

---

*QueryGenerator*

---

### Functions

---

*save\_pickle*(\*[, data, file\_path])

*load\_pickle*([file\_path])

---

### Module Contents

`dicee.query_generator.save_pickle` (\*, data: object = None, file\_path=str)

`dicee.query_generator.load_pickle` (file\_path=str)

**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)

**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.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.static\_funcs

## Classes

<i>CMult</i>	CI_(0,0) => Real Numbers
<i>Pyke</i>	A Physical Embedding Model for Knowledge Graphs
<i>DistMult</i>	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
<i>KeciBase</i>	Without learning dimension scaling
<i>Keci</i>	Base class for all neural network modules.
<i>TransE</i>	Translating Embeddings for Modeling
<i>DeCaL</i>	Base class for all neural network modules.
<i>DualE</i>	Dual Quaternion Knowledge Graph Embeddings ( <a href="https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657">https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657</a> )
<i>Complex</i>	Base class for all neural network modules.
<i>AConEx</i>	Additive Convolutional ComplEx Knowledge Graph Embeddings
<i>AConvo</i>	Additive Convolutional Octonion Knowledge Graph Embeddings
<i>AConvQ</i>	Additive Convolutional Quaternion Knowledge Graph Embeddings
<i>ConvQ</i>	Convolutional Quaternion Knowledge Graph Embeddings
<i>Convo</i>	Base class for all neural network modules.
<i>ConEx</i>	Convolutional ComplEx Knowledge Graph Embeddings
<i>QMult</i>	Base class for all neural network modules.
<i>OMult</i>	Base class for all neural network modules.
<i>Shallom</i>	A shallow neural model for relation prediction ( <a href="https://arxiv.org/abs/2101.09090">https://arxiv.org/abs/2101.09090</a> )
<i>LFMult</i>	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
<i>PykeenKGE</i>	A class for using knowledge graph embedding models implemented in Pykeen
<i>Byte</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.

## Functions

<i>create_recipriocal_triples(x)</i>	Add inverse triples into dask dataframe
<i>get_er_vocab(data[, file_path])</i>	
<i>get_re_vocab(data[, file_path])</i>	
<i>get_ee_vocab(data[, file_path])</i>	
<i>timeit(func)</i>	
<i>save_pickle(*[, data, file_path])</i>	

continues on next page

Table 2 – continued from previous page

<code>load_pickle([file_path])</code>	
<code>select_model(args[, is_continual_training, storage_path])</code>	
<code>load_model(→ Tuple[object, Tuple[dict, dict]])</code>	Load weights and initialize pytorch module from namespace arguments
<code>load_model_ensemble(...)</code>	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
<code>save_numpy_ndarray(*, data, file_path)</code>	
<code>numpy_data_type_changer(→ numpy.ndarray)</code>	Detect most efficient data type for a given triples
<code>save_checkpoint_model(→ None)</code>	Store Pytorch model into disk
<code>store(→ None)</code>	Store trained_model model and save embeddings into csv file.
<code>add_noisy_triples(→ pandas.DataFrame)</code>	Add randomly constructed triples
<code>read_or_load_kg(args, cls)</code>	
<code>intialize_model(→ Tuple[object, str])</code>	
<code>load_json(→ dict)</code>	
<code>save_embeddings(→ None)</code>	Save it as CSV if memory allows.
<code>random_prediction(pre_trained_kge)</code>	
<code>deploy_triple_prediction(pre_trained_kge, str_subject, ...)</code>	
<code>deploy_tail_entity_prediction(pre_trained_]</code>	
<code>...)</code>	
<code>deploy_head_entity_prediction(pre_trained_]</code>	
<code>...)</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>	storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data
<code>exponential_function(→ torch.FloatTensor)</code>	
<code>load_numpy(→ numpy.ndarray)</code>	
<code>evaluate(entity_to_idx, scores, easy_answers, hard_answers)</code>	# @TODO: CD: Renamed this function
<code>download_file(url[, destination_folder])</code>	
<code>download_files_from_url(→ None)</code>	
	<b>param base_url</b>
<code>download_pretrained_model(→ str)</code>	

## Module Contents

**class** `dicee.static_funcs.CMult` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

`Cl_(0,0)` => Real Numbers

`Cl_(0,1)` =>

A multivector  $\mathbf{a} = a_0 + a_1 e_1$  A multivector  $\mathbf{b} = b_0 + b_1 e_1$

multiplication is isomorphic to the product of two complex numbers

$$\mathbf{a} \text{ imes } \mathbf{b} = a_0 b_0 + a_0 b_1 e_1 + a_1 b_1 e_1 e_1 \\ = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) e_1$$

`Cl_(2,0)` =>

A multivector  $\mathbf{a} = a_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 e_2$  A multivector  $\mathbf{b} = b_0 + b_1 e_1 + b_2 e_2 + b_{12} e_1 e_2$

$$\mathbf{a} \text{ imes } \mathbf{b} = a_0 b_0 + a_0 b_1 e_1 + a_0 b_2 e_2 + a_0 b_{12} e_1 e_2 \\ + a_1 b_0 e_1 + a_1 b_1 e_1 e_1 + \dots$$

`Cl_(0,2)` => Quaternions

**clifford\_mul** (*x: torch.FloatTensor, y: torch.FloatTensor, p: int, q: int*) → tuple

Clifford multiplication  $Cl_{\{p,q\}}$  ( $\mathbb{R}$ )

$e_i^2 = +1$  for  $i \leq p$   $e_j^2 = -1$  for  $p < j \leq p+q$   $e_i e_j = -e_j e_i$  for  $i$

$e_j$

*x*: torch.FloatTensor with (n,d) shape

*y*: torch.FloatTensor with (n,d) shape

*p*: a non-negative integer  $p \geq 0$  *q*: a non-negative integer  $q \geq 0$

**score** (*head\_ent\_emb, rel\_ent\_emb, tail\_ent\_emb*)

**forward\_triples** (*x: torch.LongTensor*) → torch.FloatTensor

Compute batch triple scores

### Parameter

*x*: torch.LongTensor with shape n by 3

**rtype**

torch.LongTensor with shape n

**forward\_k\_vs\_all** (*x: torch.Tensor*) → torch.FloatTensor

Compute batch KvsAll triple scores

## Parameter

x: torch.LongTensor with shape n by 3

**rtype**

torch.LongTensor with shape n

**class** dicee.static\_funcs.**Pyke**(args)

Bases: *[dicee.models.base\\_model.BaseKGE](#)*

A Physical Embedding Model for Knowledge Graphs

**forward\_triples**(x: torch.LongTensor)

**Parameters**

**x**

**class** dicee.static\_funcs.**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>

**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.static\_funcs.**KeciBase**(args)

Bases: *[Keci](#)*

Without learning dimension scaling

**class** dicee.static\_funcs.**Keci**(args)

Bases: *[dicee.models.base\\_model.BaseKGE](#)*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)
```

(continues on next page)

```
def forward(self, x):
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

## Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

### compute\_sigma\_pp (*hp, rp*)

Compute  $\text{sigma}_{\{pp\}} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k$

$\text{sigma}_{\{pp\}}$  captures the interactions between along  $p$  bases For instance, let  $p$   $e_1, e_2, e_3$ , we compute interactions between  $e_1 e_2, e_1 e_3$ , and  $e_2 e_3$  This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

    for k in range(i + 1, p):

        results.append(hp[:, :, i] \* rp[:, :, k] - hp[:, :, k] \* rp[:, :, i])

sigma\_pp = torch.stack(results, dim=2) assert sigma\_pp.shape == (b, r, int((p \* (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all  $p$ , e.g.,  $e_1 e_1, e_1 e_2, e_1 e_3$ ,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals:  $e_1 e_2, e_1 e_3, e_2 e_3$ .

### compute\_sigma\_qq (*hq, rq*)

Compute  $\text{sigma}_{\{qq\}} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k$   $\text{sigma}_{\{q\}}$  captures the interactions between along  $q$  bases For instance, let  $q$   $e_1, e_2, e_3$ , we compute interactions between  $e_1 e_2, e_1 e_3$ , and  $e_2 e_3$  This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

    for k in range(j + 1, q):

        results.append(hq[:, :, j] \* rq[:, :, k] - hq[:, :, k] \* rq[:, :, j])

sigma\_qq = torch.stack(results, dim=2) assert sigma\_qq.shape == (b, r, int((q \* (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all  $p$ , e.g.,  $e_1 e_1, e_1 e_2, e_1 e_3$ ,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

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*)

$\sum_{i=1}^{p-1} \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j$

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 (h0, hp, hq, r0, rp, rq)
    Multiplying a base vector with its scalar coefficient

clifford_multiplication (h0, hp, hq, r0, rp, rq)
    Compute our CL multiplication

    
$$h = h_0 + \sum_{i=1}^p h_i e_i + \sum_{j=p+1}^{p+q} h_j e_j$$


$$r = r_0 + \sum_{i=1}^p r_i e_i + \sum_{j=p+1}^{p+q} r_j e_j$$


    
$$e_i^2 = +1 \text{ for } i \leq p, e_j^2 = -1 \text{ for } p < j \leq p+q, e_i e_j = -e_j e_i \text{ for } i \neq j$$


    
$$h r = \sigma_0 + \sigma_p + \sigma_q + \sigma_{pp} + \sigma_{pq} + \sigma_{qq}$$

    where
    (1)  $\sigma_0 = h_0 r_0 + \sum_{i=1}^p (h_0 r_i - h_i r_0) e_i - \sum_{j=p+1}^{p+q} (h_j r_j) e_j$ 
    (2)  $\sigma_p = \sum_{i=1}^p (h_0 r_i + h_i r_0) e_i$ 
    (3)  $\sigma_q = \sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j$ 
    (4)  $\sigma_{pp} = \sum_{i=1}^p \sum_{k=i+1}^p (h_i r_k - h_k r_i) e_i e_k$ 
    (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



**forward\_k\_vs\_sample** (*x: torch.LongTensor, target\_entity\_idx: torch.LongTensor*)  
→ 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

### Parameter

*x*: torch.LongTensor with (n,2) shape

**rtype**

torch.FloatTensor with (n, **IE**) 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.static\_funcs.**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>

**score** (*head\_ent\_emb, rel\_ent\_emb, tail\_ent\_emb*)

**forward\_k\_vs\_all** (*x: torch.Tensor*) → torch.FloatTensor

**class** dicee.static\_funcs.**DeCaL** (*args*)

Bases: *dicee.models.base\_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)
```

(continues on next page)

```
def forward(self, x):
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

### Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**forward\_triples** (*x: torch.Tensor*) → *torch.FloatTensor*

### Parameter

*x*: *torch.LongTensor* with (*n*,) shape

**rtype**

*torch.FloatTensor* with (*n*) shape

**cl\_pqr** (*a: torch.tensor*) → *torch.tensor*

Input: *tensor(batch\_size, emb\_dim)* → output: *tensor* with *1+p+q+r* components with size (*batch\_size, emb\_dim/(1+p+q+r)*) each.

1) takes a tensor of size (*batch\_size, emb\_dim*), split it into *1 + p + q + r* components, hence *1+p+q+r* must be a divisor of the *emb\_dim*. 2) Return a list of the *1+p+q+r* components vectors, each are tensors of size (*batch\_size, emb\_dim/(1+p+q+r)*)

**compute\_sigmas\_single** (*list\_h\_emb, list\_r\_emb, list\_t\_emb*)

here we compute all the sums with no others vectors interaction taken with the scalar product with *t*, that is,

$$s0 = h_0 r_0 t_0 s1 = \sum_{i=1}^p h_i r_i t_0 s2 = \sum_{j=p+1}^{p+q} h_j r_j t_0 s3 = \sum_{i=1}^q (h_0 r_i t_i + h_i r_0 t_i) s4 = \sum_{i=p+1}^{p+q} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+1}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i)$$

and return:

$$\sigma_0 t = \sigma_0 \cdot t_0 = s0 + s1 - s2 s3, s4 \text{ and } s5$$

**compute\_sigmas\_multivect** (*list\_h\_emb, list\_r\_emb*)

Here we compute and return all the sums with vectors interaction for the same and different bases.

For same bases vectors interaction we have

$$\sigma_p p = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (h_i r_{i'} - h_{i'} r_i) (\text{model the interactions between } e_i \text{ and } e_{i'} \text{ for } 1 \leq i, i' \leq p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{j'} r_j)$$

For different base vector interactions, we have

$$\sigma_{pq} = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) (\text{interactions between } e_i \text{ and } e_j \text{ for } 1 \leq i \leq p \text{ and } p+1 \leq j \leq p+q) \sigma_p r = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) \sigma_p r$$

**forward\_k\_vs\_all** (*x*: torch.Tensor) → torch.FloatTensor

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations
- (2) Construct head entity and relation embeddings according to  $Cl_{\{p,q,r\}}(\mathbb{R}^d)$ .
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

forward\_k\_vs\_with\_explicit and this functions are identical Parameter ——— *x*: torch.LongTensor with (n, ) shape :rtype: torch.FloatTensor with (n, **IE**) shape

**apply\_coefficients** (*h0, hp, hq, hk, r0, rp, rq, rk*)

Multiplying a base vector with its scalar coefficient

**construct\_cl\_multivector** (*x*: torch.FloatTensor, *re*: int, *p*: int, *q*: int, *r*: int)  
→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors  $Cl_{\{p,q,r\}}(\mathbb{R}^d)$

## Parameter

*x*: torch.FloatTensor with (n,d) shape

**returns**

- **a0** (torch.FloatTensor)
- **ap** (torch.FloatTensor)
- **aq** (torch.FloatTensor)
- **ar** (torch.FloatTensor)

**compute\_sigma\_pp** (*hp, rp*)

Compute .. math:

$$\sigma_{pp} = \sum_{i=1}^p \sum_{j=i+1}^p (x_{iy_i} - x_{ij} y_j)$$

$\sigma_{pp}$  captures the interactions between along *p* bases For instance, let *p* e<sub>1</sub>, e<sub>2</sub>, e<sub>3</sub>, we compute interactions between e<sub>1</sub> e<sub>2</sub>, e<sub>1</sub> e<sub>3</sub>, and e<sub>2</sub> e<sub>3</sub> This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

**for k in range(i + 1, p):**

        results.append(hp[:, :, i] \* rp[:, :, k] - hp[:, :, k] \* rp[:, :, i])

sigma\_pp = torch.stack(results, dim=2) assert sigma\_pp.shape == (b, r, int((p \* (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all *p*, e.g., e<sub>1</sub>e<sub>1</sub>, e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>,

e<sub>2</sub>e<sub>1</sub>, e<sub>2</sub>e<sub>2</sub>, e<sub>2</sub>e<sub>3</sub>, e<sub>3</sub>e<sub>1</sub>, e<sub>3</sub>e<sub>2</sub>, e<sub>3</sub>e<sub>3</sub>

Then select the triangular matrix without diagonals: e<sub>1</sub>e<sub>2</sub>, e<sub>1</sub>e<sub>3</sub>, e<sub>2</sub>e<sub>3</sub>.

**compute\_sigma\_qq** (*hq, rq*)

Compute

$$\sigma_{q,q}^* = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (x_j y_{j'} - x_{j'} y_j) \quad Eq.16$$

$\sigma_{q,q}$  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.static_funcs.DualE (args)
    Bases: dicee.models.base_model.BaseKGE
    Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)
    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)

```
class dicee.static_funcs.Complex(args)
```

Bases: *`dicee.models.base_model.BaseKGE`*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

## Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**static score** (*head\_ent\_emb: torch.FloatTensor, rel\_ent\_emb: torch.FloatTensor, tail\_ent\_emb: torch.FloatTensor*)

**static k\_vs\_all\_score** (*emb\_h: torch.FloatTensor, emb\_r: torch.FloatTensor, emb\_E: torch.FloatTensor*)

## Parameters

- **emb\_h**
- **emb\_r**
- **emb\_E**

**forward\_k\_vs\_all** (*x: torch.LongTensor*)  $\rightarrow$  torch.FloatTensor

```

class dicee.static_funcs.AConEx (args)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional ComplEx Knowledge Graph Embeddings

    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.static_funcs.AConvo (args: dict)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional Octonion Knowledge Graph Embeddings

    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.static_funcs.AConvQ (args)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional Quaternion Knowledge Graph Embeddings

    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.static_funcs.ConvQ (args)
    Bases: dicee.models.base_model.BaseKGE
    Convolutional Quaternion Knowledge Graph Embeddings

```

**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.static\_funcs.**ConvO** (*args: dict*)

Bases: *dicee.models.base\_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

**Variables**

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**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.static_funcs.ConEx (args)
```

Bases: `dicee.models.base_model.BaseKGE`

Convolutional ComplEx Knowledge Graph Embeddings

```
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.static_funcs.QMult (args)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

## Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

```
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,l Entities)

**class** dicee.static\_funcs.**OMult** (*args*)

Bases: *dicee.models.base\_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
```

(continues on next page)

```

super().__init__()
self.conv1 = nn.Conv2d(1, 20, 5)
self.conv2 = nn.Conv2d(20, 20, 5)

def forward(self, x):
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

### Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**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.static_funcs.Shallom` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

A shallow neural model for relation prediction (<https://arxiv.org/abs/2101.09090>)

**get\_embeddings** ()  $\rightarrow$  Tuple[numpy.ndarray, None]

**forward\_k\_vs\_all** (*x*)  $\rightarrow$  torch.FloatTensor

**forward\_triples** (*x*)  $\rightarrow$  torch.FloatTensor

### Parameters

**x**

### Returns

**class** `dicee.static_funcs.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 different scoring function as in the paper to evaluate the score. We also consider combining with Neural Networks.

**forward\_triples** (*idx\_triple*)

### Parameters

**x**

**construct\_multi\_coeff** (*x*)

**poly\_NN** (*x*, *coefh*, *coefr*, *coeft*)  
Constructing a 2 layers NN to represent the embeddings.  $h = \text{sigma}(wh^T x + bh)$ ,  $r = \text{sigma}(wr^T x + br)$ ,  
 $t = \text{sigma}(wt^T x + bt)$

**linear** (*x*, *w*, *b*)

**scalar\_batch\_NN** (*a*, *b*, *c*)  
element wise multiplication between a,b and c: Inputs : a, b, c ==> torch.tensor of size batch\_size x m x d  
Output : a tensor of size batch\_size x d

**tri\_score** (*coeff\_h*, *coeff\_r*, *coeff\_t*)  
this part implement the trilinear scoring techniques:  

$$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i b_j c_k\} \{1+(i+j+k)d\}$$

1. generate the range for i,j and k from [0 d-1]
2. perform  $\text{dfrac}\{a_i b_j c_k\} \{1+(i+j+k)d\}$  in parallel for every batch
3. take the sum over each batch

**vtp\_score** (*h*, *r*, *t*)  
this part implement the vector triple product scoring techniques:  

$$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i c_j b_k - b_i c_j a_k\} \{(1+(i+j)d)(1+k)\}$$

1. generate the range for i,j and k from [0 d-1]
2. Compute the first and second terms of the sum
3. Multiply with then denominator and take the sum
4. take the sum over each batch

**comp\_func** (*h*, *r*, *t*)  
this part implement the function composition scoring techniques: i.e.  $\text{score} = \langle h \circ r, t \rangle$

**polynomial** (*coeff*, *x*, *degree*)  
This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,...d] and return a vector tensor ( $\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d$ ,  
 $\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d$ )

**pop** (*coeff*, *x*, *degree*)  
This function allow us to evaluate the composition of two polynomes without for loops :) it takes a matrix tensor of coefficients (coeff), a matrix tensor of points x and range of integer [0,1,...d]  
and return a tensor ( $\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d$ ,  
 $\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d$ )

**class** dicee.static\_funcs.**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:

**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.static\_funcs.**BytE** (\*args, \*\*kwargs)

Bases: *[dicee.models.base\\_model.BaseKGE](#)*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

### Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**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.static_funcs.BaseKGE` (*args: dict*)

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

## Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**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

- $\mathbf{x}$
- $y\_idx$
- **ordered\_bpe\_entities**

**forward\_triples** ( $x$ : *torch.LongTensor*)  $\rightarrow$  *torch.Tensor*

#### Parameters

$\mathbf{x}$

**forward\_k\_vs\_all** ( $*args$ ,  $**kwargs$ )

**forward\_k\_vs\_sample** ( $*args$ ,  $**kwargs$ )

**get\_triple\_representation** ( $idx\_hrt$ )

**get\_head\_relation\_representation** ( $indexed\_triple$ )

**get\_sentence\_representation** ( $x$ : *torch.LongTensor*)

#### Parameters

- ( $\mathbf{b}$  ( $x$  *shape*))
- 3
- $\mathbf{t}$ )

**get\_bpe\_head\_and\_relation\_representation** ( $x$ : *torch.LongTensor*)  
 $\rightarrow$  *Tuple[torch.FloatTensor, torch.FloatTensor]*

#### Parameters

$\mathbf{x}$  ( $B \times 2 \times T$ )

**get\_embeddings** ()  $\rightarrow$  *Tuple[numpy.ndarray, numpy.ndarray]*

`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.select_model (args: dict, is_continual_training: bool = None,
                                storage_path: str = None)

dicee.static_funcs.load_model (path_of_experiment_folder: str, model_name='model.pt',
                                verbose=0) → Tuple[object, Tuple[dict, dict]]
    Load weights and initialize pytorch module from namespace arguments

dicee.static_funcs.load_model_ensemble (path_of_experiment_folder: str)
    → Tuple[dicee.models.base_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]
    Construct Ensemble Of weights and initialize pytorch module from namespace arguments

    (1) Detect models under given path
    (2) Accumulate parameters of detected models
    (3) Normalize parameters
    (4) Insert (3) into model.

dicee.static_funcs.save_numpy_ndarray (*, data: numpy.ndarray, file_path: str)

dicee.static_funcs.numpy_data_type_changer (train_set: numpy.ndarray, num: int)
    → numpy.ndarray
    Detect most efficient data type for a given triples :param train_set: :param num: :return:

dicee.static_funcs.save_checkpoint_model (model, path: str) → None
    Store Pytorch model into disk

dicee.static_funcs.store (trainer, trained_model, model_name: str = 'model',
                        full_storage_path: str = None, save_embeddings_as_csv=False) → None
    Store trained_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param
    full_storage_path: path to save parameters. :param model_name: string representation of the name of the model.
    :param trained_model: an instance of BaseKGE see core.models.base_model . :param save_embeddings_as_csv:
    for easy access of embeddings. :return:

dicee.static_funcs.add_noisy_triples (train_set: pandas.DataFrame, add_noise_rate: float)
    → pandas.DataFrame
    Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

dicee.static_funcs.read_or_load_kg (args, cls)

dicee.static_funcs.intialize_model (args: dict, verbose=0) → Tuple[object, str]

dicee.static_funcs.load_json (p: str) → dict

dicee.static_funcs.save_embeddings (embeddings: numpy.ndarray, indexes, path: str) → None
    Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:

dicee.static_funcs.random_prediction (pre_trained_kge)

dicee.static_funcs.deploy_triple_prediction (pre_trained_kge, str_subject, str_predicate,
                                            str_object)

dicee.static_funcs.deploy_tail_entity_prediction (pre_trained_kge, str_subject,
                                                  str_predicate, top_k)

```

```

dicee.static_funcs.deploy_head_entity_prediction (pre_trained_kge, str_object,
str_predicate, top_k)

dicee.static_funcs.deploy_relation_prediction (pre_trained_kge, str_subject, str_object,
top_k)

dicee.static_funcs.vocab_to_parquet (vocab_to_idx, name, path_for_serialization, print_into)

dicee.static_funcs.create_experiment_folder (folder_name='Experiments')

dicee.static_funcs.continual_training_setup_executor (executor) → None
    storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data
    full_storage_path:str A path leading to a subdirectory containing KGE related data

dicee.static_funcs.exponential_function (x: numpy.ndarray, lam: float,
ascending_order=True) → torch.FloatTensor

dicee.static_funcs.load_numpy (path) → numpy.ndarray

dicee.static_funcs.evaluate (entity_to_idx, scores, easy_answers, hard_answers)
    # @TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

dicee.static_funcs.download_file (url, destination_folder='.')

dicee.static_funcs.download_files_from_url (base_url: str, destination_folder='.') → None

```

#### Parameters

- **base\_url** (e.g. “<https://files.dice-research.org/projects/DiceEmbeddings/KINSHIP-Keci-dim128-epoch256-KvsAll>”)
- **destination\_folder** (e.g. “*KINSHIP-Keci-dim128-epoch256-KvsAll*”)

```

dicee.static_funcs.download_pretrained_model (url: str) → str

```

## `dicee.static_funcs_training`

### Functions

```

evaluate_lp(model, triple_idx, num_entities, Evaluate model in a standard link prediction task
er_vocab, ...)

