DICE Embeddings

Release 0.1.3.2

Caglar Demir

Jul 16, 2024

Contents:

1	Dicee Manual	2
2	Installation 2.1 Installation from Source	3
3	Download Knowledge Graphs	3
4	Knowledge Graph Embedding Models	3
5	How to Train	3
6	Creating an Embedding Vector Database 6.1 Learning Embeddings	5 5 6 6
7	Answering Complex Queries	6
8	Predicting Missing Links	8
9	Downloading Pretrained Models	8
10	How to Deploy	8
11	Docker	8
12	How to cite	9
13	dicee 13.1 Subpackages 13.2 Submodules 13.3 Attributes 13.4 Classes 13.5 Functions 13.6 Package Contents	10 10 94 134 135 136 137
Py	thon Module Index	175

Index 176

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

1 Dicee Manual

Version: dicee 0.1.3.2

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

Publisher and maintainer: Caglar Demir²

Contact: caglar.demir@upb.de

License: OSI Approved :: MIT License

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

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

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

- 1. Pandas³ & Co. to use parallelism at preprocessing a large knowledge graph,
- 2. PyTorch⁴ & Co. to learn knowledge graph embeddings via multi-CPUs, GPUs, TPUs or computing cluster, and
- 3. **Huggingface**⁵ to ease the deployment of pre-trained models.

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

Why PyTorch⁷ & Co. ? PyTorch is one of the most popular machine learning frameworks available at the time of writing. PytorchLightning facilitates scaling the training procedure of PyTorch without boilerplate. In our framework, we combine PyTorch⁸ & PytorchLightning⁹. Users can choose the trainer class (e.g., DDP by Pytorch) to train large knowledge graph embedding models with billions of parameters. PytorchLightning allows us to use state-of-the-art model parallelism techniques (e.g. Fully Sharded Training, FairScale, or DeepSpeed) without extra effort. With our framework, practitioners can directly use PytorchLightning for model parallelism to train gigantic embedding models.

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

- https://github.com/dice-group/dice-embeddings
- ² https://github.com/Demirrr
- 3 https://pandas.pydata.org/
- 4 https://pytorch.org/
- 5 https://huggingface.co/
- 6 https://pandas.pydata.org/
- ⁷ https://pytorch.org/
- 8 https://pytorch.org/
- 9 https://www.pytorchlightning.ai/
- 10 https://huggingface.co/gradio

2 Installation

2.1 Installation from Source

```
git clone https://github.com/dice-group/dice-embeddings.git conda create -n dice python=3.10.13 --no-default-packages && conda activate dice && \rightarrow 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
```

(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 lighning 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 lighning as a default trainer

```
# Two equivalent executions
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.
\hookrightarrow 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
# {'H01': 0.6951588502269289, 'H03': 0.9039334341906202, 'H010': 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

# {'H01': 0.9518788343558282, 'H03': 0.9988496932515337, 'H010': 1.0, 'MRR': 0.
→9753123402351737}

# Evaluate Keci on Validation set: Evaluate Keci on Validation set

# {'H01': 0.6932515337423313, 'H03': 0.9041411042944786, 'H010': 0.9754601226993865,
→'MRR': 0.8072499937521418}

# Evaluate Keci on Test set: Evaluate Keci on Test set

{'H01': 0.6951588502269289, 'H03': 0.9039334341906202, 'H010': 0.9750378214826021,
→'MRR': 0.8064032293278861}
```