```

```

evaluate_bpe_lp(model, triple_idx, ..., info)

```

```

efficient_zero_grad(model)

```

## Module Contents

```
dicee.static_funcs_training.evaluate_lp(model, triple_idx, num_entities,  
    er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List], info='Eval Starts')
```

Evaluate model in a standard link prediction task

for each triple the rank is computed by taking the mean of the filtered missing head entity rank and the filtered missing tail entity rank :param model: :param triple\_idx: :param info: :return:

```
dicee.static_funcs_training.evaluate_bpe_lp(model, triple_idx: List[Tuple],  
    all_bpe_shaped_entities, er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List],  
    info='Eval Starts')
```

```
dicee.static_funcs_training.efficient_zero_grad(model)
```

## dicee.static\_preprocess\_funcs

### Attributes

---

```
enable_log
```

---

### Functions

---

```
sanity_checking_with_arguments(args)
```

```
timeit(func)
```

```
preprocesses_input_args(args)
```

Sanity Checking in input arguments

```
create_constraints(→ Tuple[dict, dict, dict,  
dict])
```

(1) Extract domains and ranges of relations

```
get_er_vocab(data)
```

```
get_re_vocab(data)
```

```
get_ee_vocab(data)
```

```
mapping_from_first_two_cols_to_third(tri
```

---

## Module Contents

`dicee.static_preprocess_funcs.sanity_checking_with_arguments (args)`

`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)`

## 13.3 Attributes

`__version__`

## 13.4 Classes

<i>CMult</i>	Cl <sub>(0,0)</sub> => Real Numbers
<i>Pyke</i>	A Physical Embedding Model for Knowledge Graphs
<i>DistMult</i>	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
<i>KeciBase</i>	Without learning dimension scaling
<i>Keci</i>	Base class for all neural network modules.
<i>TransE</i>	Translating Embeddings for Modeling
<i>DeCaL</i>	Base class for all neural network modules.
<i>DualE</i>	Dual Quaternion Knowledge Graph Embeddings ( <a href="https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657">https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657</a> )
<i>Complex</i>	Base class for all neural network modules.
<i>AConEx</i>	Additive Convolutional Complex Knowledge Graph Embeddings
<i>AConvo</i>	Additive Convolutional Octonion Knowledge Graph Embeddings

continues on next page

Table 3 – continued from previous page

<i>AConvQ</i>	Additive Convolutional Quaternion Knowledge Graph Embeddings
<i>ConvQ</i>	Convolutional Quaternion Knowledge Graph Embeddings
<i>ConvO</i>	Base class for all neural network modules.
<i>ConEx</i>	Convolutional ComplEx Knowledge Graph Embeddings
<i>QMult</i>	Base class for all neural network modules.
<i>OMult</i>	Base class for all neural network modules.
<i>Shallom</i>	A shallow neural model for relation prediction ( <a href="https://arxiv.org/abs/2101.09090">https://arxiv.org/abs/2101.09090</a> )
<i>LFMult</i>	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
<i>PykeenKGE</i>	A class for using knowledge graph embedding models implemented in Pykeen
<i>Byte</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>DICE_Trainer</i>	DICE_Trainer implement
<i>KGE</i>	Knowledge Graph Embedding Class for interactive usage of pre-trained models
<i>Execute</i>	A class for Training, Retraining and Evaluation a model.
<i>BPE_NegativeSamplingDataset</i>	An abstract class representing a Dataset.
<i>MultiLabelDataset</i>	An abstract class representing a Dataset.
<i>MultiClassClassificationDataset</i>	Dataset for the 1vsALL training strategy
<i>OnevsAllDataset</i>	Dataset for the 1vsALL training strategy
<i>KvsAll</i>	Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.
<i>AllvsAll</i>	Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.
<i>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>	

## 13.5 Functions

<i>create_recipriocal_triples(x)</i>	Add inverse triples into dask dataframe
<i>get_er_vocab(data[, file_path])</i>	
<i>get_re_vocab(data[, file_path])</i>	
<i>get_ee_vocab(data[, file_path])</i>	
<i>timeit(func)</i>	
<i>save_pickle(*[, data, file_path])</i>	

continues on next page

Table 4 – continued from previous page

<code>load_pickle([file_path])</code>	
<code>select_model(args[, is_continual_training, storage_path])</code>	
<code>load_model(→ Tuple[object, Tuple[dict, dict]])</code>	Load weights and initialize pytorch module from namespace arguments
<code>load_model_ensemble(...)</code>	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
<code>save_numpy_ndarray(*, data, file_path)</code>	
<code>numpy_data_type_changer(→ numpy.ndarray)</code>	Detect most efficient data type for a given triples
<code>save_checkpoint_model(→ None)</code>	Store Pytorch model into disk
<code>store(→ None)</code>	Store trained_model model and save embeddings into csv file.
<code>add_noisy_triples(→ pandas.DataFrame)</code>	Add randomly constructed triples
<code>read_or_load_kg(args, cls)</code>	
<code>intialize_model(→ Tuple[object, str])</code>	
<code>load_json(→ dict)</code>	
<code>save_embeddings(→ None)</code>	Save it as CSV if memory allows.
<code>random_prediction(pre_trained_kge)</code>	
<code>deploy_triple_prediction(pre_trained_kge, str_subject, ...)</code>	
<code>deploy_tail_entity_prediction(pre_trained_]</code>	
<code>...)</code>	
<code>deploy_head_entity_prediction(pre_trained_]</code>	
<code>...)</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>	storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data
<code>exponential_function(→ torch.FloatTensor)</code>	
<code>load_numpy(→ numpy.ndarray)</code>	
<code>evaluate(entity_to_idx, scores, easy_answers, hard_answers)</code>	# @TODO: CD: Renamed this function
<code>download_file(url[, destination_folder])</code>	
<code>download_files_from_url(→ None)</code>	
	<b>param base_url</b>
<code>download_pretrained_model(→ str)</code>	

continues on next page

Table 4 – continued from previous page

<code>mapping_from_first_two_cols_to_third(tr</code>	
<code>timeit(func)</code>	
<code>load_pickle([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>	

## 13.6 Package Contents

**class** `dicee.CMult` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

`Cl_(0,0)` => Real Numbers

`Cl_(0,1)` =>

A multivector  $\mathbf{a} = a_0 + a_1 e_1$  A multivector  $\mathbf{b} = b_0 + b_1 e_1$

multiplication is isomorphic to the product of two complex numbers

$\mathbf{a} \mathbf{b} = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) e_1$   
 $= (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) e_1$

`Cl_(2,0)` =>

A multivector  $\mathbf{a} = a_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 e_2$  A multivector  $\mathbf{b} = b_0 + b_1 e_1 + b_2 e_2 + b_{12} e_1 e_2$

$\mathbf{a} \mathbf{b} = (a_0 b_0 + a_0 b_{12} e_1 e_2 + a_1 b_1 + a_1 b_{12} e_2 + a_2 b_2 + a_2 b_{12} e_1 + a_{12} b_0 + a_{12} b_1 e_1 + a_{12} b_2 e_2 + a_{12} b_{12})$   
 $\bullet a_1 b_0 e_1 + a_1 b_1 e_1 e_1 ..$

`Cl_(0,2)` => Quaternions

**clifford\_mul** (*x: torch.FloatTensor, y: torch.FloatTensor, p: int, q: int*) → tuple

Clifford multiplication  $Cl_{\{p,q\}}$  ( $\mathbb{R}$ )

$e_i^2 = +1$  for  $i \leq p$   $e_j^2 = -1$  for  $p < j \leq p+q$   $e_i e_j = -e_j e_i$  for  $i$

$e_j$

*x*: torch.FloatTensor with (n,d) shape

*y*: torch.FloatTensor with (n,d) shape

*p*: a non-negative integer  $p \geq 0$  *q*: a non-negative integer  $q \geq 0$

**score** (*head\_ent\_emb, rel\_ent\_emb, tail\_ent\_emb*)

**forward\_triples** (*x: torch.LongTensor*) → torch.FloatTensor

Compute batch triple scores

## Parameter

x: torch.LongTensor with shape n by 3

**rtype**

torch.LongTensor with shape n

**forward\_k\_vs\_all** (x: torch.Tensor) → torch.FloatTensor

Compute batch KvsAll triple scores

## Parameter

x: torch.LongTensor with shape n by 3

**rtype**

torch.LongTensor with shape n

**class** dicee.Pyke (args)

Bases: *dicee.models.base\_model.BaseKGE*

A Physical Embedding Model for Knowledge Graphs

**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>

**k\_vs\_all\_score** (emb\_h: torch.FloatTensor, emb\_r: torch.FloatTensor, emb\_E: torch.FloatTensor)

**Parameters**

- **emb\_h**
- **emb\_r**
- **emb\_E**

**forward\_k\_vs\_all** (x: torch.LongTensor)

**forward\_k\_vs\_sample** (x: torch.LongTensor, target\_entity\_idx: torch.LongTensor)

**score** (h, r, t)

**class** dicee.KeciBase (args)

Bases: *Keci*

Without learning dimension scaling

**class** dicee.Keci (args)

Bases: *dicee.models.base\_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:



```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

### Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

### compute\_sigma\_pp (*hp, rp*)

Compute  $\text{sigma\_pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_{i \ r_k} - h_{k \ r_i}) e_i e_k$

$\text{sigma\_pp}$  captures the interactions between along  $p$  bases For instance, let  $p$   $e_1, e_2, e_3$ , we compute interactions between  $e_1 e_2, e_1 e_3$ , and  $e_2 e_3$  This can be implemented with a nested two for loops

results = [] for i in range(p - 1):

    for k in range(i + 1, p):

        results.append(hp[:, :, i] \* rp[:, :, k] - hp[:, :, k] \* rp[:, :, i])

sigma\_pp = torch.stack(results, dim=2) assert sigma\_pp.shape == (b, r, int((p \* (p - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all  $p$ , e.g.,  $e_1 e_1, e_1 e_2, e_1 e_3$ ,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals:  $e_1 e_2, e_1 e_3, e_2 e_3$ .

### compute\_sigma\_qq (*hq, rq*)

Compute  $\text{sigma\_qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_{j \ r_k} - h_{k \ r_j}) e_j e_k \text{sigma\_q}$  captures the interactions between along  $q$  bases For instance, let  $q$   $e_1, e_2, e_3$ , we compute interactions between  $e_1 e_2, e_1 e_3$ , and  $e_2 e_3$  This can be implemented with a nested two for loops

results = [] for j in range(q - 1):

    for k in range(j + 1, q):

        results.append(hq[:, :, j] \* rq[:, :, k] - hq[:, :, k] \* rq[:, :, j])

sigma\_qq = torch.stack(results, dim=2) assert sigma\_qq.shape == (b, r, int((q \* (q - 1)) / 2))

Yet, this computation would be quite inefficient. Instead, we compute interactions along all  $p$ , e.g.,  $e_1 e_1, e_1 e_2, e_1 e_3$ ,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals:  $e_1e_2, e_1e_3, e_2e_3$ .

**compute\_sigma\_pq** (\*, hp, hq, rp, rq)

$\text{sum}_{\{i=1\}^p} \text{sum}_{\{j=p+1\}^{p+q}} (h_{i \ r_j} - h_{j \ r_i}) e_i e_j$

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** (h0, hp, hq, r0, rp, rq)

Multiplying a base vector with its scalar coefficient

**clifford\_multiplication** (h0, hp, hq, r0, rp, rq)

Compute our CL multiplication

$h = h_0 + \text{sum}_{\{i=1\}^p} h_i e_i + \text{sum}_{\{j=p+1\}^{p+q}} h_j e_j$   
 $r = r_0 + \text{sum}_{\{i=1\}^p} r_i e_i + \text{sum}_{\{j=p+1\}^{p+q}} r_j e_j$

$e_i^2 = +1$  for  $i \leq p$   $e_j^2 = -1$  for  $p < j \leq p+q$   $e_i e_j = -e_j e_i$  for  $i$

$e_j$

$h \ r = \text{sigma}_0 + \text{sigma}_p + \text{sigma}_q + \text{sigma}_{\{pp\}} + \text{sigma}_{\{q\}} + \text{sigma}_{\{pq\}}$  where

(1)  $\text{sigma}_0 = h_0 r_0 + \text{sum}_{\{i=1\}^p} (h_0 r_i) e_i - \text{sum}_{\{j=p+1\}^{p+q}} (h_j r_j) e_j$

(2)  $\text{sigma}_p = \text{sum}_{\{i=1\}^p} (h_0 r_i + h_i r_0) e_i$

(3)  $\text{sigma}_q = \text{sum}_{\{j=p+1\}^{p+q}} (h_0 r_j + h_j r_0) e_j$

(4)  $\text{sigma}_{\{pp\}} = \text{sum}_{\{i=1\}^{p-1}} \text{sum}_{\{k=i+1\}^p} (h_i r_k - h_k r_i) e_i e_k$

(5)  $\text{sigma}_{\{qq\}} = \text{sum}_{\{j=1\}^{p+q-1}} \text{sum}_{\{k=j+1\}^{p+q}} (h_j r_k - h_k r_j) e_j e_k$

(6)  $\text{sigma}_{\{pq\}} = \text{sum}_{\{i=1\}^p} \text{sum}_{\{j=p+1\}^{p+q}} (h_i r_j - h_j r_i) e_i e_j$

**construct\_cl\_multivector** (x: torch.FloatTensor, r: int, p: int, q: int)

→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors  $\text{Cl}_{\{p,q\}}(\mathbb{R}^d)$

## Parameter

x: torch.FloatTensor with (n,d) shape

**returns**

- **a0** (torch.FloatTensor with (n,r) shape)
- **ap** (torch.FloatTensor with (n,r,p) shape)
- **aq** (torch.FloatTensor with (n,r,q) shape)

**forward\_k\_vs\_with\_explicit** (x: torch.Tensor)

**k\_vs\_all\_score** (bpe\_head\_ent\_emb, bpe\_rel\_ent\_emb, E)

**forward\_k\_vs\_all** (x: torch.Tensor) → torch.FloatTensor

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations  $\mathbb{R}^d$ .
- (2) Construct head entity and relation embeddings according to  $\text{Cl}_{\{p,q\}}(\mathbb{R}^d)$ .

- (3) Perform CI multiplication
- (4) Inner product of (3) and all entity embeddings

`forward_k_vs_with_explicit` and this function are identical Parameter ——— `x`: `torch.LongTensor` with (n,2) shape :rtype: `torch.FloatTensor` with (n, **IEI**) shape

**forward\_k\_vs\_sample** (*x: torch.LongTensor, target\_entity\_idx: torch.LongTensor*)  
→ `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  $CI_{\{p,q\}}(\mathbb{R}^d)$ .
- (3) Perform CI multiplication
- (4) Inner product of (3) and all entity embeddings

### Parameter

`x`: `torch.LongTensor` with (n,2) shape

**rtype**  
`torch.FloatTensor` with (n, **IEI**) 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>

**score** (*head\_ent\_emb, rel\_ent\_emb, tail\_ent\_emb*)

**forward\_k\_vs\_all** (*x: torch.Tensor*) → `torch.FloatTensor`

**class** `dicee.DeCaL` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

### Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**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.

$$\sigma_p p = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (h_i r_{i'} - h_{i'} r_i) (\text{model the interactions between } e_i \text{ and } e_{i'} \text{ for } 1 \leq i, i' \leq p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'}$$
$$\sigma_p q = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) (\text{interactions between } e_i \text{ and } e_j \text{ for } 1 \leq i \leq p \text{ and } p+1 \leq j \leq p+q) \sigma_p r = \sum_{i=1}^p$$

- (1) Retrieve real-valued embedding vectors for heads and relations
- (2) Construct head entity and relation embeddings according to  $\text{Cl}_{\{p, q, r\}}(\mathbb{R}^d)$ .
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

### Multiplying a base vector with its scalar coefficient

Construct a batch of multivectors  $\text{Cl}_{\{p,q,r\}}(\mathbb{R}^d)$

- **a0** (*torch.FloatTensor*)
- **ap** (*torch.FloatTensor*)
- **aq** (*torch.FloatTensor*)
- **ar** (*torch.FloatTensor*)

$$\sigma_{p,p}^* = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (x_i y_{i'} - x_{i'} y_i)$$

```
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.DualE (args)
```

Bases: *dicee.models.base\_model.BaseKGE*

Dual Quaternion Knowledge Graph Embeddings (<https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657>)

```
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: torch.tensor)  $\rightarrow$  torch.tensor

Transpose function

Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)

**class** dicee.Complex (args)

Bases: *dicee.models.base\_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

## Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**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

class dicee.AConEx (args)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional ComplEx Knowledge Graph Embeddings
    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
    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
    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

```
residual_convolution(Q_1, Q_2)
```

```
forward_triples(indexed_triple: torch.Tensor) → torch.Tensor
```

**Parameters**

**x**

```
forward_k_vs_all(x: torch.Tensor)
```

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **Entities**) Given a batch of head entities and relations => shape (size of batch,|Entities|)

```
class dicee.ConvO(args: dict)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

**Variables**

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

```
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

**residual\_convolution** (*C\_1*: Tuple[torch.Tensor, torch.Tensor],  
*C\_2*: Tuple[torch.Tensor, torch.Tensor]) → torch.FloatTensor

Compute residual score of two complex-valued embeddings. :param C\_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C\_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

**forward\_k\_vs\_all** (*x*: torch.Tensor) → torch.FloatTensor

**forward\_triples** (*x*: torch.Tensor) → torch.FloatTensor

**Parameters**

**x**

**forward\_k\_vs\_sample** (*x*: torch.Tensor, *target\_entity\_idx*: torch.Tensor)

**class** dicee.QMult (*args*)

Bases: *dicee.models.base\_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

## Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**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 to nest them in a tree structure. You can assign the submodules as regular attributes:

```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

### Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**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.Shallom** (*args*)

Bases: *dicee.models.base\_model.BaseKGE*

A shallow neural model for relation prediction (<https://arxiv.org/abs/2101.09090>)

**get\_embeddings** ()  $\rightarrow$  Tuple[numpy.ndarray, None]

**forward\_k\_vs\_all** (*x*)  $\rightarrow$  torch.FloatTensor

**forward\_triples** (*x*)  $\rightarrow$  torch.FloatTensor

### Parameters

**x**

### Returns

**class dicee.LFMult** (*args*)

Bases: *dicee.models.base\_model.BaseKGE*

Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:  $f(x) = \sum_{i=0}^{d-1} a_i x^{i \% d}$  and use the three different scoring function as in the paper to evaluate the score. We also consider combining with Neural Networks.

**forward\_triples** (*idx\_triple*)

#### Parameters

**x**

**construct\_multi\_coeff** (*x*)

**poly\_NN** (*x, coefh, coefr, coeft*)

Constructing a 2 layers NN to represent the embeddings.  $h = \sigma(w_h^T x + b_h)$ ,  $r = \sigma(w_r^T x + b_r)$ ,  
 $t = \sigma(w_t^T x + b_t)$

**linear** (*x, w, b*)

**scalar\_batch\_NN** (*a, b, c*)

element wise multiplication between a,b and c: Inputs : a, b, c ==> torch.tensor of size batch\_size x m x d  
Output : a tensor of size batch\_size x d

**tri\_score** (*coeff\_h, coeff\_r, coeff\_t*)

this part implement the trilinear scoring techniques:

$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*b_j*c_k\} \{1+(i+j+k)\%d\}$

1. generate the range for i,j and k from [0 d-1]
2. perform  $\text{dfrac}\{a_i*b_j*c_k\} \{1+(i+j+k)\%d\}$  in parallel for every batch
3. take the sum over each batch

**vtp\_score** (*h, r, t*)

this part implement the vector triple product scoring techniques:

$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*c_j*b_k - b_i*c_j*a_k\} \{(1+(i+j)\%d)(1+k)\}$

1. generate the range for i,j and k from [0 d-1]
2. Compute the first and second terms of the sum
3. Multiply with then denominator and take the sum
4. take the sum over each batch

**comp\_func** (*h, r, t*)

this part implement the function composition scoring techniques: i.e. score = <hor, t>

**polynomial** (*coeff, x, degree*)

This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,...d] and return a vector tensor (coeff[0][0] + coeff[0][1]x +...+ coeff[0][d]x^d,

**coeff[1][0] + coeff[1][1]x +...+ coeff[1][d]x^d)**

**pop** (*coeff, x, degree*)

This function allow us to evaluate the composition of two polynomes without for loops :) it takes a matrix tensor of coefficients (coeff), a matrix tensor of points x and range of integer [0,1,...d]

**and return a tensor (coeff[0][0] + coeff[0][1]x +...+ coeff[0][d]x^d,**

**coeff[1][0] + coeff[1][1]x +...+ coeff[1][d]x^d)**

```

class dicee.PykeenKGE (args: dict)
    Bases: dicee.models.base_model.BaseKGE

    A class for using knowledge graph embedding models implemented in Pykeen

    Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Py-
    keen_HolE:

    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 to nest them in a tree structure. You can assign the submodules as regular attributes:

```

import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))

```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

### Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**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 to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

---

**Note:** As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

---

## Variables

**training** (*bool*) – Boolean represents whether this module is in training or evaluation mode.

**forward\_byte\_pair\_encoded\_k\_vs\_all** (*x: torch.LongTensor*)

#### Parameters

**x** ( $B \times 2 \times T$ )

**forward\_byte\_pair\_encoded\_triple** ( $x$ : *Tuple[torch.LongTensor, torch.LongTensor]*)

byte pair encoded neural link predictors

#### Parameters

-----

**init\_params\_with\_sanity\_checking** ()

**forward** ( $x$ : *torch.LongTensor* | *Tuple[torch.LongTensor, torch.LongTensor]*,  
 $y\_idx$ : *torch.LongTensor* = *None*)

#### Parameters

- **x**
- **y\_idx**
- **ordered\_bpe\_entities**

**forward\_triples** ( $x$ : *torch.LongTensor*) → *torch.Tensor*

#### Parameters

**x**

**forward\_k\_vs\_all** (\*args, \*\*kwargs)

**forward\_k\_vs\_sample** (\*args, \*\*kwargs)

**get\_triple\_representation** ( $idx\_hrt$ )

**get\_head\_relation\_representation** ( $indexed\_triple$ )

**get\_sentence\_representation** ( $x$ : *torch.LongTensor*)

#### Parameters

- (**b** ( $x$  *shape*))
- **3**
- **t**)

**get\_bpe\_head\_and\_relation\_representation** ( $x$ : *torch.LongTensor*)  
→ *Tuple[torch.FloatTensor, torch.FloatTensor]*

#### Parameters

**x** ( $B \times 2 \times T$ )

**get\_embeddings** () → *Tuple[numpy.ndarray, numpy.ndarray]*

**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.select_model(args: dict, is_continual_training: bool = None, storage_path: str = None)`

`dicee.load_model(path_of_experiment_folder: str, model_name='model.pt', verbose=0)`  
→ Tuple[object, Tuple[dict, dict]]

Load weights and initialize pytorch module from namespace arguments

`dicee.load_model_ensemble(path_of_experiment_folder: str)`  
→ Tuple[dicee.models.base\_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]

Construct Ensemble Of weights and initialize pytorch module from namespace arguments

- (1) Detect models under given path
- (2) Accumulate parameters of detected models
- (3) Normalize parameters
- (4) Insert (3) into model.

`dicee.save_numpy_ndarray(*, data: numpy.ndarray, file_path: str)`

`dicee.numpy_data_type_changer(train_set: numpy.ndarray, num: int) → numpy.ndarray`  
Detect most efficient data type for a given triples :param train\_set: :param num: :return:

`dicee.save_checkpoint_model(model, path: str) → None`  
Store Pytorch model into disk

`dicee.store(trainer, trained_model, model_name: str = 'model', full_storage_path: str = None, save_embeddings_as_csv=False) → None`  
Store trained\_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param full\_storage\_path: path to save parameters. :param model\_name: string representation of the name of the model. :param trained\_model: an instance of BaseKGE see core.models.base\_model . :param save\_embeddings\_as\_csv: for easy access of embeddings. :return:

`dicee.add_noisy_triples(train_set: pandas.DataFrame, add_noise_rate: float) → pandas.DataFrame`  
Add randomly constructed triples :param train\_set: :param add\_noise\_rate: :return:

`dicee.read_or_load_kg(args, cls)`

`dicee.intialize_model(args: dict, verbose=0) → Tuple[object, str]`

`dicee.load_json(p: str) → dict`

`dicee.save_embeddings(embeddings: numpy.ndarray, indexes, path: str) → None`  
Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:

`dicee.random_prediction(pre_trained_kge)`

`dicee.deploy_triple_prediction(pre_trained_kge, str_subject, str_predicate, str_object)`

`dicee.deploy_tail_entity_prediction(pre_trained_kge, str_subject, str_predicate, top_k)`

`dicee.deploy_head_entity_prediction(pre_trained_kge, str_object, str_predicate, top_k)`

`dicee.deploy_relation_prediction(pre_trained_kge, str_subject, str_object, top_k)`

`dicee.vocab_to_parquet(vocab_to_idx, name, path_for_serialization, print_into)`

```

dicee.create_experiment_folder(folder_name='Experiments')

dicee.continual_training_setup_executor(executor) → None
    storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data
    full_storage_path:str A path leading to a subdirectory containing KGE related data

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

```

```

class dicee.DICE_Trainer(args, is_continual_training, storage_path, evaluator=None)

```

#### DICE\_Trainer implement

- 1- Pytorch Lightning trainer (<https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html>)
- 2- Multi-GPU Trainer(<https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>)
- 3- CPU Trainer

args

is\_continual\_training:bool

storage\_path:str

evaluator:

report:dict

**continual\_start()**

- (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

Initialize Trainer from input arguments

**initialize\_or\_load\_model** ()

**initialize\_data\_loader** (*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

**initialize\_dataset** (*dataset: dicee.knowledge\_graph.KG, form\_of\_labelling*)  
→ torch.utils.data.Dataset

**start** (*knowledge\_graph: dicee.knowledge\_graph.KG*) → Tuple[dicee.models.base\_model.BaseKGE, str]  
Train selected model via the selected training strategy

**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, apply\_semantic\_constraint=False*)

Bases: `dicee.abstracts.BaseInteractiveKGE`

Knowledge Graph Embedding Class for interactive usage of pre-trained models

**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=""*)

**\_\_str\_\_** ()

Return str(self).

**eval\_lp\_performance** (*dataset=List[Tuple[str, str, str]], filtered=True*)

**predict\_missing\_head\_entity** (*relation: List[str] | str, tail\_entity: List[str] | str, within=None*)  
→ Tuple

Given a relation and a tail entity, return top k ranked head entity.

$\text{argmax}_{\{e \in E\}} f(e, r, t)$ , where  $r \in R, t \in E$ .

### Parameter

relation: Union[List[str], str]

String representation of selected relations.

tail\_entity: Union[List[str], str]

String representation of selected entities.

k: int

Highest ranked k entities.

### Returns: Tuple

Highest K scores and entities

**predict\_missing\_relations** (*head\_entity: List[str] | str, tail\_entity: List[str] | str, within=None*)  
→ Tuple

Given a head entity and a tail entity, return top k ranked relations.

$\text{argmax}_{\{r \in R\}} f(h, r, t)$ , where  $h, t \in E$ .

### Parameter

head\_entity: List[str]

String representation of selected entities.

tail\_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

### Returns: Tuple

Highest K scores and entities

**predict\_missing\_tail\_entity** (*head\_entity: List[str] | str, relation: List[str] | str, within: List[str] = None*) → torch.FloatTensor

Given a head entity and a relation, return top k ranked entities

$\text{argmax}_{\{e \in E\}} f(h, r, e)$ , where  $h \in E$  and  $r \in R$ .

## Parameter

head\_entity: List[str]

String representation of selected entities.

tail\_entity: List[str]

String representation of selected entities.

## Returns: Tuple

scores

**predict** (\*, h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, within=None, logits=True) → torch.FloatTensor

### Parameters

- **logits**
- **h**
- **r**
- **t**
- **within**

**predict\_topk** (\*, h: List[str] = None, r: List[str] = None, t: List[str] = None, topk: int = 10, within: List[str] = None)

Predict missing item in a given 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.

k: int

Highest ranked k item.

## Returns: Tuple

Highest K scores and items

**triple\_score** (*h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, logits=False*)  
→ torch.FloatTensor  
Predict triple score

## Parameter

head\_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail\_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

## Returns: Tuple

pytorch tensor of triple score

**t\_norm** (*tens\_1: torch.Tensor, tens\_2: torch.Tensor, tnorm: str = 'min'*) → torch.Tensor

**tensor\_t\_norm** (*subquery\_scores: torch.FloatTensor, tnorm: str = 'min'*) → torch.FloatTensor

Compute T-norm over  $[0,1]^{n \times d}$  where n denotes the number of hops and d denotes number of entities

**t\_conorm** (*tens\_1: torch.Tensor, tens\_2: torch.Tensor, tconorm: str = 'min'*) → torch.Tensor

**negnorm** (*tens\_1: torch.Tensor, lambda\_: float, neg\_norm: str = 'standard'*) → torch.Tensor

**return\_multi\_hop\_query\_results** (*aggregated\_query\_for\_all\_entities, k: int, only\_scores*)

**single\_hop\_query\_answering** (*query: tuple, only\_scores: bool = True, k: int = None*)

**answer\_multi\_hop\_query** (*query\_type: str = None,*  
*query: Tuple[str | Tuple[str, str], Ellipsis] = None,*  
*queries: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, tnorm: str = 'prod',*  
*neg\_norm: str = 'standard', lambda\_: float = 0.0, k: int = 10, only\_scores=False*)  
→ List[Tuple[str, torch.Tensor]]

# @TODO: Refactoring is needed # @TODO: Score computation for each query type should be done in a static function

Find an answer set for EPFO queries including negation and disjunction



## Parameter

query\_type: str The type of the query, e.g., “2p”.

query: Union[str, Tuple[str, Tuple[str, str]]] The query itself, either a string or a nested tuple.

queries: List of Tuple[Union[str, Tuple[str, str]], ...]

tnorm: str The t-norm operator.

neg\_norm: str The negation norm.

**lambda\_**: float lambda parameter for sugeno and yager negation norms

k: int The top-k substitutions for intermediate variables.

### returns

- *List[Tuple[str, torch.Tensor]]*
- *Entities and corresponding scores sorted in the descening order of scores*

**find\_missing\_triples** (*confidence: float, entities: List[str] = None, relations: List[str] = None, topk: int = 10, at\_most: int = sys.maxsize*) → Set

Find missing triples

Iterative over a set of entities E and a set of relation R :

orall e in E and orall r in R f(e,r,x)

Return (e,r,x)

otin G and f(e,r,x) > confidence

confidence: float

A threshold for an output of a sigmoid function given a triple.

topk: int

Highest ranked k item to select triples with f(e,r,x) > confidence .

at\_most: int

Stop after finding at\_most missing triples

{(e,r,x) | f(e,r,x) > confidence land (e,r,x)}

otin G

**deploy** (*share: bool = False, top\_k: int = 10*)

**train\_triples** (*h: List[str], r: List[str], t: List[str], labels: List[float], iteration=2, optimizer=None*)

**train\_k\_vs\_all** (*h, r, iteration=1, lr=0.001*)

Train k vs all :param head\_entity: :param relation: :param iteration: :param lr: :return:

**train** (*kg, lr=0.1, epoch=10, batch\_size=32, neg\_sample\_ratio=10, num\_workers=1*) → None

Retrained a pretrain model on an input KG via negative sampling.

**class** dicee.**Execute** (*args, continuous\_training=False*)

A class for Training, Retraining and Evaluation a model.

(1) Loading & Preprocessing & Serializing input data.

(2) Training & Validation & Testing

(3) Storing all necessary info

**read\_or\_load\_kg()**

**read\_preprocess\_index\_serialize\_data()** → None

Read & Preprocess & Index & Serialize Input Data

- (1) Read or load the data from disk into memory.
- (2) Store the statistics of the data.

#### Parameter

**rtype**

None

**load\_indexed\_data()** → None

Load the indexed data from disk into memory

#### Parameter

**rtype**

None

**save\_trained\_model()** → None

Save a knowledge graph embedding model

- (1) Send model to eval mode and cpu.
- (2) Store the memory footprint of the model.
- (3) Save the model into disk.
- (4) Update the stats of KG again ?

#### Parameter

**rtype**

None

**end(form\_of\_labelling: str)** → dict

End training

- (1) Store trained model.
- (2) Report runtimes.
- (3) Eval model if required.

## Parameter

### **rtype**

A dict containing information about the training and/or evaluation

**write\_report** () → None

Report training related information in a report.json file

**start** () → dict

Start training

# (1) Loading the Data # (2) Create an evaluator object. # (3) Create a trainer object. # (4) Start the training

## Parameter

### **rtype**

A dict containing information about the training and/or evaluation

`dicee.mapping_from_first_two_cols_to_third(train_set_idx)`

`dicee.timeit(func)`

`dicee.load_pickle(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.

---

`__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.

---

```
__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\_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

```
__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

`__len__()``__getitem__(idx)`

```
class dicee.KvsAll (train_set_idx: numpy.ndarray, entity_idx, relation_idx, form, store=None,
                    label_smoothing_rate: float = 0.0)
```

Bases: torch.utils.data.Dataset

**Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.**

Let D denote a dataset for KvsAll training and be defined as  $D := \{(x, y)_i\}_i^N$ , where  $x: (h, r)$  is a unique tuple of an entity  $h$  in  $E$  and a relation  $r$  in  $R$  that has been seed in the input graph.  $y$ : denotes a multi-label vector in  $[0, 1]^{|E|}$  **|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\_idx**

[dictionary] string representation of an entity to its integer id

**relation\_idx**

[dictionary] string representation of a relation to its integer id

self : torch.utils.data.Dataset

```
>>> a = KvsAll()
>>> a
? array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

`__len__()``__getitem__(idx)`

```
class dicee.AllvsAll (train_set_idx: numpy.ndarray, entity_idx, relation_idx,
                      label_smoothing_rate=0.0)
```

Bases: torch.utils.data.Dataset

**Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.**

Let D denote a dataset for AllvsAll training and be defined as  $D := \{(x, y)_i\}_i^N$ , where  $x: (h, r)$  is a possible unique tuple of an entity  $h$  in  $E$  and a relation  $r$  in  $R$ . Hence  $N = |E| \times |R|$   $y$ : denotes a multi-label vector in  $[0, 1]^{|E|}$  **|E|** is a binary label.

orall  $y_i = 1$  s.t.  $(h \ r \ E_i)$  in KG

---

**Note:**

**AllvsAll extends KvsAll via none existing (h,r). Hence, it adds data points that are labelled without 1s, only with 0s.**

---

**train\_set\_idx**  
[numpy.ndarray] n by 3 array representing n triples

**entity\_idx**  
[dictionary] string representation of an entity to its integer id

**relation\_idx**  
[dictionary] string representation of a relation to its integer id

self : torch.utils.data.Dataset

```
>>> a = AllvsAll()
>>> a
? array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

**\_\_len\_\_()**

**\_\_getitem\_\_**(idx)

**class** dicee.**KvsSampleDataset** (train\_set: numpy.ndarray, num\_entities, num\_relations,  
neg\_sample\_ratio: int = 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]<sup>{|E|}</sup> 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

**\_\_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.

---

```
__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\_idx**

mapping.

**relation\_idx**

mapping.

**form**

?

**store**

?

label\_smoothing\_rate

collate\_fn: batch:List[torch.IntTensor] Returns —— torch.utils.data.Dataset

```
__len__ ()
```

```
__getitem__ (idx)
```

```
collate_fn (batch: List[torch.Tensor])
```

```
class dicee.CVDataModule (train_set_idx: numpy.ndarray, num_entities, num_relations,  
                        neg_sample_ratio, batch_size, num_workers)
```

Bases: `pytorch_lightning.LightningDataModule`

Create a Dataset for cross validation

#### Parameters

- **train\_set\_idx** – Indexed triples for the training.
- **num\_entities** – entity to index mapping.
- **num\_relations** – relation to index mapping.
- **batch\_size** – int
- **form** – ?
- **num\_workers** – int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

#### Return type

?

**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:
```

(continues on next page)

(continued from previous page)

```
        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()
```

(continues on next page)

(continued from previous page)

```
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)

    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__ = '0.1.4'
```

## Python Module Index

### d

- `dicee`, 10
- `dicee.abstracts`, 128
- `dicee.analyse_experiments`, 134
- `dicee.callbacks`, 135
- `dicee.config`, 143
- `dicee.dataset_classes`, 145
- `dicee.eval_static_funcs`, 154
- `dicee.evaluator`, 160
- `dicee.executer`, 161
- `dicee.knowledge_graph`, 167
- `dicee.knowledge_graph_embeddings`, 168
- `dicee.models`, 10
  - `dicee.models.base_model`, 10
  - `dicee.models.clifford`, 18
  - `dicee.models.complex`, 27
  - `dicee.models.dualE`, 30
  - `dicee.models.function_space`, 33
  - `dicee.models.octonion`, 37
  - `dicee.models.pykeen_models`, 41
  - `dicee.models.quaternion`, 44
  - `dicee.models.real`, 49
  - `dicee.models.static_funcs`, 51
  - `dicee.models.transformers`, 52
- `dicee.query_generator`, 177
- `dicee.read_preprocess_save_load_kg`, 95
- `dicee.read_preprocess_save_load_kg.preprocess`, 95
- `dicee.read_preprocess_save_load_kg.read_from_disk`, 97
- `dicee.read_preprocess_save_load_kg.save_load_disk`, 98
- `dicee.read_preprocess_save_load_kg.util`, 99
- `dicee.sanity_checkers`, 178
- `dicee.scripts`, 103
  - `dicee.scripts.index`, 103
  - `dicee.scripts.run`, 104
  - `dicee.scripts.serve`, 106
- `dicee.static_funcs`, 178
- `dicee.static_funcs_training`, 202
- `dicee.static_preprocess_funcs`, 203
- `dicee.trainer`, 111
  - `dicee.trainer.dice_trainer`, 111
  - `dicee.trainer.torch_trainer`, 121
  - `dicee.trainer.torch_trainer_ddp`, 123

# Index

## Non-alphabetical

`__call__` () (*dicее.models.base\_model.IdentityClass method*), 17  
`__call__` () (*dicее.models.IdentityClass method*), 67, 74, 79  
`__call__` () (*dicее.models.octonion.IdentityClass method*), 39  
`__call__` () (*dicее.models.quaternion.IdentityClass method*), 46  
`__getitem__` () (*dicее.AllvsAll method*), 238  
`__getitem__` () (*dicее.BPE\_NegativeSamplingDataset method*), 235  
`__getitem__` () (*dicее.dataset\_classes.AllvsAll method*), 149  
`__getitem__` () (*dicее.dataset\_classes.BPE\_NegativeSamplingDataset method*), 147  
`__getitem__` () (*dicее.dataset\_classes.KvsAll method*), 148  
`__getitem__` () (*dicее.dataset\_classes.KvsSampleDataset method*), 150  
`__getitem__` () (*dicее.dataset\_classes.MultiClassClassificationDataset method*), 148  
`__getitem__` () (*dicее.dataset\_classes.MultiLabelDataset method*), 147  
`__getitem__` () (*dicее.dataset\_classes.NegSampleDataset method*), 150  
`__getitem__` () (*dicее.dataset\_classes.OnevsAllDataset method*), 148  
`__getitem__` () (*dicее.dataset\_classes.TriplePredictionDataset method*), 151  
`__getitem__` () (*dicее.knowledge\_graph\_embeddings.TriplePredictionDataset method*), 171  
`__getitem__` () (*dicее.KvsAll method*), 237  
`__getitem__` () (*dicее.KvsSampleDataset method*), 238  
`__getitem__` () (*dicее.MultiClassClassificationDataset method*), 236  
`__getitem__` () (*dicее.MultiLabelDataset method*), 236  
`__getitem__` () (*dicее.NegSampleDataset method*), 239  
`__getitem__` () (*dicее.OnevsAllDataset method*), 237  
`__getitem__` () (*dicее.TriplePredictionDataset method*), 239  
`__iter__` () (*dicее.config.Namespace method*), 145  
`__len__` () (*dicее.AllvsAll method*), 238  
`__len__` () (*dicее.BPE\_NegativeSamplingDataset method*), 235  
`__len__` () (*dicее.dataset\_classes.AllvsAll method*), 149  
`__len__` () (*dicее.dataset\_classes.BPE\_NegativeSamplingDataset method*), 147  
`__len__` () (*dicее.dataset\_classes.KvsAll method*), 148  
`__len__` () (*dicее.dataset\_classes.KvsSampleDataset method*), 150  
`__len__` () (*dicее.dataset\_classes.MultiClassClassificationDataset method*), 147  
`__len__` () (*dicее.dataset\_classes.MultiLabelDataset method*), 147  
`__len__` () (*dicее.dataset\_classes.NegSampleDataset method*), 150  
`__len__` () (*dicее.dataset\_classes.OnevsAllDataset method*), 148  
`__len__` () (*dicее.dataset\_classes.TriplePredictionDataset method*), 151  
`__len__` () (*dicее.knowledge\_graph\_embeddings.TriplePredictionDataset method*), 171  
`__len__` () (*dicее.KvsAll method*), 237  
`__len__` () (*dicее.KvsSampleDataset method*), 238  
`__len__` () (*dicее.MultiClassClassificationDataset method*), 236  
`__len__` () (*dicее.MultiLabelDataset method*), 236  
`__len__` () (*dicее.NegSampleDataset method*), 239  
`__len__` () (*dicее.OnevsAllDataset method*), 237  
`__len__` () (*dicее.TriplePredictionDataset method*), 239  
`__str__` () (*dicее.eval\_static\_funcs.KGE method*), 155  
`__str__` () (*dicее.KGE method*), 229  
`__str__` () (*dicее.knowledge\_graph\_embeddings.KGE method*), 172  
`__str__` () (*dicее.scripts.serve.KGE method*), 107  
`__version__` (in module *dicее*), 243

## A

`AbstractCallback` (class in *dicее.abstracts*), 132  
`AbstractCallback` (class in *dicее.callbacks*), 136  
`AbstractPPECallback` (class in *dicее.abstracts*), 133  
`AbstractTrainer` (class in *dicее.abstracts*), 129  
`AbstractTrainer` (class in *dicее.trainer.torch\_trainer*), 121  
`AbstractTrainer` (class in *dicее.trainer.torch\_trainer\_ddp*), 124  
`AccumulateEpochLossCallback` (class in *dicее.callbacks*), 137  
`AccumulateEpochLossCallback` (class in *dicее.trainer.dice\_trainer*), 117  
`achieve_answer` () (*dicее.query\_generator.QueryGenerator method*), 177  
`achieve_answer` () (*dicее.QueryGenerator method*), 243  
`AConEx` (class in *dicее*), 217  
`AConEx` (class in *dicее.models*), 71  
`AConEx` (class in *dicее.models.complex*), 29