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

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 <a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#type">http://www.w3.org/2002/07/owl</a>
_:1 <a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#type">http://www.w3.org/1999/02/22-rdf-syntax-ns#type</a>
<a href="http://www.w3.org/2002/07/owl#0bjectProperty">http://www.w3.org/2002/07/owl#0bjectProperty</a>
<a href="http://www.benchmark.org/family#hasParent">http://www.w3.org/1999/02/22-rdf-syntax-ons#type</a> <a href="http://www.w3.org/2002/07/owl#0bjectProperty">http://www.w3.org/2002/07/owl#0bjectProperty</a>
<a href="http://www.w3.org/2002
```

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

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
```

```
pre_trained_kge = KGE(path=result['path_experiment_folder'])
# (3) Single-hop guery 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,
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://
\rightarrowwww.w3.org/1999/02/22-rdf-syntax-ns#type")),
                                                     tnorm="min", k=5)
top_entities = [topk_entity for topk_entity, query_score in predictions]
print (top_entities)
assert "http://www.benchmark.org/family#Person" in top_entities
assert "http://www.benchmark.org/family#Father" in top_entities
assert "http://www.benchmark.org/family#Male" in top_entities
```

For more, please refer to examples/multi_hop_query_answering.

8 Predicting Missing Links

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

9 Downloading Pretrained Models

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

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

10 How to Deploy

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

11 Docker

To build the Docker image:

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

To test the Docker image:

```
docker run --rm -v ~/.local/share/dicee/KGs:/dicee/KGs dice-embeddings ./main.py --

--model AConEx --embedding_dim 16
```

¹¹ 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}
@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},
                   {Balasubramanian, Vineeth N. and Tsang, Ivor},
  editor =
  volume =
                    {Proceedings of Machine Learning Research},
  series =
                   \{17 - -19 \text{ Nov}\},
  month =
  publisher =
                {PMLR},
```

(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

13 dicee

13.1 Subpackages

dicee.models

Submodules

dicee.models.base model

Classes

BaseKGELightning	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
IdentityClass	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.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
mem_of_model() \rightarrow 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)
```

(continued from previous page)

```
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()
```

1 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 Light-ningModule 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])
```

$\texttt{test_dataloader} \; () \; \rightarrow None$

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()

1 Note

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

1 Note

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

$\textbf{val_dataloader} \; (\;) \; \to None$

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

The dataloader will be reloaded unless you return not you :paramref: `~lightning.pytorch.trainer.trainer.Trainer.reload_dataloaders_every_n epochs` a positive integer.

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

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

1 Note

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

1 Note

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

$predict_dataloader() \rightarrow None$

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()

1 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.

$\texttt{train_dataloader}\,(\,)\,\to None$

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()

1 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.

1 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 $t \circ ()$, etc.

1 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
     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.

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

```
__call__(x)
static forward(x)
```

dicee.models.clifford

Classes

CMult	$Cl_{0,0} = Real Numbers$
Keci	Base class for all neural network modules.
KeciBase	Without learning dimension scaling
DeCaL	Base class for all neural network modules.

Module Contents

```
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} 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
```

```
C1(2,0) =>
     A multivector mathbf\{a\} = a_0 + a_1 e_1 + a_2 e_2 + a_4\{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} imes mathbf{b} = a_0b_0 + a_0b_1 e_1 + a_0b_2 e_2 + a_0 b_1 e_1 e_2
            • a 1 b 0 e 1 + a 1b 1 e 1 e1 ..
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\})
          ei ^2 = +1 for i = < i = < p ej ^2 = -1 for p < j = < p+q ei ej = -eje1 for i
     eq j
          x: torch.FloatTensor with (n,d) shape
          y: torch.FloatTensor with (n,d) shape
          p: a non-negative integer p \ge 0 q: a non-negative integer q \ge 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):
```

(continued from previous page)

```
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 $t \circ ()$, etc.