AConEx (class in *dicee.static\_funcs*), 190  
 AConvO (class in *dicee*), 217  
 AConvO (class in *dicee.models*), 80  
 AConvO (class in *dicee.models.octonion*), 41  
 AConvO (class in *dicee.static\_funcs*), 191  
 AConvQ (class in *dicee*), 217  
 AConvQ (class in *dicee.models*), 76  
 AConvQ (class in *dicee.models.quaternion*), 48  
 AConvQ (class in *dicee.static\_funcs*), 191  
 adaptive\_swa (*dicee.config.Namespace* attribute), 145  
 add\_new\_entity\_embeddings () (*dicee.abstracts.BaseInteractiveKGE* method), 131  
 add\_new\_entity\_embeddings () (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE* method), 170  
 add\_noise\_rate (*dicee.config.Namespace* attribute), 143  
 add\_noisy\_triples () (in module *dicee*), 227  
 add\_noisy\_triples () (in module *dicee.static\_funcs*), 201  
 add\_noisy\_triples\_into\_training () (*dicee.knowledge\_graph.ReadFromDisk* method), 167  
 add\_noisy\_triples\_into\_training () (*dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk.ReadFromDisk* method), 98  
 add\_noisy\_triples\_into\_training () (*dicee.read\_preprocess\_save\_load\_kg.ReadFromDisk* method), 103  
 AllvsAll (class in *dicee*), 237  
 AllvsAll (class in *dicee.dataset\_classes*), 149  
 analyse () (in module *dicee.analyse\_experiments*), 135  
 answer\_multi\_hop\_query () (*dicee.eval\_static\_funcs.KGE* method), 158  
 answer\_multi\_hop\_query () (*dicee.KGE* method), 232  
 answer\_multi\_hop\_query () (*dicee.knowledge\_graph\_embeddings.KGE* method), 175  
 answer\_multi\_hop\_query () (*dicee.scripts.serve.KGE* method), 110  
 app (in module *dicee.scripts.serve*), 111  
 apply\_coefficients () (*dicee.DeCaL* method), 213  
 apply\_coefficients () (*dicee.Keci* method), 210  
 apply\_coefficients () (*dicee.models.clifford.DeCaL* method), 25  
 apply\_coefficients () (*dicee.models.clifford.Keci* method), 22  
 apply\_coefficients () (*dicee.models.DeCaL* method), 86  
 apply\_coefficients () (*dicee.models.Keci* method), 82  
 apply\_coefficients () (*dicee.static\_funcs.DeCaL* method), 187  
 apply\_coefficients () (*dicee.static\_funcs.Keci* method), 184  
 apply\_reciprical\_or\_noise () (in module *dicee.read\_preprocess\_save\_load\_kg.preprocess*), 96  
 apply\_reciprical\_or\_noise () (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 100  
 ASWA (class in *dicee.callbacks*), 140  
 ASWA (class in *dicee.trainer.dice\_trainer*), 114

## B

backend (*dicee.config.Namespace* attribute), 144  
 BaseInteractiveKGE (class in *dicee.abstracts*), 130  
 BaseInteractiveKGE (class in *dicee.knowledge\_graph\_embeddings*), 169  
 BaseKGE (class in *dicee*), 225  
 BaseKGE (class in *dicee.models*), 65, 67, 69, 72, 76, 88, 90  
 BaseKGE (class in *dicee.models.base\_model*), 15  
 BaseKGE (class in *dicee.models.clifford*), 18  
 BaseKGE (class in *dicee.models.complex*), 27  
 BaseKGE (class in *dicee.models.dualE*), 30  
 BaseKGE (class in *dicee.models.function\_space*), 33  
 BaseKGE (class in *dicee.models.octonion*), 37  
 BaseKGE (class in *dicee.models.pykeen\_models*), 42  
 BaseKGE (class in *dicee.models.quaternion*), 44  
 BaseKGE (class in *dicee.models.real*), 49  
 BaseKGE (class in *dicee.models.transformers*), 52  
 BaseKGE (class in *dicee.static\_funcs*), 199  
 BaseKGE (class in *dicee.trainer.dice\_trainer*), 112  
 BaseKGELightning (class in *dicee.models*), 60  
 BaseKGELightning (class in *dicee.models.base\_model*), 10  
 batch\_kronecker\_product () (*dicee.callbacks.KronE* static method), 142  
 batch\_kronecker\_product () (*dicee.trainer.dice\_trainer.KronE* static method), 116  
 batch\_size (*dicee.config.Namespace* attribute), 143  
 bias (*dicee.models.transformers.GPTConfig* attribute), 57  
 Block (class in *dicee.models.transformers*), 57  
 block\_size (*dicee.config.Namespace* attribute), 145  
 block\_size (*dicee.models.transformers.GPTConfig* attribute), 57  
 BPE\_NegativeSamplingDataset (class in *dicee*), 235

BPE\_NegativeSamplingDataset (class in *dicee.dataset\_classes*), 146  
 build\_chain\_funcs () (*dicee.models.FMult2* method), 92  
 build\_chain\_funcs () (*dicee.models.function\_space.FMult2* method), 35  
 build\_func () (*dicee.models.FMult2* method), 92  
 build\_func () (*dicee.models.function\_space.FMult2* method), 35  
 Byte (class in *dicee*), 223  
 Byte (class in *dicee.models.transformers*), 53  
 Byte (class in *dicee.static\_funcs*), 197  
 byte\_pair\_encoding (*dicee.config.Namespace* attribute), 145

## C

callbacks (*dicee.config.Namespace* attribute), 144  
 CausalSelfAttention (class in *dicee.models.transformers*), 55  
 chain\_func () (*dicee.models.FMult* method), 92  
 chain\_func () (*dicee.models.function\_space.FMult* method), 34  
 chain\_func () (*dicee.models.function\_space.GFMult* method), 35  
 chain\_func () (*dicee.models.GFMult* method), 92  
 cl\_pqr () (*dicee.DeCaL* method), 212  
 cl\_pqr () (*dicee.models.clifford.DeCaL* method), 24  
 cl\_pqr () (*dicee.models.DeCaL* method), 86  
 cl\_pqr () (*dicee.static\_funcs.DeCaL* method), 186  
 clifford\_mul () (*dicee.CMult* method), 207  
 clifford\_mul () (*dicee.models.clifford.CMult* method), 20  
 clifford\_mul () (*dicee.models.CMult* method), 84  
 clifford\_mul () (*dicee.static\_funcs.CMult* method), 181  
 clifford\_multiplication () (*dicee.Keci* method), 210  
 clifford\_multiplication () (*dicee.models.clifford.Keci* method), 22  
 clifford\_multiplication () (*dicee.models.Keci* method), 82  
 clifford\_multiplication () (*dicee.static\_funcs.Keci* method), 184  
 CMult (class in *dicee*), 207  
 CMult (class in *dicee.models*), 84  
 CMult (class in *dicee.models.clifford*), 19  
 CMult (class in *dicee.static\_funcs*), 181  
 collate\_fn () (*dicee.BPE\_NegativeSamplingDataset* method), 235  
 collate\_fn () (*dicee.dataset\_classes.BPE\_NegativeSamplingDataset* method), 147  
 collate\_fn () (*dicee.dataset\_classes.TriplePredictionDataset* method), 151  
 collate\_fn () (*dicee.knowledge\_graph\_embeddings.TriplePredictionDataset* method), 171  
 collate\_fn () (*dicee.TriplePredictionDataset* method), 239  
 comp\_func () (*dicee.LFMult* method), 222  
 comp\_func () (*dicee.models.function\_space.LFMult* method), 36  
 comp\_func () (*dicee.models.LFMult* method), 93  
 comp\_func () (*dicee.static\_funcs.LFMult* method), 196  
 Complex (class in *dicee*), 216  
 Complex (class in *dicee.models*), 71  
 Complex (class in *dicee.models.complex*), 29  
 Complex (class in *dicee.static\_funcs*), 190  
 compute\_convergence () (in module *dicee.callbacks*), 140  
 compute\_func () (*dicee.models.FMult* method), 92  
 compute\_func () (*dicee.models.FMult2* method), 92  
 compute\_func () (*dicee.models.function\_space.FMult* method), 34  
 compute\_func () (*dicee.models.function\_space.FMult2* method), 35  
 compute\_func () (*dicee.models.function\_space.GFMult* method), 35  
 compute\_func () (*dicee.models.GFMult* method), 92  
 compute\_mrr () (*dicee.callbacks.ASWA* static method), 140  
 compute\_mrr () (*dicee.trainer.dice\_trainer.ASWA* static method), 114  
 compute\_sigma\_pp () (*dicee.DeCaL* method), 213  
 compute\_sigma\_pp () (*dicee.Keci* method), 209  
 compute\_sigma\_pp () (*dicee.models.clifford.DeCaL* method), 25  
 compute\_sigma\_pp () (*dicee.models.clifford.Keci* method), 21  
 compute\_sigma\_pp () (*dicee.models.DeCaL* method), 87  
 compute\_sigma\_pp () (*dicee.models.Keci* method), 81  
 compute\_sigma\_pp () (*dicee.static\_funcs.DeCaL* method), 187  
 compute\_sigma\_pp () (*dicee.static\_funcs.Keci* method), 183  
 compute\_sigma\_pq () (*dicee.DeCaL* method), 214  
 compute\_sigma\_pq () (*dicee.Keci* method), 210  
 compute\_sigma\_pq () (*dicee.models.clifford.DeCaL* method), 26  
 compute\_sigma\_pq () (*dicee.models.clifford.Keci* method), 21

`compute_sigma_pq()` (*dicee.models.DeCaL method*), 88  
`compute_sigma_pq()` (*dicee.models.Keci method*), 82  
`compute_sigma_pq()` (*dicee.static\_funcs.DeCaL method*), 188  
`compute_sigma_pq()` (*dicee.static\_funcs.Keci method*), 183  
`compute_sigma_pr()` (*dicee.DeCaL method*), 214  
`compute_sigma_pr()` (*dicee.models.clifford.DeCaL method*), 26  
`compute_sigma_pr()` (*dicee.models.DeCaL method*), 88  
`compute_sigma_pr()` (*dicee.static\_funcs.DeCaL method*), 188  
`compute_sigma_qq()` (*dicee.DeCaL method*), 214  
`compute_sigma_qq()` (*dicee.Keci method*), 209  
`compute_sigma_qq()` (*dicee.models.clifford.DeCaL method*), 25  
`compute_sigma_qq()` (*dicee.models.clifford.Keci method*), 21  
`compute_sigma_qq()` (*dicee.models.DeCaL method*), 87  
`compute_sigma_qq()` (*dicee.models.Keci method*), 82  
`compute_sigma_qq()` (*dicee.static\_funcs.DeCaL method*), 187  
`compute_sigma_qq()` (*dicee.static\_funcs.Keci method*), 183  
`compute_sigma_qr()` (*dicee.DeCaL method*), 214  
`compute_sigma_qr()` (*dicee.models.clifford.DeCaL method*), 26  
`compute_sigma_qr()` (*dicee.models.DeCaL method*), 88  
`compute_sigma_qr()` (*dicee.static\_funcs.DeCaL method*), 188  
`compute_sigma_rr()` (*dicee.DeCaL method*), 214  
`compute_sigma_rr()` (*dicee.models.clifford.DeCaL method*), 26  
`compute_sigma_rr()` (*dicee.models.DeCaL method*), 87  
`compute_sigma_rr()` (*dicee.static\_funcs.DeCaL method*), 188  
`compute_sigmas_multivect()` (*dicee.DeCaL method*), 212  
`compute_sigmas_multivect()` (*dicee.models.clifford.DeCaL method*), 24  
`compute_sigmas_multivect()` (*dicee.models.DeCaL method*), 86  
`compute_sigmas_multivect()` (*dicee.static\_funcs.DeCaL method*), 186  
`compute_sigmas_single()` (*dicee.DeCaL method*), 212  
`compute_sigmas_single()` (*dicee.models.clifford.DeCaL method*), 24  
`compute_sigmas_single()` (*dicee.models.DeCaL method*), 86  
`compute_sigmas_single()` (*dicee.static\_funcs.DeCaL method*), 186  
`ConEx` (*class in dicee*), 219  
`ConEx` (*class in dicee.models*), 71  
`ConEx` (*class in dicee.models.complex*), 28  
`ConEx` (*class in dicee.static\_funcs*), 193  
`configure_optimizers()` (*dicee.models.base\_model.BaseKGELightning method*), 14  
`configure_optimizers()` (*dicee.models.BaseKGELightning method*), 64  
`configure_optimizers()` (*dicee.models.transformers.GPT method*), 58  
`construct_cl_multivector()` (*dicee.DeCaL method*), 213  
`construct_cl_multivector()` (*dicee.Keci method*), 210  
`construct_cl_multivector()` (*dicee.models.clifford.DeCaL method*), 25  
`construct_cl_multivector()` (*dicee.models.clifford.Keci method*), 22  
`construct_cl_multivector()` (*dicee.models.DeCaL method*), 86  
`construct_cl_multivector()` (*dicee.models.Keci method*), 82  
`construct_cl_multivector()` (*dicee.static\_funcs.DeCaL method*), 187  
`construct_cl_multivector()` (*dicee.static\_funcs.Keci method*), 184  
`construct_dataset()` (*in module dicee*), 235  
`construct_dataset()` (*in module dicee.dataset\_classes*), 146  
`construct_dataset()` (*in module dicee.trainer.dice\_trainer*), 118  
`construct_graph()` (*dicee.query\_generator.QueryGenerator method*), 177  
`construct_graph()` (*dicee.QueryGenerator method*), 243  
`construct_input_and_output()` (*dicee.abstracts.BaseInteractiveKGE method*), 132  
`construct_input_and_output()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 171  
`construct_multi_coeff()` (*dicee.LFMult method*), 222  
`construct_multi_coeff()` (*dicee.models.function\_space.LFMult method*), 35  
`construct_multi_coeff()` (*dicee.models.LFMult method*), 93  
`construct_multi_coeff()` (*dicee.static\_funcs.LFMult method*), 195  
`continual_learning` (*dicee.config.Namespace attribute*), 145  
`continual_start()` (*dicee.DICE\_Trainer method*), 228  
`continual_start()` (*dicee.executer.ContinuousExecute method*), 166  
`continual_start()` (*dicee.executer.DICE\_Trainer method*), 163  
`continual_start()` (*dicee.scripts.run.ContinuousExecute method*), 106  
`continual_start()` (*dicee.trainer.DICE\_Trainer method*), 127  
`continual_start()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 120  
`continual_training_setup_executor()` (*in module dicee*), 228  
`continual_training_setup_executor()` (*in module dicee.executer*), 164  
`continual_training_setup_executor()` (*in module dicee.static\_funcs*), 202



ContinuousExecute (class in *dicee.executer*), 166  
 ContinuousExecute (class in *dicee.scripts.run*), 105  
 ConvO (class in *dicee*), 218  
 ConvO (class in *dicee.models*), 79  
 ConvO (class in *dicee.models.octonion*), 40  
 ConvO (class in *dicee.static\_funcs*), 192  
 ConvQ (class in *dicee*), 217  
 ConvQ (class in *dicee.models*), 76  
 ConvQ (class in *dicee.models.quaternion*), 48  
 ConvQ (class in *dicee.static\_funcs*), 191  
 create\_constraints() (in module *dicee.read\_preprocess\_save\_load\_kg.preprocess*), 96  
 create\_constraints() (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 101  
 create\_constraints() (in module *dicee.static\_preprocess\_funcs*), 204  
 create\_experiment\_folder() (in module *dicee*), 227  
 create\_experiment\_folder() (in module *dicee.static\_funcs*), 202  
 create\_random\_data() (*dicee.callbacks.PseudoLabellingCallback* method), 140  
 create\_recipriocal\_triples() (in module *dicee*), 226  
 create\_recipriocal\_triples() (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 101  
 create\_recipriocal\_triples() (in module *dicee.static\_funcs*), 200  
 create\_vector\_database() (*dicee.eval\_static\_funcs.KGE* method), 155  
 create\_vector\_database() (*dicee.KGE* method), 229  
 create\_vector\_database() (*dicee.knowledge\_graph\_embeddings.KGE* method), 172  
 create\_vector\_database() (*dicee.scripts.serve.KGE* method), 107  
 crop\_block\_size() (*dicee.models.transformers.GPT* method), 58  
 CVDataModule (class in *dicee*), 239  
 CVDataModule (class in *dicee.dataset\_classes*), 151

## D

dataset\_dir (*dicee.config.Namespace* attribute), 143  
 dataset\_sanity\_checking() (in module *dicee.read\_preprocess\_save\_load\_kg.preprocess*), 96  
 dataset\_sanity\_checking() (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 101  
 DDPTrainer (class in *dicee.trainer.torch\_trainer\_ddp*), 126  
 DeCaL (class in *dicee*), 211  
 DeCaL (class in *dicee.models*), 85  
 DeCaL (class in *dicee.models.clifford*), 23  
 DeCaL (class in *dicee.static\_funcs*), 185  
 decide() (*dicee.callbacks.ASWA* method), 140  
 decide() (*dicee.trainer.dice\_trainer.ASWA* method), 114  
 deploy() (*dicee.eval\_static\_funcs.KGE* method), 158  
 deploy() (*dicee.KGE* method), 233  
 deploy() (*dicee.knowledge\_graph\_embeddings.KGE* method), 176  
 deploy() (*dicee.scripts.serve.KGE* method), 111  
 deploy\_head\_entity\_prediction() (in module *dicee*), 227  
 deploy\_head\_entity\_prediction() (in module *dicee.knowledge\_graph\_embeddings*), 172  
 deploy\_head\_entity\_prediction() (in module *dicee.static\_funcs*), 201  
 deploy\_relation\_prediction() (in module *dicee*), 227  
 deploy\_relation\_prediction() (in module *dicee.knowledge\_graph\_embeddings*), 172  
 deploy\_relation\_prediction() (in module *dicee.static\_funcs*), 202  
 deploy\_tail\_entity\_prediction() (in module *dicee*), 227  
 deploy\_tail\_entity\_prediction() (in module *dicee.knowledge\_graph\_embeddings*), 172  
 deploy\_tail\_entity\_prediction() (in module *dicee.static\_funcs*), 201  
 deploy\_triple\_prediction() (in module *dicee*), 227  
 deploy\_triple\_prediction() (in module *dicee.knowledge\_graph\_embeddings*), 172  
 deploy\_triple\_prediction() (in module *dicee.static\_funcs*), 201  
 DICE\_Trainer (class in *dicee*), 228  
 DICE\_Trainer (class in *dicee.executer*), 163  
 DICE\_Trainer (class in *dicee.trainer*), 126  
 DICE\_Trainer (class in *dicee.trainer.dice\_trainer*), 120  
 dicee  
     module, 10  
 dicee.abstracts  
     module, 128  
 dicee.analyse\_experiments  
     module, 134  
 dicee.callbacks  
     module, 135  
 dicee.config

- module, 143
- dicee.dataset\_classes
  - module, 145
- dicee.eval\_static\_funcs
  - module, 154
- dicee.evaluator
  - module, 160
- dicee.executer
  - module, 161
- dicee.knowledge\_graph
  - module, 167
- dicee.knowledge\_graph\_embeddings
  - module, 168
- dicee.models
  - module, 10
- dicee.models.base\_model
  - module, 10
- dicee.models.clifford
  - module, 18
- dicee.models.complex
  - module, 27
- dicee.models.dualE
  - module, 30
- dicee.models.function\_space
  - module, 33
- dicee.models.octonion
  - module, 37
- dicee.models.pykeen\_models
  - module, 41
- dicee.models.quaternion
  - module, 44
- dicee.models.real
  - module, 49
- dicee.models.static\_funcs
  - module, 51
- dicee.models.transformers
  - module, 52
- dicee.query\_generator
  - module, 177
- dicee.read\_preprocess\_save\_load\_kg
  - module, 95
- dicee.read\_preprocess\_save\_load\_kg.preprocess
  - module, 95
- dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk
  - module, 97
- dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk
  - module, 98
- dicee.read\_preprocess\_save\_load\_kg.util
  - module, 99
- dicee.sanity\_checkers
  - module, 178
- dicee.scripts
  - module, 103
- dicee.scripts.index
  - module, 103
- dicee.scripts.run
  - module, 104
- dicee.scripts.serve
  - module, 106
- dicee.static\_funcs
  - module, 178
- dicee.static\_funcs\_training
  - module, 202
- dicee.static\_preprocess\_funcs
  - module, 203
- dicee.trainer
  - module, 111
- dicee.trainer.dice\_trainer

- module, 111
- dicee.trainer.torch\_trainer
  - module, 121
- dicee.trainer.torch\_trainer\_ddp
  - module, 123
- DistMult (class in dicee), 208
- DistMult (class in dicee.models), 68
- DistMult (class in dicee.models.real), 50
- DistMult (class in dicee.static\_funcs), 182
- download\_file() (in module dicee), 228
- download\_file() (in module dicee.static\_funcs), 202
- download\_files\_from\_url() (in module dicee), 228
- download\_files\_from\_url() (in module dicee.static\_funcs), 202
- download\_pretrained\_model() (in module dicee), 228
- download\_pretrained\_model() (in module dicee.abstracts), 128
- download\_pretrained\_model() (in module dicee.static\_funcs), 202
- dropout (dicee.models.transformers.GPTConfig attribute), 57
- DualE (class in dicee), 215
- DualE (class in dicee.models), 94
- DualE (class in dicee.models.dualE), 31
- DualE (class in dicee.static\_funcs), 189
- dummy\_eval() (dicee.evaluator.Evaluator method), 161
- dummy\_eval() (dicee.executer.Evaluator method), 163

## E

- efficient\_zero\_grad() (in module dicee.static\_funcs\_training), 203
- efficient\_zero\_grad() (in module dicee.trainer.torch\_trainer\_ddp), 125
- embedding\_dim (dicee.config.Namespace attribute), 143
- enable\_log (in module dicee.static\_preprocess\_funcs), 204
- end() (dicee.Execute method), 234
- end() (dicee.executer.Execute method), 165
- end() (dicee.scripts.run.Execute method), 105
- entities\_str (dicee.evaluator.KG property), 160
- entities\_str (dicee.executer.KG property), 162
- entities\_str (dicee.knowledge\_graph.KG property), 168
- entities\_str (dicee.trainer.dice\_trainer.KG property), 119
- estimate\_mfu() (dicee.models.transformers.GPT method), 58
- estimate\_q() (in module dicee.callbacks), 140
- Eval (class in dicee.callbacks), 141
- Eval (class in dicee.trainer.dice\_trainer), 115
- eval() (dicee.evaluator.Evaluator method), 160
- eval() (dicee.executer.Evaluator method), 162
- eval\_lp\_performance() (dicee.eval\_static\_funcs.KGE method), 155
- eval\_lp\_performance() (dicee.KGE method), 229
- eval\_lp\_performance() (dicee.knowledge\_graph\_embeddings.KGE method), 172
- eval\_lp\_performance() (dicee.scripts.serve.KGE method), 107
- eval\_model (dicee.config.Namespace attribute), 144
- eval\_rank\_of\_head\_and\_tail\_byte\_pair\_encoded\_entity() (dicee.evaluator.Evaluator method), 161
- eval\_rank\_of\_head\_and\_tail\_byte\_pair\_encoded\_entity() (dicee.executer.Evaluator method), 162
- eval\_rank\_of\_head\_and\_tail\_entity() (dicee.evaluator.Evaluator method), 161
- eval\_rank\_of\_head\_and\_tail\_entity() (dicee.executer.Evaluator method), 162
- eval\_with\_bpe\_vs\_all() (dicee.evaluator.Evaluator method), 161
- eval\_with\_bpe\_vs\_all() (dicee.executer.Evaluator method), 162
- eval\_with\_byte() (dicee.evaluator.Evaluator method), 161
- eval\_with\_byte() (dicee.executer.Evaluator method), 162
- eval\_with\_data() (dicee.evaluator.Evaluator method), 161
- eval\_with\_data() (dicee.executer.Evaluator method), 163
- eval\_with\_vs\_all() (dicee.evaluator.Evaluator method), 161
- eval\_with\_vs\_all() (dicee.executer.Evaluator method), 163
- evaluate() (in module dicee), 228
- evaluate() (in module dicee.static\_funcs), 202
- evaluate\_bpe\_lp() (in module dicee.evaluator), 160
- evaluate\_bpe\_lp() (in module dicee.static\_funcs\_training), 203
- evaluate\_link\_prediction\_performance() (in module dicee.eval\_static\_funcs), 159
- evaluate\_link\_prediction\_performance\_with\_bpe() (in module dicee.eval\_static\_funcs), 159
- evaluate\_link\_prediction\_performance\_with\_bpe\_reciprocals() (in module dicee.eval\_static\_funcs), 159
- evaluate\_link\_prediction\_performance\_with\_reciprocals() (in module dicee.eval\_static\_funcs), 159

[evaluate\\_lp\(\)](#) (*dicee.evaluator.Evaluator method*), 161  
[evaluate\\_lp\(\)](#) (*dicee.executer.Evaluator method*), 163  
[evaluate\\_lp\(\)](#) (in module *dicee.evaluator*), 160  
[evaluate\\_lp\(\)](#) (in module *dicee.knowledge\_graph\_embeddings*), 172  
[evaluate\\_lp\(\)](#) (in module *dicee.static\_funcs\_training*), 203  
[evaluate\\_lp\\_bpe\\_k\\_vs\\_all\(\)](#) (*dicee.evaluator.Evaluator method*), 161  
[evaluate\\_lp\\_bpe\\_k\\_vs\\_all\(\)](#) (*dicee.executer.Evaluator method*), 163  
[evaluate\\_lp\\_bpe\\_k\\_vs\\_all\(\)](#) (in module *dicee.eval\_static\_funcs*), 159  
[evaluate\\_lp\\_k\\_vs\\_all\(\)](#) (*dicee.evaluator.Evaluator method*), 161  
[evaluate\\_lp\\_k\\_vs\\_all\(\)](#) (*dicee.executer.Evaluator method*), 163  
[evaluate\\_lp\\_with\\_byte\(\)](#) (*dicee.evaluator.Evaluator method*), 161  
[evaluate\\_lp\\_with\\_byte\(\)](#) (*dicee.executer.Evaluator method*), 163  
[Evaluator](#) (class in *dicee.evaluator*), 160  
[Evaluator](#) (class in *dicee.executer*), 162  
[Execute](#) (class in *dicee*), 233  
[Execute](#) (class in *dicee.executer*), 164  
[Execute](#) (class in *dicee.scripts.run*), 104  
[Experiment](#) (class in *dicee.analyse\_experiments*), 135  
[exponential\\_function\(\)](#) (in module *dicee*), 228  
[exponential\\_function\(\)](#) (in module *dicee.static\_funcs*), 202  
[extract\\_input\\_outputs\(\)](#) (*dicee.trainer.torch\_trainer\_ddp.DDPTrainer method*), 126  
[extract\\_input\\_outputs\(\)](#) (*dicee.trainer.torch\_trainer\_ddp.NodeTrainer method*), 126  
[extract\\_input\\_outputs\\_set\\_device\(\)](#) (*dicee.trainer.dice\_trainer.TorchTrainer method*), 119  
[extract\\_input\\_outputs\\_set\\_device\(\)](#) (*dicee.trainer.torch\_trainer.TorchTrainer method*), 123

## F

[feature\\_map\\_dropout\\_rate](#) (*dicee.config.Namespace attribute*), 145  
[fill\\_query\(\)](#) (*dicee.query\_generator.QueryGenerator method*), 177  
[fill\\_query\(\)](#) (*dicee.QueryGenerator method*), 243  
[find\\_missing\\_triples\(\)](#) (*dicee.eval\_static\_funcs.KGE method*), 158  
[find\\_missing\\_triples\(\)](#) (*dicee.KGE method*), 233  
[find\\_missing\\_triples\(\)](#) (*dicee.knowledge\_graph\_embeddings.KGE method*), 176  
[find\\_missing\\_triples\(\)](#) (*dicee.scripts.serve.KGE method*), 110  
[fit\(\)](#) (*dicee.trainer.dice\_trainer.TorchDDPTrainer method*), 119  
[fit\(\)](#) (*dicee.trainer.dice\_trainer.TorchTrainer method*), 118  
[fit\(\)](#) (*dicee.trainer.torch\_trainer\_ddp.TorchDDPTrainer method*), 126  
[fit\(\)](#) (*dicee.trainer.torch\_trainer.TorchTrainer method*), 123  
[FMult](#) (class in *dicee.models*), 92  
[FMult](#) (class in *dicee.models.function\_space*), 34  
[FMult2](#) (class in *dicee.models*), 92  
[FMult2](#) (class in *dicee.models.function\_space*), 35  
[forward\(\)](#) (*dicee.BaseKGE method*), 226  
[forward\(\)](#) (*dicee.BytE method*), 224  
[forward\(\)](#) (*dicee.models.base\_model.BaseKGE method*), 16  
[forward\(\)](#) (*dicee.models.base\_model.IdentityClass static method*), 17  
[forward\(\)](#) (*dicee.models.BaseKGE method*), 66, 68, 70, 73, 77, 89, 91  
[forward\(\)](#) (*dicee.models.clifford.BaseKGE method*), 19  
[forward\(\)](#) (*dicee.models.complex.BaseKGE method*), 28  
[forward\(\)](#) (*dicee.models.dualE.BaseKGE method*), 31  
[forward\(\)](#) (*dicee.models.function\_space.BaseKGE method*), 34  
[forward\(\)](#) (*dicee.models.IdentityClass static method*), 67, 74, 79  
[forward\(\)](#) (*dicee.models.octonion.BaseKGE method*), 38  
[forward\(\)](#) (*dicee.models.octonion.IdentityClass static method*), 39  
[forward\(\)](#) (*dicee.models.pykeen\_models.BaseKGE method*), 42  
[forward\(\)](#) (*dicee.models.quaternion.BaseKGE method*), 45  
[forward\(\)](#) (*dicee.models.quaternion.IdentityClass static method*), 46  
[forward\(\)](#) (*dicee.models.real.BaseKGE method*), 50  
[forward\(\)](#) (*dicee.models.transformers.BaseKGE method*), 53  
[forward\(\)](#) (*dicee.models.transformers.Block method*), 57  
[forward\(\)](#) (*dicee.models.transformers.BytE method*), 54  
[forward\(\)](#) (*dicee.models.transformers.CausalSelfAttention method*), 56  
[forward\(\)](#) (*dicee.models.transformers.GPT method*), 58  
[forward\(\)](#) (*dicee.models.transformers.LayerNorm method*), 55  
[forward\(\)](#) (*dicee.models.transformers.MLP method*), 57  
[forward\(\)](#) (*dicee.static\_funcs.BaseKGE method*), 200  
[forward\(\)](#) (*dicee.static\_funcs.BytE method*), 198  
[forward\(\)](#) (*dicee.trainer.dice\_trainer.BaseKGE method*), 113

`forward_backward_update()` (*dicee.trainer.dice\_trainer.TorchTrainer method*), 119  
`forward_backward_update()` (*dicee.trainer.torch\_trainer.TorchTrainer method*), 123  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.BaseKGE method*), 225  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.base\_model.BaseKGE method*), 16  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.BaseKGE method*), 65, 68, 70, 73, 77, 89, 91  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.clifford.BaseKGE method*), 18  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.complex.BaseKGE method*), 27  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.dualE.BaseKGE method*), 31  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.function\_space.BaseKGE method*), 33  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.octonion.BaseKGE method*), 38  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.pykeen\_models.BaseKGE method*), 42  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.quaternion.BaseKGE method*), 45  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.real.BaseKGE method*), 49  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.transformers.BaseKGE method*), 53  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.static\_funcs.BaseKGE method*), 199  
`forward_byte_pair_encoded_k_vs_all()` (*dicee.trainer.dice\_trainer.BaseKGE method*), 113  
`forward_byte_pair_encoded_triple()` (*dicee.BaseKGE method*), 226  
`forward_byte_pair_encoded_triple()` (*dicee.models.base\_model.BaseKGE method*), 16  
`forward_byte_pair_encoded_triple()` (*dicee.models.BaseKGE method*), 66, 68, 70, 73, 77, 89, 91  
`forward_byte_pair_encoded_triple()` (*dicee.models.clifford.BaseKGE method*), 18  
`forward_byte_pair_encoded_triple()` (*dicee.models.complex.BaseKGE method*), 27  
`forward_byte_pair_encoded_triple()` (*dicee.models.dualE.BaseKGE method*), 31  
`forward_byte_pair_encoded_triple()` (*dicee.models.function\_space.BaseKGE method*), 33  
`forward_byte_pair_encoded_triple()` (*dicee.models.octonion.BaseKGE method*), 38  
`forward_byte_pair_encoded_triple()` (*dicee.models.pykeen\_models.BaseKGE method*), 42  
`forward_byte_pair_encoded_triple()` (*dicee.models.quaternion.BaseKGE method*), 45  
`forward_byte_pair_encoded_triple()` (*dicee.models.real.BaseKGE method*), 49  
`forward_byte_pair_encoded_triple()` (*dicee.models.transformers.BaseKGE method*), 53  
`forward_byte_pair_encoded_triple()` (*dicee.static\_funcs.BaseKGE method*), 200  
`forward_byte_pair_encoded_triple()` (*dicee.trainer.dice\_trainer.BaseKGE method*), 113  
`forward_k_vs_all()` (*dicee.AConEx method*), 217  
`forward_k_vs_all()` (*dicee.AConvO method*), 217  
`forward_k_vs_all()` (*dicee.AConvQ method*), 217  
`forward_k_vs_all()` (*dicee.BaseKGE method*), 226  
`forward_k_vs_all()` (*dicee.CMult method*), 208  
`forward_k_vs_all()` (*dicee.ComplEx method*), 217  
`forward_k_vs_all()` (*dicee.ConEx method*), 219  
`forward_k_vs_all()` (*dicee.ConvO method*), 218  
`forward_k_vs_all()` (*dicee.ConvQ method*), 218  
`forward_k_vs_all()` (*dicee.DeCaL method*), 213  
`forward_k_vs_all()` (*dicee.DistMult method*), 208  
`forward_k_vs_all()` (*dicee.DualE method*), 215  
`forward_k_vs_all()` (*dicee.Keci method*), 210  
`forward_k_vs_all()` (*dicee.models.AConEx method*), 71  
`forward_k_vs_all()` (*dicee.models.AConvO method*), 81  
`forward_k_vs_all()` (*dicee.models.AConvQ method*), 76  
`forward_k_vs_all()` (*dicee.models.base\_model.BaseKGE method*), 16  
`forward_k_vs_all()` (*dicee.models.BaseKGE method*), 66, 68, 70, 73, 78, 89, 91  
`forward_k_vs_all()` (*dicee.models.clifford.BaseKGE method*), 19  
`forward_k_vs_all()` (*dicee.models.clifford.CMult method*), 20  
`forward_k_vs_all()` (*dicee.models.clifford.DeCaL method*), 24  
`forward_k_vs_all()` (*dicee.models.clifford.Keci method*), 22  
`forward_k_vs_all()` (*dicee.models.CMult method*), 85  
`forward_k_vs_all()` (*dicee.models.ComplEx method*), 72  
`forward_k_vs_all()` (*dicee.models.complex.AConEx method*), 29  
`forward_k_vs_all()` (*dicee.models.complex.BaseKGE method*), 28  
`forward_k_vs_all()` (*dicee.models.complex.ComplEx method*), 30  
`forward_k_vs_all()` (*dicee.models.complex.ConEx method*), 28  
`forward_k_vs_all()` (*dicee.models.ConEx method*), 71  
`forward_k_vs_all()` (*dicee.models.ConvO method*), 80  
`forward_k_vs_all()` (*dicee.models.ConvQ method*), 76  
`forward_k_vs_all()` (*dicee.models.DeCaL method*), 86  
`forward_k_vs_all()` (*dicee.models.DistMult method*), 69  
`forward_k_vs_all()` (*dicee.models.DualE method*), 94  
`forward_k_vs_all()` (*dicee.models.dualE.BaseKGE method*), 31  
`forward_k_vs_all()` (*dicee.models.dualE.DualE method*), 32  
`forward_k_vs_all()` (*dicee.models.function\_space.BaseKGE method*), 34  
`forward_k_vs_all()` (*dicee.models.Keci method*), 83

`forward_k_vs_all()` (*dicee.models.octonion.AConvO method*), 41  
`forward_k_vs_all()` (*dicee.models.octonion.BaseKGE method*), 38  
`forward_k_vs_all()` (*dicee.models.octonion.ConvO method*), 41  
`forward_k_vs_all()` (*dicee.models.octonion.OMult method*), 40  
`forward_k_vs_all()` (*dicee.models.OMult method*), 79  
`forward_k_vs_all()` (*dicee.models.pykeen\_models.BaseKGE method*), 43  
`forward_k_vs_all()` (*dicee.models.pykeen\_models.PykeenKGE method*), 43  
`forward_k_vs_all()` (*dicee.models.PykeenKGE method*), 90  
`forward_k_vs_all()` (*dicee.models.QMult method*), 76  
`forward_k_vs_all()` (*dicee.models.quaternion.AConvQ method*), 48  
`forward_k_vs_all()` (*dicee.models.quaternion.BaseKGE method*), 45  
`forward_k_vs_all()` (*dicee.models.quaternion.ConvQ method*), 48  
`forward_k_vs_all()` (*dicee.models.quaternion.QMult method*), 48  
`forward_k_vs_all()` (*dicee.models.real.BaseKGE method*), 50  
`forward_k_vs_all()` (*dicee.models.real.DistMult method*), 50  
`forward_k_vs_all()` (*dicee.models.real.Shallom method*), 51  
`forward_k_vs_all()` (*dicee.models.real.TransE method*), 51  
`forward_k_vs_all()` (*dicee.models.Shallom method*), 69  
`forward_k_vs_all()` (*dicee.models.TransE method*), 69  
`forward_k_vs_all()` (*dicee.models.transformers.BaseKGE method*), 53  
`forward_k_vs_all()` (*dicee.OMult method*), 221  
`forward_k_vs_all()` (*dicee.PykeenKGE method*), 223  
`forward_k_vs_all()` (*dicee.QMult method*), 220  
`forward_k_vs_all()` (*dicee.Shallom method*), 221  
`forward_k_vs_all()` (*dicee.static\_funcs.AConEx method*), 191  
`forward_k_vs_all()` (*dicee.static\_funcs.AConvO method*), 191  
`forward_k_vs_all()` (*dicee.static\_funcs.AConvQ method*), 191  
`forward_k_vs_all()` (*dicee.static\_funcs.BaseKGE method*), 200  
`forward_k_vs_all()` (*dicee.static\_funcs.CMult method*), 181  
`forward_k_vs_all()` (*dicee.static\_funcs.ComplEx method*), 190  
`forward_k_vs_all()` (*dicee.static\_funcs.ConEx method*), 193  
`forward_k_vs_all()` (*dicee.static\_funcs.ConvO method*), 192  
`forward_k_vs_all()` (*dicee.static\_funcs.ConvQ method*), 192  
`forward_k_vs_all()` (*dicee.static\_funcs.DeCaL method*), 187  
`forward_k_vs_all()` (*dicee.static\_funcs.DistMult method*), 182  
`forward_k_vs_all()` (*dicee.static\_funcs.DualE method*), 189  
`forward_k_vs_all()` (*dicee.static\_funcs.Keci method*), 184  
`forward_k_vs_all()` (*dicee.static\_funcs.OMult method*), 195  
`forward_k_vs_all()` (*dicee.static\_funcs.PykeenKGE method*), 196  
`forward_k_vs_all()` (*dicee.static\_funcs.QMult method*), 194  
`forward_k_vs_all()` (*dicee.static\_funcs.Shallom method*), 195  
`forward_k_vs_all()` (*dicee.static\_funcs.TransE method*), 185  
`forward_k_vs_all()` (*dicee.trainer.dice\_trainer.BaseKGE method*), 113  
`forward_k_vs_all()` (*dicee.TransE method*), 211  
`forward_k_vs_sample()` (*dicee.AConEx method*), 217  
`forward_k_vs_sample()` (*dicee.BaseKGE method*), 226  
`forward_k_vs_sample()` (*dicee.ConEx method*), 219  
`forward_k_vs_sample()` (*dicee.DistMult method*), 208  
`forward_k_vs_sample()` (*dicee.Keci method*), 211  
`forward_k_vs_sample()` (*dicee.models.AConEx method*), 71  
`forward_k_vs_sample()` (*dicee.models.base\_model.BaseKGE method*), 16  
`forward_k_vs_sample()` (*dicee.models.BaseKGE method*), 66, 68, 70, 73, 78, 89, 91  
`forward_k_vs_sample()` (*dicee.models.clifford.BaseKGE method*), 19  
`forward_k_vs_sample()` (*dicee.models.clifford.Keci method*), 22  
`forward_k_vs_sample()` (*dicee.models.complex.AConEx method*), 29  
`forward_k_vs_sample()` (*dicee.models.complex.BaseKGE method*), 28  
`forward_k_vs_sample()` (*dicee.models.complex.ConEx method*), 28  
`forward_k_vs_sample()` (*dicee.models.ConEx method*), 71  
`forward_k_vs_sample()` (*dicee.models.DistMult method*), 69  
`forward_k_vs_sample()` (*dicee.models.dualE.BaseKGE method*), 31  
`forward_k_vs_sample()` (*dicee.models.function\_space.BaseKGE method*), 34  
`forward_k_vs_sample()` (*dicee.models.Keci method*), 83  
`forward_k_vs_sample()` (*dicee.models.octonion.BaseKGE method*), 38  
`forward_k_vs_sample()` (*dicee.models.pykeen\_models.BaseKGE method*), 43  
`forward_k_vs_sample()` (*dicee.models.pykeen\_models.PykeenKGE method*), 44  
`forward_k_vs_sample()` (*dicee.models.PykeenKGE method*), 90  
`forward_k_vs_sample()` (*dicee.models.QMult method*), 76  
`forward_k_vs_sample()` (*dicee.models.quaternion.BaseKGE method*), 45