1 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_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops

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

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

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

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

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

```
e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
```

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

$compute_sigma_qq(hq, rq)$

Compute sigma_ $\{qq\}$ = sum_ $\{j=1\}^{p+q-1}$ sum_ $\{k=j+1\}^{p+q}$ (h_j r_k - h_k r_j) e_j e_k sigma_ $\{q\}$ 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_q = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))
```

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

```
e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
```

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

```
compute_sigma_pq(*, hp, hq, rp, rq)
                  sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j - h_j r_i) e_i e_j
                  results = [] sigma_pq = torch.zeros(b, r, p, q) for i in range(p):
                                 for j in range(q):
                                               sigma_pq[:, :, i, j] = hp[:, :, i] * rq[:, :, j] - hq[:, :, j] * rp[:, :, i]
                  print(sigma_pq.shape)
apply_coefficients(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 h_j e_j r = r_0 + sum_{i=1}^p r_i e_i + sum_{j=p+1}^n h_j e_j r = r_0 + sum_{i=1}^n h_j e_j r = r_0 + sum_{j=1}^n h_j e_j r = r_0 + sum_{j
                                 sum_{j=p+1}^{p+q} r_j e_j
                                 ei ^2 = +1 for i =< i =< p ej ^2 = -1 for p < j =< p+q ei ej = -eje1 for i
                  eq j
                                 h r = sigma_0 + sigma_p + sigma_q + sigma_{pp} + sigma_{q} + sig
                                 (1) sigma_0 = h_0 r_0 + sum_{i=1}^p (h_0 r_i) e_i - sum_{j=p+1}^{p+q} (h_j r_j) e_j
                                 (2) sigma_p = sum_{i=1}^p (h_0 r_i + h_i r_0) e_i
                                 (3) sigma_q = sum_{j=p+1}^{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)
```

Kvsall training

 $forward_k_vs_all(x: torch.Tensor) \rightarrow torch.FloatTensor$

(2) Construct head entity and relation embeddings according to $Cl_{p,q}(mathbb{R}^d)$.

(1) Retrieve real-valued embedding vectors for heads and relations 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, |E|) shape

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

Parameter

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)
```

(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.

1 Note

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

Variables

training $(b \circ o 1)$ – 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) \longrightarrow 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) 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) s4 = \sum_{i=p+1}^{p+q+r} (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) s4 = \sum_{i=p+q+1}^{p+q+r} (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) s4 = \sum_{i=p+q+1}^{p+q+r} (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) s4 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} ($$

and return:

$$sigma_0t = \sigma_0 \cdot t_0 = s0 + s1 - s2s3, s4ands5$$

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) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_j) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j'=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_j - h_{i'} r_j) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j'=p+1}^{p+q} \sum_{j'=j+1}^{p+q} (h_j r_j - h_{i'} r_j) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j'=p+1}^{p+q} \sum_{j'=p+1}^{p+q} (h_j r_j - h_{i'} r_j) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j'=p+1}^{p+q} \sum_{j'=p+1}^{p+q} (h_j r_j - h_{i'} r_j) (models the interactions between e_i and e'_i for 1 <= i, i' <= p) \sigma_q q = \sum_{j'=p+1}^{p+q} \sum_{j'=p+1}^{p+q} (h_j r_j - h_{i'} r_j) (models the interactions between e_i and e'_i for 1 <= i, i' <$$

For different base vector interactions, we have

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

forward_k_vs_all (x: torch.Tensor) \rightarrow 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 funcitons are identical Parameter — x: torch.LongTensor with (n,) shape :rtype: torch.FloatTensor with (n, |E|) shape

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

Multiplying a base vector with its scalar coefficient

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

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

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

Parameter

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

returns

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

$compute_sigma_pp(hp, rp)$

Compute .. math:

```
\label{eq:sigma_pp}^* = \sum_{i=1}^{p-1}\sum_{i'=i+1}^{p} (x_{i'}-x_{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,

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

 $compute_sigma_qq(hq, rq)$

Compute

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

sigma_{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):

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

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

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

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

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

ConEx	Convolutional ComplEx Knowledge Graph Embeddings
AConEx	Additive Convolutional ComplEx Knowledge Graph Embeddings
ComplEx	Base class for all neural network modules.

Module Contents

```
class dicee.models.complex.ConEx (args)
     Bases: dicee.models.base_model.BaseKGE
     Convolutional ComplEx Knowledge Graph Embeddings
     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
     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.

1 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.