`forward_k_vs_sample()` (*dicee.models.quaternion.QMult method*), 48  
`forward_k_vs_sample()` (*dicee.models.real.BaseKGE method*), 50  
`forward_k_vs_sample()` (*dicee.models.real.DistMult method*), 51  
`forward_k_vs_sample()` (*dicee.models.transformers.BaseKGE method*), 53  
`forward_k_vs_sample()` (*dicee.PykeenKGE method*), 223  
`forward_k_vs_sample()` (*dicee.QMult method*), 220  
`forward_k_vs_sample()` (*dicee.static\_funcs.AConEx method*), 191  
`forward_k_vs_sample()` (*dicee.static\_funcs.BaseKGE method*), 200  
`forward_k_vs_sample()` (*dicee.static\_funcs.ConEx method*), 193  
`forward_k_vs_sample()` (*dicee.static\_funcs.DistMult method*), 182  
`forward_k_vs_sample()` (*dicee.static\_funcs.Keci method*), 185  
`forward_k_vs_sample()` (*dicee.static\_funcs.PykeenKGE method*), 197  
`forward_k_vs_sample()` (*dicee.static\_funcs.QMult method*), 194  
`forward_k_vs_sample()` (*dicee.trainer.dice\_trainer.BaseKGE method*), 113  
`forward_k_vs_with_explicit()` (*dicee.Keci method*), 210  
`forward_k_vs_with_explicit()` (*dicee.models.clifford.Keci method*), 22  
`forward_k_vs_with_explicit()` (*dicee.models.Keci method*), 83  
`forward_k_vs_with_explicit()` (*dicee.static\_funcs.Keci method*), 184  
`forward_triples()` (*dicee.AConEx method*), 217  
`forward_triples()` (*dicee.AConvO method*), 217  
`forward_triples()` (*dicee.AConvQ method*), 217  
`forward_triples()` (*dicee.BaseKGE method*), 226  
`forward_triples()` (*dicee.CMult method*), 207  
`forward_triples()` (*dicee.ConEx method*), 219  
`forward_triples()` (*dicee.ConvO method*), 218  
`forward_triples()` (*dicee.ConvQ method*), 218  
`forward_triples()` (*dicee.DeCaL method*), 212  
`forward_triples()` (*dicee.DualE method*), 215  
`forward_triples()` (*dicee.Keci method*), 211  
`forward_triples()` (*dicee.LFMult method*), 221  
`forward_triples()` (*dicee.models.AConEx method*), 71  
`forward_triples()` (*dicee.models.AConvO method*), 80  
`forward_triples()` (*dicee.models.AConvQ method*), 76  
`forward_triples()` (*dicee.models.base\_model.BaseKGE method*), 16  
`forward_triples()` (*dicee.models.BaseKGE method*), 66, 68, 70, 73, 77, 89, 91  
`forward_triples()` (*dicee.models.clifford.BaseKGE method*), 19  
`forward_triples()` (*dicee.models.clifford.CMult method*), 20  
`forward_triples()` (*dicee.models.clifford.DeCaL method*), 24  
`forward_triples()` (*dicee.models.clifford.Keci method*), 23  
`forward_triples()` (*dicee.models.CMult method*), 84  
`forward_triples()` (*dicee.models.complex.AConEx method*), 29  
`forward_triples()` (*dicee.models.complex.BaseKGE method*), 28  
`forward_triples()` (*dicee.models.complex.ConEx method*), 28  
`forward_triples()` (*dicee.models.ConEx method*), 71  
`forward_triples()` (*dicee.models.ConvO method*), 80  
`forward_triples()` (*dicee.models.ConvQ method*), 76  
`forward_triples()` (*dicee.models.DeCaL method*), 85  
`forward_triples()` (*dicee.models.DualE method*), 94  
`forward_triples()` (*dicee.models.dualE.BaseKGE method*), 31  
`forward_triples()` (*dicee.models.dualE.DualE method*), 32  
`forward_triples()` (*dicee.models.FMult method*), 92  
`forward_triples()` (*dicee.models.FMult2 method*), 92  
`forward_triples()` (*dicee.models.function\_space.BaseKGE method*), 34  
`forward_triples()` (*dicee.models.function\_space.FMult method*), 34  
`forward_triples()` (*dicee.models.function\_space.FMult2 method*), 35  
`forward_triples()` (*dicee.models.function\_space.GFMult method*), 35  
`forward_triples()` (*dicee.models.function\_space.LFMult method*), 35  
`forward_triples()` (*dicee.models.function\_space.LFMult1 method*), 35  
`forward_triples()` (*dicee.models.GFMult method*), 92  
`forward_triples()` (*dicee.models.Keci method*), 83  
`forward_triples()` (*dicee.models.LFMult method*), 93  
`forward_triples()` (*dicee.models.LFMult1 method*), 92  
`forward_triples()` (*dicee.models.octonion.AConvO method*), 41  
`forward_triples()` (*dicee.models.octonion.BaseKGE method*), 38  
`forward_triples()` (*dicee.models.octonion.ConvO method*), 41  
`forward_triples()` (*dicee.models.pyke method*), 69  
`forward_triples()` (*dicee.models.pykeen\_models.BaseKGE method*), 42  
`forward_triples()` (*dicee.models.pykeen\_models.PykeenKGE method*), 43

`forward_triples()` (*dicee.models.PykeenKGE method*), 90  
`forward_triples()` (*dicee.models.quaternion.AConvQ method*), 48  
`forward_triples()` (*dicee.models.quaternion.BaseKGE method*), 45  
`forward_triples()` (*dicee.models.quaternion.ConvQ method*), 48  
`forward_triples()` (*dicee.models.real.BaseKGE method*), 50  
`forward_triples()` (*dicee.models.real.Pyke method*), 51  
`forward_triples()` (*dicee.models.real.Shallom method*), 51  
`forward_triples()` (*dicee.models.Shallom method*), 69  
`forward_triples()` (*dicee.models.transformers.BaseKGE method*), 53  
`forward_triples()` (*dicee.Pyke method*), 208  
`forward_triples()` (*dicee.PykeenKGE method*), 223  
`forward_triples()` (*dicee.Shallom method*), 221  
`forward_triples()` (*dicee.static\_funcs.AConEx method*), 191  
`forward_triples()` (*dicee.static\_funcs.AConvO method*), 191  
`forward_triples()` (*dicee.static\_funcs.AConvQ method*), 191  
`forward_triples()` (*dicee.static\_funcs.BaseKGE method*), 200  
`forward_triples()` (*dicee.static\_funcs.CMult method*), 181  
`forward_triples()` (*dicee.static\_funcs.ConEx method*), 193  
`forward_triples()` (*dicee.static\_funcs.ConvO method*), 192  
`forward_triples()` (*dicee.static\_funcs.ConvQ method*), 192  
`forward_triples()` (*dicee.static\_funcs.DeCaL method*), 186  
`forward_triples()` (*dicee.static\_funcs.DualE method*), 189  
`forward_triples()` (*dicee.static\_funcs.Keci method*), 185  
`forward_triples()` (*dicee.static\_funcs.LFMult method*), 195  
`forward_triples()` (*dicee.static\_funcs.Pyke method*), 182  
`forward_triples()` (*dicee.static\_funcs.PykeenKGE method*), 197  
`forward_triples()` (*dicee.static\_funcs.Shallom method*), 195  
`forward_triples()` (*dicee.trainer.dice\_trainer.BaseKGE method*), 113  
`from_pretrained()` (*dicee.models.transformers.GPT class method*), 58  
`func_triple_to_bpe_representation()` (*dicee.evaluator.KG method*), 160  
`func_triple_to_bpe_representation()` (*dicee.executer.KG method*), 162  
`func_triple_to_bpe_representation()` (*dicee.knowledge\_graph.KG method*), 168  
`func_triple_to_bpe_representation()` (*dicee.trainer.dice\_trainer.KG method*), 120  
`function()` (*dicee.models.FMult2 method*), 92  
`function()` (*dicee.models.function\_space.FMult2 method*), 35

## G

`generate()` (*dicee.BytE method*), 224  
`generate()` (*dicee.eval\_static\_funcs.KGE method*), 155  
`generate()` (*dicee.KGE method*), 229  
`generate()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 172  
`generate()` (*dicee.models.transformers.BytE method*), 54  
`generate()` (*dicee.scripts.serve.KGE method*), 107  
`generate()` (*dicee.static\_funcs.BytE method*), 198  
`generate_queries()` (*dicee.query\_generator.QueryGenerator method*), 177  
`generate_queries()` (*dicee.QueryGenerator method*), 243  
`get()` (*dicee.scripts.serve.NeuralSearcher method*), 111  
`get_aswa_state_dict()` (*dicee.callbacks.ASWA method*), 140  
`get_aswa_state_dict()` (*dicee.trainer.dice\_trainer.ASWA method*), 114  
`get_bpe_head_and_relation_representation()` (*dicee.BaseKGE method*), 226  
`get_bpe_head_and_relation_representation()` (*dicee.models.base\_model.BaseKGE method*), 17  
`get_bpe_head_and_relation_representation()` (*dicee.models.BaseKGE method*), 66, 68, 71, 74, 78, 89, 91  
`get_bpe_head_and_relation_representation()` (*dicee.models.clifford.BaseKGE method*), 19  
`get_bpe_head_and_relation_representation()` (*dicee.models.complex.BaseKGE method*), 28  
`get_bpe_head_and_relation_representation()` (*dicee.models.dualE.BaseKGE method*), 31  
`get_bpe_head_and_relation_representation()` (*dicee.models.function\_space.BaseKGE method*), 34  
`get_bpe_head_and_relation_representation()` (*dicee.models.octonion.BaseKGE method*), 38  
`get_bpe_head_and_relation_representation()` (*dicee.models.pykeen\_models.BaseKGE method*), 43  
`get_bpe_head_and_relation_representation()` (*dicee.models.quaternion.BaseKGE method*), 46  
`get_bpe_head_and_relation_representation()` (*dicee.models.real.BaseKGE method*), 50  
`get_bpe_head_and_relation_representation()` (*dicee.models.transformers.BaseKGE method*), 53  
`get_bpe_head_and_relation_representation()` (*dicee.static\_funcs.BaseKGE method*), 200  
`get_bpe_head_and_relation_representation()` (*dicee.trainer.dice\_trainer.BaseKGE method*), 114  
`get_bpe_token_representation()` (*dicee.abstracts.BaseInteractiveKGE method*), 130  
`get_bpe_token_representation()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 169  
`get_callbacks()` (*in module dicee.trainer.dice\_trainer*), 120  
`get_default_arguments()` (*in module dicee.analyse\_experiments*), 135



`get_default_arguments()` (in module `dicee.scripts.index`), 103  
`get_default_arguments()` (in module `dicee.scripts.run`), 106  
`get_default_arguments()` (in module `dicee.scripts.serve`), 111  
`get_domain_of_relation()` (`dicee.abstracts.BaseInteractiveKGE` method), 130  
`get_domain_of_relation()` (`dicee.knowledge_graph_embeddings.BaseInteractiveKGE` method), 169  
`get_ee_vocab()` (in module `dicee`), 226  
`get_ee_vocab()` (in module `dicee.read_preprocess_save_load_kg.preprocess`), 96  
`get_ee_vocab()` (in module `dicee.read_preprocess_save_load_kg.util`), 101  
`get_ee_vocab()` (in module `dicee.static_funcs`), 200  
`get_ee_vocab()` (in module `dicee.static_preprocess_funcs`), 204  
`get_embeddings()` (`dicee.BaseKGE` method), 226  
`get_embeddings()` (`dicee.models.base_model.BaseKGE` method), 17  
`get_embeddings()` (`dicee.models.BaseKGE` method), 66, 68, 71, 74, 78, 89, 92  
`get_embeddings()` (`dicee.models.clifford.BaseKGE` method), 19  
`get_embeddings()` (`dicee.models.complex.BaseKGE` method), 28  
`get_embeddings()` (`dicee.models.dualE.BaseKGE` method), 31  
`get_embeddings()` (`dicee.models.function_space.BaseKGE` method), 34  
`get_embeddings()` (`dicee.models.octonion.BaseKGE` method), 38  
`get_embeddings()` (`dicee.models.pykeen_models.BaseKGE` method), 43  
`get_embeddings()` (`dicee.models.quaternion.BaseKGE` method), 46  
`get_embeddings()` (`dicee.models.real.BaseKGE` method), 50  
`get_embeddings()` (`dicee.models.real.Shallom` method), 51  
`get_embeddings()` (`dicee.models.Shallom` method), 69  
`get_embeddings()` (`dicee.models.transformers.BaseKGE` method), 53  
`get_embeddings()` (`dicee.Shallom` method), 221  
`get_embeddings()` (`dicee.static_funcs.BaseKGE` method), 200  
`get_embeddings()` (`dicee.static_funcs.Shallom` method), 195  
`get_embeddings()` (`dicee.trainer.dice_trainer.BaseKGE` method), 114  
`get_entity_embeddings()` (`dicee.abstracts.BaseInteractiveKGE` method), 131  
`get_entity_embeddings()` (`dicee.knowledge_graph_embeddings.BaseInteractiveKGE` method), 170  
`get_entity_index()` (`dicee.abstracts.BaseInteractiveKGE` method), 131  
`get_entity_index()` (`dicee.knowledge_graph_embeddings.BaseInteractiveKGE` method), 170  
`get_er_vocab()` (in module `dicee`), 226  
`get_er_vocab()` (in module `dicee.read_preprocess_save_load_kg.preprocess`), 96  
`get_er_vocab()` (in module `dicee.read_preprocess_save_load_kg.util`), 101  
`get_er_vocab()` (in module `dicee.static_funcs`), 200  
`get_er_vocab()` (in module `dicee.static_preprocess_funcs`), 204  
`get_eval_report()` (`dicee.abstracts.BaseInteractiveKGE` method), 130  
`get_eval_report()` (`dicee.knowledge_graph_embeddings.BaseInteractiveKGE` method), 169  
`get_head_relation_representation()` (`dicee.BaseKGE` method), 226  
`get_head_relation_representation()` (`dicee.models.base_model.BaseKGE` method), 17  
`get_head_relation_representation()` (`dicee.models.BaseKGE` method), 66, 68, 70, 74, 78, 89, 91  
`get_head_relation_representation()` (`dicee.models.clifford.BaseKGE` method), 19  
`get_head_relation_representation()` (`dicee.models.complex.BaseKGE` method), 28  
`get_head_relation_representation()` (`dicee.models.dualE.BaseKGE` method), 31  
`get_head_relation_representation()` (`dicee.models.function_space.BaseKGE` method), 34  
`get_head_relation_representation()` (`dicee.models.octonion.BaseKGE` method), 38  
`get_head_relation_representation()` (`dicee.models.pykeen_models.BaseKGE` method), 43  
`get_head_relation_representation()` (`dicee.models.quaternion.BaseKGE` method), 45  
`get_head_relation_representation()` (`dicee.models.real.BaseKGE` method), 50  
`get_head_relation_representation()` (`dicee.models.transformers.BaseKGE` method), 53  
`get_head_relation_representation()` (`dicee.static_funcs.BaseKGE` method), 200  
`get_head_relation_representation()` (`dicee.trainer.dice_trainer.BaseKGE` method), 113  
`get_kronecker_triple_representation()` (`dicee.callbacks.KronE` method), 142  
`get_kronecker_triple_representation()` (`dicee.trainer.dice_trainer.KronE` method), 116  
`get_num_params()` (`dicee.models.transformers.GPT` method), 58  
`get_padded_bpe_triple_representation()` (`dicee.abstracts.BaseInteractiveKGE` method), 130  
`get_padded_bpe_triple_representation()` (`dicee.knowledge_graph_embeddings.BaseInteractiveKGE` method), 169  
`get_queries()` (`dicee.query_generator.QueryGenerator` method), 178  
`get_queries()` (`dicee.QueryGenerator` method), 243  
`get_range_of_relation()` (`dicee.abstracts.BaseInteractiveKGE` method), 130  
`get_range_of_relation()` (`dicee.knowledge_graph_embeddings.BaseInteractiveKGE` method), 169  
`get_re_vocab()` (in module `dicee`), 226  
`get_re_vocab()` (in module `dicee.read_preprocess_save_load_kg.preprocess`), 96  
`get_re_vocab()` (in module `dicee.read_preprocess_save_load_kg.util`), 101  
`get_re_vocab()` (in module `dicee.static_funcs`), 200  
`get_re_vocab()` (in module `dicee.static_preprocess_funcs`), 204  
`get_relation_embeddings()` (`dicee.abstracts.BaseInteractiveKGE` method), 132

`get_relation_embeddings()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 171  
`get_relation_index()` (*dicee.abstracts.BaseInteractiveKGE method*), 131  
`get_relation_index()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 170  
`get_sentence_representation()` (*dicee.BaseKGE method*), 226  
`get_sentence_representation()` (*dicee.models.base\_model.BaseKGE method*), 17  
`get_sentence_representation()` (*dicee.models.BaseKGE method*), 66, 68, 70, 74, 78, 89, 91  
`get_sentence_representation()` (*dicee.models.clifford.BaseKGE method*), 19  
`get_sentence_representation()` (*dicee.models.complex.BaseKGE method*), 28  
`get_sentence_representation()` (*dicee.models.dualE.BaseKGE method*), 31  
`get_sentence_representation()` (*dicee.models.function\_space.BaseKGE method*), 34  
`get_sentence_representation()` (*dicee.models.octonion.BaseKGE method*), 38  
`get_sentence_representation()` (*dicee.models.pykeen\_models.BaseKGE method*), 43  
`get_sentence_representation()` (*dicee.models.quaternion.BaseKGE method*), 45  
`get_sentence_representation()` (*dicee.models.real.BaseKGE method*), 50  
`get_sentence_representation()` (*dicee.models.transformers.BaseKGE method*), 53  
`get_sentence_representation()` (*dicee.static\_funcs.BaseKGE method*), 200  
`get_sentence_representation()` (*dicee.trainer.dice\_trainer.BaseKGE method*), 113  
`get_transductive_entity_embeddings()` (*dicee.eval\_static\_funcs.KGE method*), 155  
`get_transductive_entity_embeddings()` (*dicee.KGE method*), 229  
`get_transductive_entity_embeddings()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 172  
`get_transductive_entity_embeddings()` (*dicee.scripts.serve.KGE method*), 107  
`get_triple_representation()` (*dicee.BaseKGE method*), 226  
`get_triple_representation()` (*dicee.models.base\_model.BaseKGE method*), 16  
`get_triple_representation()` (*dicee.models.BaseKGE method*), 66, 68, 70, 73, 78, 89, 91  
`get_triple_representation()` (*dicee.models.clifford.BaseKGE method*), 19  
`get_triple_representation()` (*dicee.models.complex.BaseKGE method*), 28  
`get_triple_representation()` (*dicee.models.dualE.BaseKGE method*), 31  
`get_triple_representation()` (*dicee.models.function\_space.BaseKGE method*), 34  
`get_triple_representation()` (*dicee.models.octonion.BaseKGE method*), 38  
`get_triple_representation()` (*dicee.models.pykeen\_models.BaseKGE method*), 43  
`get_triple_representation()` (*dicee.models.quaternion.BaseKGE method*), 45  
`get_triple_representation()` (*dicee.models.real.BaseKGE method*), 50  
`get_triple_representation()` (*dicee.models.transformers.BaseKGE method*), 53  
`get_triple_representation()` (*dicee.static\_funcs.BaseKGE method*), 200  
`get_triple_representation()` (*dicee.trainer.dice\_trainer.BaseKGE method*), 113  
`GFMult` (class in *dicee.models*), 92  
`GFMult` (class in *dicee.models.function\_space*), 34  
`GPT` (class in *dicee.models.transformers*), 57  
`GPTConfig` (class in *dicee.models.transformers*), 57  
`gpu` (*dicee.config.Namespace* attribute), 143  
`gradient_accumulation_steps` (*dicee.config.Namespace* attribute), 144  
`ground_queries()` (*dicee.query\_generator.QueryGenerator method*), 177  
`ground_queries()` (*dicee.QueryGenerator method*), 243

## H

`hidden_dropout_rate` (*dicee.config.Namespace* attribute), 145

## I

`IdentityClass` (class in *dicee.models*), 66, 74, 78  
`IdentityClass` (class in *dicee.models.base\_model*), 17  
`IdentityClass` (class in *dicee.models.octonion*), 38  
`IdentityClass` (class in *dicee.models.quaternion*), 46  
`index_triple()` (*dicee.abstracts.BaseInteractiveKGE method*), 131  
`index_triple()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 170  
`index_triples_with_pandas()` (in module *dicee.read\_preprocess\_save\_load\_kg.preprocess*), 96  
`index_triples_with_pandas()` (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 101  
`init_param` (*dicee.config.Namespace* attribute), 144  
`init_params_with_sanity_checking()` (*dicee.BaseKGE method*), 226  
`init_params_with_sanity_checking()` (*dicee.models.base\_model.BaseKGE method*), 16  
`init_params_with_sanity_checking()` (*dicee.models.BaseKGE method*), 66, 68, 70, 73, 77, 89, 91  
`init_params_with_sanity_checking()` (*dicee.models.clifford.BaseKGE method*), 19  
`init_params_with_sanity_checking()` (*dicee.models.complex.BaseKGE method*), 28  
`init_params_with_sanity_checking()` (*dicee.models.dualE.BaseKGE method*), 31  
`init_params_with_sanity_checking()` (*dicee.models.function\_space.BaseKGE method*), 34  
`init_params_with_sanity_checking()` (*dicee.models.octonion.BaseKGE method*), 38  
`init_params_with_sanity_checking()` (*dicee.models.pykeen\_models.BaseKGE method*), 42  
`init_params_with_sanity_checking()` (*dicee.models.quaternion.BaseKGE method*), 45

`init_params_with_sanity_checking()` (*dicee.models.real.BaseKGE method*), 50  
`init_params_with_sanity_checking()` (*dicee.models.transformers.BaseKGE method*), 53  
`init_params_with_sanity_checking()` (*dicee.static\_funcs.BaseKGE method*), 200  
`init_params_with_sanity_checking()` (*dicee.trainer.dice\_trainer.BaseKGE method*), 113  
`initialize_dataloader()` (*dicee.DICE\_Trainer method*), 229  
`initialize_dataloader()` (*dicee.executer.DICE\_Trainer method*), 164  
`initialize_dataloader()` (*dicee.trainer.DICE\_Trainer method*), 127  
`initialize_dataloader()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 120  
`initialize_dataset()` (*dicee.DICE\_Trainer method*), 229  
`initialize_dataset()` (*dicee.executer.DICE\_Trainer method*), 164  
`initialize_dataset()` (*dicee.trainer.DICE\_Trainer method*), 127  
`initialize_dataset()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 120  
`initialize_or_load_model()` (*dicee.DICE\_Trainer method*), 229  
`initialize_or_load_model()` (*dicee.executer.DICE\_Trainer method*), 164  
`initialize_or_load_model()` (*dicee.trainer.DICE\_Trainer method*), 127  
`initialize_or_load_model()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 120  
`initialize_trainer()` (*dicee.DICE\_Trainer method*), 229  
`initialize_trainer()` (*dicee.executer.DICE\_Trainer method*), 164  
`initialize_trainer()` (*dicee.trainer.DICE\_Trainer method*), 127  
`initialize_trainer()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 120  
`initialize_trainer()` (*in module dicee.trainer.dice\_trainer*), 120  
`input_dropout_rate` (*dicee.config.Namespace attribute*), 145  
`intialize_model()` (*in module dicee*), 227  
`intialize_model()` (*in module dicee.static\_funcs*), 201  
`is_seen()` (*dicee.abstracts.BaseInteractiveKGE method*), 131  
`is_seen()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 170  
`is_sparql_endpoint_alive()` (*in module dicee.sanity\_checkers*), 178