Parameters

- emb_h
- emb_r
- emb E

 $\textbf{forward_k_vs_all} \ (\textit{x: torch.LongTensor}) \ \rightarrow \textbf{torch.FloatTensor}$

dicee.models.dualE

Classes

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

Module Contents

class dicee.models.dualE.DualE(args)

Bases: dicee.models.base_model.BaseKGE

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

kvsall_score (
$$e_1h, e_2h, e_3h, e_4h, e_5h, e_6h, e_7h, e_8h, e_1t, e_2t, e_3t, e_4t, e_5t, e_6t, e_7t, e_8t, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8) \to \text{torch.tensor}$$

KvsAll scoring function

Input

x: torch.LongTensor with (n,) shape

Output

torch.FloatTensor with (n) shape

 $forward_triples(idx_triple: torch.tensor) \rightarrow torch.tensor$

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) \rightarrow torch.tensor

Transpose function

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

dicee.models.function_space

Classes

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

Module Contents

```
forward_triples (idx\_triple: torch.Tensor) \rightarrow torch.Tensor
              Parameters
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) \rightarrow \text{torch.FloatTensor}
     function (list_W, list_b)
     trapezoid (list_W, list_b)
     forward\_triples(idx\_triple: torch.Tensor) \rightarrow torch.Tensor
              Parameters
                  ¥
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\}^{k=d-1}wk e^{kix}\}, and use the three differents scoring function as in the paper to evaluate
     the score
     forward_triples (idx_triple)
              Parameters
     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 = sigma(wh^T x + bh), r = sigma(wr^T x + br),
          t = sigma(wt^T x + bt)
     linear(x, w, b)
```

$scalar_batch_NN(a, b, c)$

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

tri_score (coeff_h, coeff_r, coeff_t)

this part implement the trilinear scoring techniques:

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

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

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

- 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 $[0][0] + \text{coeff}[0][1]x + ... + \text{coeff}[0][d]x^d$,

$$coeff[1][0] + coeff[1][1]x + ... + coeff[1][d]x^d$$

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,
$$coeff[1][0] + coeff[1][1]x +...+ coeff[1][d]x^d)$$

dicee.models.octonion

Classes

OMult	Base class for all neural network modules.
Conv0	Base class for all neural network modules.
AConv0	Additive Convolutional Octonion Knowledge Graph Embeddings

Functions

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

Module Contents

```
dicee.models.octonion.octonion_mul(*, O_1, O_2)
dicee.models.octonion.octonion_mul_norm(*, O_1, O_2)
class dicee.models.octonion.OMult(args)
    Bases: dicee.models.base_model.BaseKGE
```

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing 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.

1 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.

```
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.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.

1 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.

```
{\tt residual\_convolution}\,(O\_1,\,O\_2)
```

```
forward_triples (x: torch.Tensor) \rightarrow torch.Tensor
```

Parameters

x

forward_k_vs_all (x: torch.Tensor)

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

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

Bases: dicee.models.base_model.BaseKGE

Additive Convolutional Octonion Knowledge Graph Embeddings

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)

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

Parameters

x

forward_k_vs_all (x: torch.Tensor)

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

dicee.models.pykeen_models

Classes

PykeenKGE

A class for using knowledge graph embedding models implemented in Pykeen

Module Contents

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

Bases: dicee.models.base_model.BaseKGE

A class for using knowledge graph embedding models implemented in Pykeen

Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Pykeen_HolE: Pykeen_HolE:

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, $r = self.get_head_relation_representation(x) # (2) Reshape (1). if <math>self.last_dim > 0$:

 $\label{eq:hamma} h = h.reshape(len(x), self.embedding_dim, self.last_dim) \\ r = r.reshape(len(x), self.embedding_dim, se$

(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) \rightarrow torch.FloatTensor
```

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

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

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

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

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

dicee.models.quaternion

Classes

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

Functions

```
\begin{array}{ll} \textit{quaternion\_mul\_with\_unit\_norm(*, & Q\_1, \\ Q\_2)} \end{array}
```

Module Contents

```
dicee.models.quaternion.quaternion_mul_with_unit_norm(*,Q_1,Q_2)
class dicee.models.quaternion.QMult(args)
    Bases: dicee.models.base model.BaseKGE
```

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continued from previous page)

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

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

1 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.