## K

`k_fold_cross_validation()` (*dicee.DICE\_Trainer method*), 229  
`k_fold_cross_validation()` (*dicee.executer.DICE\_Trainer method*), 164  
`k_fold_cross_validation()` (*dicee.trainer.DICE\_Trainer method*), 127  
`k_fold_cross_validation()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 120  
`k_vs_all_score()` (*dicee.ComplEx static method*), 216  
`k_vs_all_score()` (*dicee.DistMult method*), 208  
`k_vs_all_score()` (*dicee.Keci method*), 210  
`k_vs_all_score()` (*dicee.models.clifford.Keci method*), 22  
`k_vs_all_score()` (*dicee.models.ComplEx static method*), 72  
`k_vs_all_score()` (*dicee.models.complex.ComplEx static method*), 29  
`k_vs_all_score()` (*dicee.models.DistMult method*), 68  
`k_vs_all_score()` (*dicee.models.Keci method*), 83  
`k_vs_all_score()` (*dicee.models.octonion.OMult method*), 40  
`k_vs_all_score()` (*dicee.models.OMult method*), 79  
`k_vs_all_score()` (*dicee.models.QMult method*), 76  
`k_vs_all_score()` (*dicee.models.quaternion.QMult method*), 47  
`k_vs_all_score()` (*dicee.models.real.DistMult method*), 50  
`k_vs_all_score()` (*dicee.OMult method*), 221  
`k_vs_all_score()` (*dicee.QMult method*), 220  
`k_vs_all_score()` (*dicee.static\_funcs.ComplEx static method*), 190  
`k_vs_all_score()` (*dicee.static\_funcs.DistMult method*), 182  
`k_vs_all_score()` (*dicee.static\_funcs.Keci method*), 184  
`k_vs_all_score()` (*dicee.static\_funcs.OMult method*), 195  
`k_vs_all_score()` (*dicee.static\_funcs.QMult method*), 194  
`Keci` (*class in dicee*), 208  
`Keci` (*class in dicee.models*), 81  
`Keci` (*class in dicee.models.clifford*), 20  
`Keci` (*class in dicee.static\_funcs*), 182  
`KeciBase` (*class in dicee*), 208  
`KeciBase` (*class in dicee.models*), 84  
`KeciBase` (*class in dicee.models.clifford*), 23  
`KeciBase` (*class in dicee.static\_funcs*), 182  
`kernel_size` (*dicee.config.Namespace attribute*), 145  
`KG` (*class in dicee.evaluator*), 160  
`KG` (*class in dicee.executer*), 162  
`KG` (*class in dicee.knowledge\_graph*), 168  
`KG` (*class in dicee.trainer.dice\_trainer*), 119  
`KGE` (*class in dicee*), 229

KGE (class in *dicee.eval\_static\_funcs*), 155  
 KGE (class in *dicee.knowledge\_graph\_embeddings*), 172  
 KGE (class in *dicee.scripts.serve*), 107  
 KGESaveCallback (class in *dicee.callbacks*), 138  
 KronE (class in *dicee.callbacks*), 142  
 KronE (class in *dicee.trainer.dice\_trainer*), 116  
 KvsAll (class in *dicee*), 237  
 KvsAll (class in *dicee.dataset\_classes*), 148  
 kvsall\_score () (*dicee.DualE* method), 215  
 kvsall\_score () (*dicee.models.DualE* method), 94  
 kvsall\_score () (*dicee.models.dualE.DualE* method), 31  
 kvsall\_score () (*dicee.static\_funcs.DualE* method), 189  
 KvsSampleDataset (class in *dicee*), 238  
 KvsSampleDataset (class in *dicee.dataset\_classes*), 149

## L

LayerNorm (class in *dicee.models.transformers*), 55  
 LFMult (class in *dicee*), 221  
 LFMult (class in *dicee.models*), 93  
 LFMult (class in *dicee.models.function\_space*), 35  
 LFMult (class in *dicee.static\_funcs*), 195  
 LFMult1 (class in *dicee.models*), 92  
 LFMult1 (class in *dicee.models.function\_space*), 35  
 linear () (*dicee.LFMult* method), 222  
 linear () (*dicee.models.function\_space.LFMult* method), 36  
 linear () (*dicee.models.LFMult* method), 93  
 linear () (*dicee.static\_funcs.LFMult* method), 196  
 list2tuple () (*dicee.query\_generator.QueryGenerator* method), 177  
 list2tuple () (*dicee.QueryGenerator* method), 243  
 load () (*dicee.knowledge\_graph.LoadSaveToDisk* method), 168  
 load () (*dicee.read\_preprocess\_save\_load\_kg.LoadSaveToDisk* method), 103  
 load () (*dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk.LoadSaveToDisk* method), 99  
 load\_indexed\_data () (*dicee.Execute* method), 234  
 load\_indexed\_data () (*dicee.executer.Execute* method), 165  
 load\_indexed\_data () (*dicee.scripts.run.Execute* method), 104  
 load\_json () (in module *dicee*), 227  
 load\_json () (in module *dicee.abstracts*), 128  
 load\_json () (in module *dicee.executer*), 164  
 load\_json () (in module *dicee.static\_funcs*), 201  
 load\_model () (in module *dicee*), 227  
 load\_model () (in module *dicee.abstracts*), 128  
 load\_model () (in module *dicee.static\_funcs*), 201  
 load\_model\_ensemble () (in module *dicee*), 227  
 load\_model\_ensemble () (in module *dicee.abstracts*), 128  
 load\_model\_ensemble () (in module *dicee.static\_funcs*), 201  
 load\_numpy () (in module *dicee*), 228  
 load\_numpy () (in module *dicee.static\_funcs*), 202  
 load\_numpy\_ndarray () (in module *dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk*), 99  
 load\_numpy\_ndarray () (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 101  
 load\_pickle () (in module *dicee*), 227, 235  
 load\_pickle () (in module *dicee.dataset\_classes*), 146  
 load\_pickle () (in module *dicee.knowledge\_graph\_embeddings*), 172  
 load\_pickle () (in module *dicee.query\_generator*), 177  
 load\_pickle () (in module *dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk*), 99  
 load\_pickle () (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 101  
 load\_pickle () (in module *dicee.static\_funcs*), 201  
 load\_queries () (*dicee.query\_generator.QueryGenerator* method), 178  
 load\_queries () (*dicee.QueryGenerator* method), 243  
 load\_queries\_and\_answers () (*dicee.query\_generator.QueryGenerator* static method), 178  
 load\_queries\_and\_answers () (*dicee.QueryGenerator* static method), 243  
 load\_with\_pandas () (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 101  
 LoadSaveToDisk (class in *dicee.knowledge\_graph*), 168  
 LoadSaveToDisk (class in *dicee.read\_preprocess\_save\_load\_kg*), 103  
 LoadSaveToDisk (class in *dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk*), 99  
 loss\_function () (*dicee.BytE* method), 224  
 loss\_function () (*dicee.models.base\_model.BaseKGELighting* method), 12  
 loss\_function () (*dicee.models.BaseKGELighting* method), 61

`loss_function()` (*dicee.models.transformers.BytE method*), 54  
`loss_function()` (*dicee.static\_funcs.BytE method*), 198  
`lr` (*dicee.config.Namespace attribute*), 143

## M

`main()` (*in module dicee.scripts.index*), 103  
`main()` (*in module dicee.scripts.run*), 106  
`main()` (*in module dicee.scripts.serve*), 111  
`mapping_from_first_two_cols_to_third()` (*in module dicee*), 235  
`mapping_from_first_two_cols_to_third()` (*in module dicee.dataset\_classes*), 146  
`mapping_from_first_two_cols_to_third()` (*in module dicee.static\_preprocess\_funcs*), 204  
`mem_of_model()` (*dicee.models.base\_model.BaseKGELightning method*), 11  
`mem_of_model()` (*dicee.models.BaseKGELightning method*), 60  
MLP (*class in dicee.models.transformers*), 56  
`model` (*dicee.config.Namespace attribute*), 143  
module  
    *dicee*, 10  
    *dicee.abstracts*, 128  
    *dicee.analyse\_experiments*, 134  
    *dicee.callbacks*, 135  
    *dicee.config*, 143  
    *dicee.dataset\_classes*, 145  
    *dicee.eval\_static\_funcs*, 154  
    *dicee.evaluator*, 160  
    *dicee.executer*, 161  
    *dicee.knowledge\_graph*, 167  
    *dicee.knowledge\_graph\_embeddings*, 168  
    *dicee.models*, 10  
    *dicee.models.base\_model*, 10  
    *dicee.models.clifford*, 18  
    *dicee.models.complex*, 27  
    *dicee.models.dualE*, 30  
    *dicee.models.function\_space*, 33  
    *dicee.models.octonion*, 37  
    *dicee.models.pykeen\_models*, 41  
    *dicee.models.quaternion*, 44  
    *dicee.models.real*, 49  
    *dicee.models.static\_funcs*, 51  
    *dicee.models.transformers*, 52  
    *dicee.query\_generator*, 177  
    *dicee.read\_preprocess\_save\_load\_kg*, 95  
    *dicee.read\_preprocess\_save\_load\_kg.preprocess*, 95  
    *dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk*, 97  
    *dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk*, 98  
    *dicee.read\_preprocess\_save\_load\_kg.util*, 99  
    *dicee.sanity\_checkers*, 178  
    *dicee.scripts*, 103  
    *dicee.scripts.index*, 103  
    *dicee.scripts.run*, 104  
    *dicee.scripts.serve*, 106  
    *dicee.static\_funcs*, 178  
    *dicee.static\_funcs\_training*, 202  
    *dicee.static\_preprocess\_funcs*, 203  
    *dicee.trainer*, 111  
    *dicee.trainer.dice\_trainer*, 111  
    *dicee.trainer.torch\_trainer*, 121  
    *dicee.trainer.torch\_trainer\_ddp*, 123  
MultiClassClassificationDataset (*class in dicee*), 236  
MultiClassClassificationDataset (*class in dicee.dataset\_classes*), 147  
MultiLabelDataset (*class in dicee*), 235  
MultiLabelDataset (*class in dicee.dataset\_classes*), 147

## N

`n_embd` (*dicee.models.transformers.GPTConfig attribute*), 57  
`n_head` (*dicee.models.transformers.GPTConfig attribute*), 57  
`n_layer` (*dicee.models.transformers.GPTConfig attribute*), 57  
`name` (*dicee.abstracts.BaseInteractiveKGE property*), 131



name (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE* property), 170  
 Namespace (class in *dicee.config*), 143  
 neg\_ratio (*dicee.config.Namespace* attribute), 144  
 negnorm() (*dicee.eval\_static\_funcs.KGE* method), 157  
 negnorm() (*dicee.KGE* method), 232  
 negnorm() (*dicee.knowledge\_graph\_embeddings.KGE* method), 175  
 negnorm() (*dicee.scripts.serve.KGE* method), 110  
 NegSampleDataset (class in *dicee*), 238  
 NegSampleDataset (class in *dicee.dataset\_classes*), 150  
 neural\_searcher (in module *dicee.scripts.serve*), 111  
 NeuralSearcher (class in *dicee.scripts.serve*), 111  
 NodeTrainer (class in *dicee.trainer.torch\_trainer\_ddp*), 126  
 normalization (*dicee.config.Namespace* attribute), 144  
 num\_core (*dicee.config.Namespace* attribute), 144  
 num\_epochs (*dicee.config.Namespace* attribute), 143  
 num\_folds\_for\_cv (*dicee.config.Namespace* attribute), 144  
 num\_of\_output\_channels (*dicee.config.Namespace* attribute), 145  
 numpy\_data\_type\_changer() (in module *dicee*), 227  
 numpy\_data\_type\_changer() (in module *dicee.read\_preprocess\_save\_load\_kg.preprocess*), 96  
 numpy\_data\_type\_changer() (in module *dicee.static\_funcs*), 201

## O

octonion\_mul() (in module *dicee.models*), 79  
 octonion\_mul() (in module *dicee.models.octonion*), 39  
 octonion\_mul\_norm() (in module *dicee.models*), 79  
 octonion\_mul\_norm() (in module *dicee.models.octonion*), 39  
 octonion\_normalizer() (*dicee.AConvO* static method), 217  
 octonion\_normalizer() (*dicee.ConvO* static method), 218  
 octonion\_normalizer() (*dicee.models.AConvO* static method), 80  
 octonion\_normalizer() (*dicee.models.ConvO* static method), 80  
 octonion\_normalizer() (*dicee.models.octonion.AConvO* static method), 41  
 octonion\_normalizer() (*dicee.models.octonion.ConvO* static method), 41  
 octonion\_normalizer() (*dicee.models.octonion.OMult* static method), 40  
 octonion\_normalizer() (*dicee.models.OMult* static method), 79  
 octonion\_normalizer() (*dicee.OMult* static method), 221  
 octonion\_normalizer() (*dicee.static\_funcs.AConvO* static method), 191  
 octonion\_normalizer() (*dicee.static\_funcs.ConvO* static method), 192  
 octonion\_normalizer() (*dicee.static\_funcs.OMult* static method), 195  
 OMult (class in *dicee*), 220  
 OMult (class in *dicee.models*), 79  
 OMult (class in *dicee.models.octonion*), 39  
 OMult (class in *dicee.static\_funcs*), 194  
 on\_epoch\_end() (*dicee.callbacks.KGESaveCallback* method), 139  
 on\_epoch\_end() (*dicee.callbacks.PseudoLabellingCallback* method), 140  
 on\_fit\_end() (*dicee.abstracts.AbstractCallback* method), 133  
 on\_fit\_end() (*dicee.abstracts.AbstractPPECallback* method), 134  
 on\_fit\_end() (*dicee.abstracts.AbstractTrainer* method), 129  
 on\_fit\_end() (*dicee.callbacks.AbstractCallback* method), 137  
 on\_fit\_end() (*dicee.callbacks.AccumulateEpochLossCallback* method), 137  
 on\_fit\_end() (*dicee.callbacks.ASWA* method), 140  
 on\_fit\_end() (*dicee.callbacks.Eval* method), 141  
 on\_fit\_end() (*dicee.callbacks.KGESaveCallback* method), 139  
 on\_fit\_end() (*dicee.callbacks.PrintCallback* method), 138  
 on\_fit\_end() (*dicee.trainer.dice\_trainer.AccumulateEpochLossCallback* method), 118  
 on\_fit\_end() (*dicee.trainer.dice\_trainer.ASWA* method), 114  
 on\_fit\_end() (*dicee.trainer.dice\_trainer.Eval* method), 115  
 on\_fit\_end() (*dicee.trainer.dice\_trainer.PrintCallback* method), 117  
 on\_fit\_end() (*dicee.trainer.torch\_trainer\_ddp.AbstractTrainer* method), 124  
 on\_fit\_end() (*dicee.trainer.torch\_trainer.AbstractTrainer* method), 122  
 on\_fit\_start() (*dicee.abstracts.AbstractCallback* method), 132  
 on\_fit\_start() (*dicee.abstracts.AbstractPPECallback* method), 134  
 on\_fit\_start() (*dicee.abstracts.AbstractTrainer* method), 129  
 on\_fit\_start() (*dicee.callbacks.AbstractCallback* method), 136  
 on\_fit\_start() (*dicee.callbacks.Eval* method), 141  
 on\_fit\_start() (*dicee.callbacks.KGESaveCallback* method), 139  
 on\_fit\_start() (*dicee.callbacks.KronE* method), 142  
 on\_fit\_start() (*dicee.callbacks.PrintCallback* method), 138

`on_fit_start()` (*dicee.trainer.dice\_trainer.Eval method*), 115  
`on_fit_start()` (*dicee.trainer.dice\_trainer.KronE method*), 116  
`on_fit_start()` (*dicee.trainer.dice\_trainer.PrintCallback method*), 116  
`on_fit_start()` (*dicee.trainer.torch\_trainer\_ddp.AbstractTrainer method*), 124  
`on_fit_start()` (*dicee.trainer.torch\_trainer.AbstractTrainer method*), 121  
`on_init_end()` (*dicee.abstracts.AbstractCallback method*), 132  
`on_init_end()` (*dicee.callbacks.AbstractCallback method*), 136  
`on_init_start()` (*dicee.abstracts.AbstractCallback method*), 132  
`on_init_start()` (*dicee.callbacks.AbstractCallback method*), 136  
`on_train_batch_end()` (*dicee.abstracts.AbstractCallback method*), 133  
`on_train_batch_end()` (*dicee.abstracts.AbstractTrainer method*), 129  
`on_train_batch_end()` (*dicee.callbacks.AbstractCallback method*), 137  
`on_train_batch_end()` (*dicee.callbacks.Eval method*), 141  
`on_train_batch_end()` (*dicee.callbacks.KGESaveCallback method*), 139  
`on_train_batch_end()` (*dicee.callbacks.PrintCallback method*), 138  
`on_train_batch_end()` (*dicee.trainer.dice\_trainer.Eval method*), 115  
`on_train_batch_end()` (*dicee.trainer.dice\_trainer.PrintCallback method*), 117  
`on_train_batch_end()` (*dicee.trainer.torch\_trainer\_ddp.AbstractTrainer method*), 125  
`on_train_batch_end()` (*dicee.trainer.torch\_trainer.AbstractTrainer method*), 122  
`on_train_batch_start()` (*dicee.callbacks.Perturb method*), 142  
`on_train_batch_start()` (*dicee.trainer.dice\_trainer.Perturb method*), 118  
`on_train_epoch_end()` (*dicee.abstracts.AbstractCallback method*), 133  
`on_train_epoch_end()` (*dicee.abstracts.AbstractTrainer method*), 129  
`on_train_epoch_end()` (*dicee.callbacks.AbstractCallback method*), 136  
`on_train_epoch_end()` (*dicee.callbacks.ASWA method*), 140  
`on_train_epoch_end()` (*dicee.callbacks.Eval method*), 141  
`on_train_epoch_end()` (*dicee.callbacks.KGESaveCallback method*), 139  
`on_train_epoch_end()` (*dicee.callbacks.PrintCallback method*), 138  
`on_train_epoch_end()` (*dicee.models.base\_model.BaseKGELightning method*), 12  
`on_train_epoch_end()` (*dicee.models.BaseKGELightning method*), 61  
`on_train_epoch_end()` (*dicee.trainer.dice\_trainer.ASWA method*), 114  
`on_train_epoch_end()` (*dicee.trainer.dice\_trainer.Eval method*), 115  
`on_train_epoch_end()` (*dicee.trainer.dice\_trainer.PrintCallback method*), 117  
`on_train_epoch_end()` (*dicee.trainer.torch\_trainer\_ddp.AbstractTrainer method*), 125  
`on_train_epoch_end()` (*dicee.trainer.torch\_trainer.AbstractTrainer method*), 122  
`OnevsAllDataset` (*class in dicee*), 236  
`OnevsAllDataset` (*class in dicee.dataset\_classes*), 148  
`optim` (*dicee.config.Namespace attribute*), 143

## P

`p` (*dicee.config.Namespace attribute*), 145  
`parameters()` (*dicee.abstracts.BaseInteractiveKGE method*), 132  
`parameters()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 171  
`path_single_kg` (*dicee.config.Namespace attribute*), 143  
`path_to_store_single_run` (*dicee.config.Namespace attribute*), 143  
`Perturb` (*class in dicee.callbacks*), 142  
`Perturb` (*class in dicee.trainer.dice\_trainer*), 118  
`poly_NN()` (*dicee.LFMMult method*), 222  
`poly_NN()` (*dicee.models.function\_space.LFMMult method*), 36  
`poly_NN()` (*dicee.models.LFMMult method*), 93  
`poly_NN()` (*dicee.static\_funcs.LFMMult method*), 196  
`polynomial()` (*dicee.LFMMult method*), 222  
`polynomial()` (*dicee.models.function\_space.LFMMult method*), 36  
`polynomial()` (*dicee.models.LFMMult method*), 93  
`polynomial()` (*dicee.static\_funcs.LFMMult method*), 196  
`pop()` (*dicee.LFMMult method*), 222  
`pop()` (*dicee.models.function\_space.LFMMult method*), 36  
`pop()` (*dicee.models.LFMMult method*), 94  
`pop()` (*dicee.static\_funcs.LFMMult method*), 196  
`predict()` (*dicee.eval\_static\_funcs.KGE method*), 156  
`predict()` (*dicee.KGE method*), 231  
`predict()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 174  
`predict()` (*dicee.scripts.serve.KGE method*), 108  
`predict_data_loader()` (*dicee.models.base\_model.BaseKGELightning method*), 13  
`predict_data_loader()` (*dicee.models.BaseKGELightning method*), 63  
`predict_missing_head_entity()` (*dicee.eval\_static\_funcs.KGE method*), 155  
`predict_missing_head_entity()` (*dicee.KGE method*), 229

predict\_missing\_head\_entity() (dicee.knowledge\_graph\_embeddings.KGE method), 172  
 predict\_missing\_head\_entity() (dicee.scripts.serve.KGE method), 107  
 predict\_missing\_relations() (dicee.eval\_static\_funcs.KGE method), 155  
 predict\_missing\_relations() (dicee.KGE method), 230  
 predict\_missing\_relations() (dicee.knowledge\_graph\_embeddings.KGE method), 173  
 predict\_missing\_relations() (dicee.scripts.serve.KGE method), 108  
 predict\_missing\_tail\_entity() (dicee.eval\_static\_funcs.KGE method), 156  
 predict\_missing\_tail\_entity() (dicee.KGE method), 230  
 predict\_missing\_tail\_entity() (dicee.knowledge\_graph\_embeddings.KGE method), 173  
 predict\_missing\_tail\_entity() (dicee.scripts.serve.KGE method), 108  
 predict\_topk() (dicee.eval\_static\_funcs.KGE method), 156  
 predict\_topk() (dicee.KGE method), 231  
 predict\_topk() (dicee.knowledge\_graph\_embeddings.KGE method), 174  
 predict\_topk() (dicee.scripts.serve.KGE method), 109  
 prepare\_data() (dicee.CVDataModule method), 242  
 prepare\_data() (dicee.dataset\_classes.CVDataModule method), 153  
 preprocess\_with\_byte\_pair\_encoding() (dicee.knowledge\_graph.PreprocessKG method), 167  
 preprocess\_with\_byte\_pair\_encoding() (dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method), 102  
 preprocess\_with\_byte\_pair\_encoding() (dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method), 97  
 preprocess\_with\_byte\_pair\_encoding\_with\_padding() (dicee.knowledge\_graph.PreprocessKG method), 167  
 preprocess\_with\_byte\_pair\_encoding\_with\_padding() (dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method), 102  
 preprocess\_with\_byte\_pair\_encoding\_with\_padding() (dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method), 97  
 97  
 preprocess\_with\_pandas() (dicee.knowledge\_graph.PreprocessKG method), 167  
 preprocess\_with\_pandas() (dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method), 102  
 preprocess\_with\_pandas() (dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method), 97  
 preprocess\_with\_polars() (dicee.knowledge\_graph.PreprocessKG method), 168  
 preprocess\_with\_polars() (dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method), 102  
 preprocess\_with\_polars() (dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method), 97  
 preprocesses\_input\_args() (in module dicee.executer), 163  
 preprocesses\_input\_args() (in module dicee.static\_preprocess\_funcs), 204  
 PreprocessKG (class in dicee.knowledge\_graph), 167  
 PreprocessKG (class in dicee.read\_preprocess\_save\_load\_kg), 102  
 PreprocessKG (class in dicee.read\_preprocess\_save\_load\_kg.preprocess), 96  
 print\_peak\_memory() (in module dicee.trainer.torch\_trainer\_ddp), 125  
 PrintCallback (class in dicee.callbacks), 137  
 PrintCallback (class in dicee.trainer.dice\_trainer), 116  
 PseudoLabellingCallback (class in dicee.callbacks), 139  
 Pyke (class in dicee), 208  
 Pyke (class in dicee.models), 69  
 Pyke (class in dicee.models.real), 51  
 Pyke (class in dicee.static\_funcs), 182  
 pykeen\_model\_kwargs (dicee.config.Namespace attribute), 144  
 PykeenKGE (class in dicee), 222  
 PykeenKGE (class in dicee.models), 90  
 PykeenKGE (class in dicee.models.pykeen\_models), 43  
 PykeenKGE (class in dicee.static\_funcs), 196

## Q

q (dicee.config.Namespace attribute), 145  
 QMult (class in dicee), 219  
 QMult (class in dicee.models), 75  
 QMult (class in dicee.models.quaternion), 46  
 QMult (class in dicee.static\_funcs), 193  
 quaternion\_mul() (in module dicee.models), 72  
 quaternion\_mul() (in module dicee.models.quaternion), 44  
 quaternion\_mul() (in module dicee.models.static\_funcs), 52  
 quaternion\_mul\_with\_unit\_norm() (in module dicee.models), 74  
 quaternion\_mul\_with\_unit\_norm() (in module dicee.models.quaternion), 46  
 quaternion\_multiplication\_followed\_by\_inner\_product() (dicee.models.QMult method), 75  
 quaternion\_multiplication\_followed\_by\_inner\_product() (dicee.models.quaternion.QMult method), 47  
 quaternion\_multiplication\_followed\_by\_inner\_product() (dicee.QMult method), 219  
 quaternion\_multiplication\_followed\_by\_inner\_product() (dicee.static\_funcs.QMult method), 193  
 quaternion\_normalizer() (dicee.models.QMult static method), 75  
 quaternion\_normalizer() (dicee.models.quaternion.QMult static method), 47  
 quaternion\_normalizer() (dicee.QMult static method), 220  
 quaternion\_normalizer() (dicee.static\_funcs.QMult static method), 194



QueryGenerator (class in *dicee*), 243  
QueryGenerator (class in *dicee.query\_generator*), 177

## R

random\_prediction() (in module *dicee*), 227  
random\_prediction() (in module *dicee.knowledge\_graph\_embeddings*), 172  
random\_prediction() (in module *dicee.static\_funcs*), 201  
random\_seed(*dicee.config.Namespace* attribute), 144  
read\_from\_disk() (in module *dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk*), 98  
read\_from\_disk() (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 101  
read\_from\_triple\_store() (in module *dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk*), 98  
read\_from\_triple\_store() (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 101  
read\_only\_few(*dicee.config.Namespace* attribute), 144  
read\_or\_load\_kg() (*dicee.Execute* method), 234  
read\_or\_load\_kg() (*dicee.executer.Execute* method), 165  
read\_or\_load\_kg() (*dicee.scripts.run.Execute* method), 104  
read\_or\_load\_kg() (in module *dicee*), 227  
read\_or\_load\_kg() (in module *dicee.executer*), 164  
read\_or\_load\_kg() (in module *dicee.static\_funcs*), 201  
read\_preprocess\_index\_serialize\_data() (*dicee.Execute* method), 234  
read\_preprocess\_index\_serialize\_data() (*dicee.executer.Execute* method), 165  
read\_preprocess\_index\_serialize\_data() (*dicee.scripts.run.Execute* method), 104  
read\_with\_pandas() (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 100  
read\_with\_polars() (in module *dicee.read\_preprocess\_save\_load\_kg.util*), 100  
ReadFromDisk (class in *dicee.knowledge\_graph*), 167  
ReadFromDisk (class in *dicee.read\_preprocess\_save\_load\_kg*), 103  
ReadFromDisk (class in *dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk*), 98  
relations\_str(*dicee.evaluator.KG* property), 160  
relations\_str(*dicee.executer.KG* property), 162  
relations\_str(*dicee.knowledge\_graph.KG* property), 168  
relations\_str(*dicee.trainer.dice\_trainer.KG* property), 120  
reload\_dataset() (in module *dicee*), 235  
reload\_dataset() (in module *dicee.dataset\_classes*), 146  
reload\_dataset() (in module *dicee.trainer.dice\_trainer*), 118  
remove\_triples\_from\_train\_with\_condition() (*dicee.knowledge\_graph.PreprocessKG* method), 168  
remove\_triples\_from\_train\_with\_condition() (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG* method), 102  
remove\_triples\_from\_train\_with\_condition() (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG* method), 97  
residual\_convolution() (*dicee.AConEx* method), 217  
residual\_convolution() (*dicee.AConvO* method), 217  
residual\_convolution() (*dicee.AConvQ* method), 217  
residual\_convolution() (*dicee.ConEx* method), 219  
residual\_convolution() (*dicee.ConvO* method), 218  
residual\_convolution() (*dicee.ConvQ* method), 218  
residual\_convolution() (*dicee.models.AConEx* method), 71  
residual\_convolution() (*dicee.models.AConvO* method), 80  
residual\_convolution() (*dicee.models.AConvQ* method), 76  
residual\_convolution() (*dicee.models.complex.AConEx* method), 29  
residual\_convolution() (*dicee.models.complex.ConEx* method), 28  
residual\_convolution() (*dicee.models.ConEx* method), 71  
residual\_convolution() (*dicee.models.ConvO* method), 80  
residual\_convolution() (*dicee.models.ConvQ* method), 76  
residual\_convolution() (*dicee.models.octonion.AConvO* method), 41  
residual\_convolution() (*dicee.models.octonion.ConvO* method), 41  
residual\_convolution() (*dicee.models.quaternion.AConvQ* method), 48  
residual\_convolution() (*dicee.models.quaternion.ConvQ* method), 48  
residual\_convolution() (*dicee.static\_funcs.AConEx* method), 191  
residual\_convolution() (*dicee.static\_funcs.AConvO* method), 191  
residual\_convolution() (*dicee.static\_funcs.AConvQ* method), 191  
residual\_convolution() (*dicee.static\_funcs.ConEx* method), 193  
residual\_convolution() (*dicee.static\_funcs.ConvO* method), 192  
residual\_convolution() (*dicee.static\_funcs.ConvQ* method), 191  
retrieve\_embeddings() (in module *dicee.scripts.serve*), 111  
return\_multi\_hop\_query\_results() (*dicee.eval\_static\_funcs.KGE* method), 157  
return\_multi\_hop\_query\_results() (*dicee.KGE* method), 232  
return\_multi\_hop\_query\_results() (*dicee.knowledge\_graph\_embeddings.KGE* method), 175  
return\_multi\_hop\_query\_results() (*dicee.scripts.serve.KGE* method), 110  
root() (in module *dicee.scripts.serve*), 111

## S

`sample_entity()` (*dicee.abstracts.BaseInteractiveKGE method*), 131  
`sample_entity()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 170  
`sample_relation()` (*dicee.abstracts.BaseInteractiveKGE method*), 131  
`sample_relation()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 170  
`sample_triples_ratio` (*dicee.config.Namespace attribute*), 144  
`sanity_checking_with_arguments()` (*in module dicee.sanity\_checkers*), 178  
`sanity_checking_with_arguments()` (*in module dicee.static\_preprocess\_funcs*), 204  
`save()` (*dicee.abstracts.BaseInteractiveKGE method*), 131  
`save()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 170  
`save()` (*dicee.knowledge\_graph.LoadSaveToDisk method*), 168  
`save()` (*dicee.read\_preprocess\_save\_load\_kg.LoadSaveToDisk method*), 103  
`save()` (*dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk.LoadSaveToDisk method*), 99  
`save_checkpoint()` (*dicee.abstracts.AbstractTrainer static method*), 130  
`save_checkpoint()` (*dicee.trainer.torch\_trainer\_ddp.AbstractTrainer static method*), 125  
`save_checkpoint()` (*dicee.trainer.torch\_trainer.AbstractTrainer static method*), 122  
`save_checkpoint_model()` (*in module dicee*), 227  
`save_checkpoint_model()` (*in module dicee.abstracts*), 128  
`save_checkpoint_model()` (*in module dicee.callbacks*), 136  
`save_checkpoint_model()` (*in module dicee.static\_funcs*), 201  
`save_embeddings()` (*in module dicee*), 227  
`save_embeddings()` (*in module dicee.static\_funcs*), 201  
`save_embeddings_as_csv` (*dicee.config.Namespace attribute*), 143  
`save_experiment()` (*dicee.analyse\_experiments.Experiment method*), 135  
`save_model_at_every_epoch` (*dicee.config.Namespace attribute*), 144  
`save_numpy_ndarray()` (*in module dicee*), 227  
`save_numpy_ndarray()` (*in module dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk*), 99  
`save_numpy_ndarray()` (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 101  
`save_numpy_ndarray()` (*in module dicee.static\_funcs*), 201  
`save_pickle()` (*in module dicee*), 226  
`save_pickle()` (*in module dicee.callbacks*), 136  
`save_pickle()` (*in module dicee.query\_generator*), 177  
`save_pickle()` (*in module dicee.read\_preprocess\_save\_load\_kg.save\_load\_disk*), 99  
`save_pickle()` (*in module dicee.read\_preprocess\_save\_load\_kg.util*), 101  
`save_pickle()` (*in module dicee.static\_funcs*), 200  
`save_queries()` (*dicee.query\_generator.QueryGenerator method*), 178  
`save_queries()` (*dicee.QueryGenerator method*), 243  
`save_queries_and_answers()` (*dicee.query\_generator.QueryGenerator static method*), 178  
`save_queries_and_answers()` (*dicee.QueryGenerator static method*), 243  
`save_trained_model()` (*dicee.Execute method*), 234  
`save_trained_model()` (*dicee.executer.Execute method*), 165  
`save_trained_model()` (*dicee.scripts.run.Execute method*), 105  
`scalar_batch_NN()` (*dicee.LFMMult method*), 222  
`scalar_batch_NN()` (*dicee.models.function\_space.LFMMult method*), 36  
`scalar_batch_NN()` (*dicee.models.LFMMult method*), 93  
`scalar_batch_NN()` (*dicee.static\_funcs.LFMMult method*), 196  
`score()` (*dicee.CMult method*), 207  
`score()` (*dicee.ComplEx static method*), 216  
`score()` (*dicee.DistMult method*), 208  
`score()` (*dicee.Keci method*), 211  
`score()` (*dicee.models.clifford.CMult method*), 20  
`score()` (*dicee.models.clifford.Keci method*), 23  
`score()` (*dicee.models.CMult method*), 84  
`score()` (*dicee.models.ComplEx static method*), 72  
`score()` (*dicee.models.complex.ComplEx static method*), 29  
`score()` (*dicee.models.DistMult method*), 69  
`score()` (*dicee.models.Keci method*), 83  
`score()` (*dicee.models.octonion.OMult method*), 40  
`score()` (*dicee.models.OMult method*), 79  
`score()` (*dicee.models.QMult method*), 76  
`score()` (*dicee.models.quaternion.QMult method*), 47  
`score()` (*dicee.models.real.DistMult method*), 51  
`score()` (*dicee.models.real.TransE method*), 51  
`score()` (*dicee.models.TransE method*), 69  
`score()` (*dicee.OMult method*), 221  
`score()` (*dicee.QMult method*), 220  
`score()` (*dicee.static\_funcs.CMult method*), 181  
`score()` (*dicee.static\_funcs.ComplEx static method*), 190

`score()` (*dicee.static\_funcs.DistMult method*), 182  
`score()` (*dicee.static\_funcs.Keci method*), 185  
`score()` (*dicee.static\_funcs.OMult method*), 195  
`score()` (*dicee.static\_funcs.QMult method*), 194  
`score()` (*dicee.static\_funcs.TransE method*), 185  
`score()` (*dicee.TransE method*), 211  
`scoring_technique` (*dicee.config.Namespace attribute*), 144  
`search()` (*dicee.scripts.serve.NeuralSearcher method*), 111  
`search_embeddings()` (*in module dicee.scripts.serve*), 111  
`select_model()` (*in module dicee*), 227  
`select_model()` (*in module dicee.static\_funcs*), 201  
`select_model()` (*in module dicee.trainer.dice\_trainer*), 114  
`sequential_vocabulary_construction()` (*dicee.knowledge\_graph.PreprocessKG method*), 168  
`sequential_vocabulary_construction()` (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method*), 102  
`sequential_vocabulary_construction()` (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method*), 97  
`set_global_seed()` (*dicee.query\_generator.QueryGenerator method*), 177  
`set_global_seed()` (*dicee.QueryGenerator method*), 243  
`set_model_eval_mode()` (*dicee.abstracts.BaseInteractiveKGE method*), 131  
`set_model_eval_mode()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 170  
`set_model_train_mode()` (*dicee.abstracts.BaseInteractiveKGE method*), 130  
`set_model_train_mode()` (*dicee.knowledge\_graph\_embeddings.BaseInteractiveKGE method*), 170  
`setup()` (*dicee.CVDDataModule method*), 240  
`setup()` (*dicee.dataset\_classes.CVDDataModule method*), 152  
`Shallom` (*class in dicee*), 221  
`Shallom` (*class in dicee.models*), 69  
`Shallom` (*class in dicee.models.real*), 51  
`Shallom` (*class in dicee.static\_funcs*), 195  
`single_hop_query_answering()` (*dicee.eval\_static\_funcs.KGE method*), 158  
`single_hop_query_answering()` (*dicee.KGE method*), 232  
`single_hop_query_answering()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 175  
`single_hop_query_answering()` (*dicee.scripts.serve.KGE method*), 110  
`sparql_endpoint` (*dicee.config.Namespace attribute*), 143  
`start()` (*dicee.DICE\_Trainer method*), 229  
`start()` (*dicee.Execute method*), 235  
`start()` (*dicee.executor.DICE\_Trainer method*), 164  
`start()` (*dicee.executor.Execute method*), 166  
`start()` (*dicee.knowledge\_graph.PreprocessKG method*), 167  
`start()` (*dicee.knowledge\_graph.ReadFromDisk method*), 167  
`start()` (*dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method*), 102  
`start()` (*dicee.read\_preprocess\_save\_load\_kg.preprocess.PreprocessKG method*), 96  
`start()` (*dicee.read\_preprocess\_save\_load\_kg.read\_from\_disk.ReadFromDisk method*), 98  
`start()` (*dicee.read\_preprocess\_save\_load\_kg.ReadFromDisk method*), 103  
`start()` (*dicee.scripts.run.Execute method*), 105  
`start()` (*dicee.trainer.DICE\_Trainer method*), 127  
`start()` (*dicee.trainer.dice\_trainer.DICE\_Trainer method*), 120  
`storage_path` (*dicee.config.Namespace attribute*), 143  
`store()` (*in module dicee*), 227  
`store()` (*in module dicee.executor*), 164  
`store()` (*in module dicee.static\_funcs*), 201  
`store_ensemble()` (*dicee.abstracts.AbstractPPECallback method*), 134  
`swa` (*dicee.config.Namespace attribute*), 145

## T

`T()` (*dicee.DualE method*), 216  
`T()` (*dicee.models.DualE method*), 95  
`T()` (*dicee.models.dualE.DualE method*), 32  
`T()` (*dicee.static\_funcs.DualE method*), 190  
`t_conorm()` (*dicee.eval\_static\_funcs.KGE method*), 157  
`t_conorm()` (*dicee.KGE method*), 232  
`t_conorm()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 175  
`t_conorm()` (*dicee.scripts.serve.KGE method*), 110  
`t_norm()` (*dicee.eval\_static\_funcs.KGE method*), 157  
`t_norm()` (*dicee.KGE method*), 232  
`t_norm()` (*dicee.knowledge\_graph\_embeddings.KGE method*), 175  
`t_norm()` (*dicee.scripts.serve.KGE method*), 110  
`tensor_t_norm()` (*dicee.eval\_static\_funcs.KGE method*), 157  
`tensor_t_norm()` (*dicee.KGE method*), 232

tensor\_t\_norm() (dicee.knowledge\_graph\_embeddings.KGE method), 175  
 tensor\_t\_norm() (dicee.scripts.serve.KGE method), 110  
 test\_data\_loader() (dicee.models.base\_model.BaseKGELightning method), 12  
 test\_data\_loader() (dicee.models.BaseKGELightning method), 62  
 test\_epoch\_end() (dicee.models.base\_model.BaseKGELightning method), 12  
 test\_epoch\_end() (dicee.models.BaseKGELightning method), 62  
 timeit() (in module dicee), 226, 235  
 timeit() (in module dicee.dataset\_classes), 146  
 timeit() (in module dicee.executer), 164  
 timeit() (in module dicee.read\_preprocess\_save\_load\_kg.preprocess), 96  
 timeit() (in module dicee.read\_preprocess\_save\_load\_kg.util), 100  
 timeit() (in module dicee.static\_funcs), 200  
 timeit() (in module dicee.static\_preprocess\_funcs), 204  
 timeit() (in module dicee.trainer.dice\_trainer), 119  
 to\_df() (dicee.analyse\_experiments.Experiment method), 135  
 TorchDDPTrainer (class in dicee.trainer.dice\_trainer), 119  
 TorchDDPTrainer (class in dicee.trainer.torch\_trainer\_ddp), 125  
 TorchTrainer (class in dicee.trainer.dice\_trainer), 118  
 TorchTrainer (class in dicee.trainer.torch\_trainer), 123  
 train() (dicee.eval\_static\_funcs.KGE method), 159  
 train() (dicee.KGE method), 233  
 train() (dicee.knowledge\_graph\_embeddings.KGE method), 176  
 train() (dicee.scripts.serve.KGE method), 111  
 train() (dicee.trainer.torch\_trainer\_ddp.DDPTrainer method), 126  
 train() (dicee.trainer.torch\_trainer\_ddp.NodeTrainer method), 126  
 train\_data\_loader() (dicee.CVDataModule method), 240  
 train\_data\_loader() (dicee.dataset\_classes.CVDataModule method), 151  
 train\_data\_loader() (dicee.models.base\_model.BaseKGELightning method), 14  
 train\_data\_loader() (dicee.models.BaseKGELightning method), 63  
 train\_k\_vs\_all() (dicee.eval\_static\_funcs.KGE method), 159  
 train\_k\_vs\_all() (dicee.KGE method), 233  
 train\_k\_vs\_all() (dicee.knowledge\_graph\_embeddings.KGE method), 176  
 train\_k\_vs\_all() (dicee.scripts.serve.KGE method), 111  
 train\_triples() (dicee.eval\_static\_funcs.KGE method), 159  
 train\_triples() (dicee.KGE method), 233  
 train\_triples() (dicee.knowledge\_graph\_embeddings.KGE method), 176  
 train\_triples() (dicee.scripts.serve.KGE method), 111  
 trainer (dicee.config.Namespace attribute), 144  
 training\_step() (dicee.BytE method), 224  
 training\_step() (dicee.models.base\_model.BaseKGELightning method), 11  
 training\_step() (dicee.models.BaseKGELightning method), 60  
 training\_step() (dicee.models.transformers.BytE method), 54  
 training\_step() (dicee.static\_funcs.BytE method), 198  
 TransE (class in dicee), 211  
 TransE (class in dicee.models), 69  
 TransE (class in dicee.models.real), 51  
 TransE (class in dicee.static\_funcs), 185  
 transfer\_batch\_to\_device() (dicee.CVDataModule method), 241  
 transfer\_batch\_to\_device() (dicee.dataset\_classes.CVDataModule method), 152  
 trapezoid() (dicee.models.FMult2 method), 92  
 trapezoid() (dicee.models.function\_space.FMult2 method), 35  
 tri\_score() (dicee.LFMult method), 222  
 tri\_score() (dicee.models.function\_space.LFMult method), 36  
 tri\_score() (dicee.models.function\_space.LFMult1 method), 35  
 tri\_score() (dicee.models.LFMult method), 93  
 tri\_score() (dicee.models.LFMult1 method), 93  
 tri\_score() (dicee.static\_funcs.LFMult method), 196  
 triple\_score() (dicee.eval\_static\_funcs.KGE method), 157  
 triple\_score() (dicee.KGE method), 232  
 triple\_score() (dicee.knowledge\_graph\_embeddings.KGE method), 175  
 triple\_score() (dicee.scripts.serve.KGE method), 109  
 TriplePredictionDataset (class in dicee), 239  
 TriplePredictionDataset (class in dicee.dataset\_classes), 150  
 TriplePredictionDataset (class in dicee.knowledge\_graph\_embeddings), 171  
 tuple2list() (dicee.query\_generator.QueryGenerator method), 177  
 tuple2list() (dicee.QueryGenerator method), 243

## U

`unmap()` (*dicee.query\_generator.QueryGenerator method*), 177  
`unmap()` (*dicee.QueryGenerator method*), 243  
`unmap_query()` (*dicee.query\_generator.QueryGenerator method*), 177  
`unmap_query()` (*dicee.QueryGenerator method*), 243

## V

`val_dataloader()` (*dicee.models.base\_model.BaseKGELightning method*), 13  
`val_dataloader()` (*dicee.models.BaseKGELightning method*), 62  
`validate_knowledge_graph()` (in module *dicee.sanity\_checkers*), 178  
`vocab_preparation()` (*dicee.evaluator.Evaluator method*), 160  
`vocab_preparation()` (*dicee.executer.Evaluator method*), 162  
`vocab_size` (*dicee.models.transformers.GPTConfig attribute*), 57  
`vocab_to_parquet()` (in module *dicee*), 227  
`vocab_to_parquet()` (in module *dicee.static\_funcs*), 202  
`vtp_score()` (*dicee.LFMult method*), 222  
`vtp_score()` (*dicee.models.function\_space.LFMult method*), 36  
`vtp_score()` (*dicee.models.function\_space.LFMult1 method*), 35  
`vtp_score()` (*dicee.models.LFMult method*), 93  
`vtp_score()` (*dicee.models.LFMult1 method*), 93  
`vtp_score()` (*dicee.static\_funcs.LFMult method*), 196

## W

`weight_decay` (*dicee.config.Namespace attribute*), 144  
`write_links()` (*dicee.query\_generator.QueryGenerator method*), 177  
`write_links()` (*dicee.QueryGenerator method*), 243  
`write_report()` (*dicee.Execute method*), 235  
`write_report()` (*dicee.executer.Execute method*), 166  
`write_report()` (*dicee.scripts.run.Execute method*), 105