 $\verb"quaternion_multiplication_followed_by_inner_product" (h, r, t)$

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:

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

Parameters

 \mathbf{x} – The vector.

Returns

The normalized vector.

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

 $residual_convolution(Q_1, Q_2)$

 $\textbf{forward_triples} \ (\textit{indexed_triple: torch.Tensor}) \ \rightarrow \textbf{torch.Tensor}) \ \rightarrow \textbf{torch.Tensor}$

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)

 $forward_triples(indexed_triple: torch.Tensor) \rightarrow torch.Tensor$

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

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

Module Contents

```
class dicee.models.real.DistMult(args)
     Bases: dicee.models.base_model.BaseKGE
     Embedding Entities and Relations for Learning and Inference in Knowledge Bases https://arxiv.org/abs/1412.6575
     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) \rightarrow 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) \rightarrow \text{torch.FloatTensor}
     forward_triples (x) \rightarrow \text{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

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

Module Contents

```
\label{eq:diceemodels.static_funcs.quaternion_mul} \begin{subarray}{ll} $(*,Q\_1,Q\_2)$ & $\to$ Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor] \\ Perform quaternion multiplication :param Q_1: :param Q_2: :return: \\ \end{subarray}
```

dicee.models.transformers

Classes

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

Module Contents

```
class dicee.models.transformers.BytE(*args, **kwargs)
Bases: dicee.models.base_model.BaseKGE
```

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)

(continued from previous page)

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

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

1 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

```
\mathbf{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()
```

1 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):
```

(continues on next page)

(continued from previous page)

```
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.

1 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)

```
\textbf{class} \ \texttt{dicee.models.transformers.MLP} \ (\textit{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.

1 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.

1 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.

1 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.

Classes

BaseKGELightning	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
IdentityClass	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
DistMult	Embedding Entities and Relations for Learning and Infer-
	ence in Knowledge Bases
TransE	Translating Embeddings for Modeling

continues on next page

Table 1 - continued from previous page

Table 1 centinae	a nom previous page
Shallom	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
Pyke	A Physical Embedding Model for Knowledge Graphs
BaseKGE	Base class for all neural network modules.
ConEx	Convolutional ComplEx Knowledge Graph Embeddings
AConEx	Additive Convolutional ComplEx Knowledge Graph Em-
110011211	beddings
ComplEx	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
IdentityClass	Base class for all neural network modules.
QMult	Base class for all neural network modules.
ConvO	Convolutional Quaternion Knowledge Graph Embed-
Convg	dings
AConvQ	Additive Convolutional Quaternion Knowledge Graph Embeddings
BaseKGE	Base class for all neural network modules.
IdentityClass	Base class for all neural network modules.
OMult	Base class for all neural network modules.
ConvO	Base class for all neural network modules.
AConvO	Additive Convolutional Octonion Knowledge Graph Em-
ACONVO	beddings
Keci	Base class for all neural network modules.
KeciBase	Without learning dimension scaling
CMult	$Cl_{0,0} = Real Numbers$
DeCaL	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
PykeenKGE	A class for using knowledge graph embedding models im-
	plemented in Pykeen
BaseKGE	Base class for all neural network modules.
FMult	Learning Knowledge Neural Graphs
GFMult	Learning Knowledge Neural Graphs
FMult2	Learning Knowledge Neural Graphs
LFMult1	Embedding with trigonometric functions. We represent
	all entities and relations in the complex number space as:
LFMult	Embedding with polynomial functions. We represent all
	entities and relations in the polynomial space as:
DualE	Dual Quaternion Knowledge Graph Embeddings
	(https://ojs.aaai.org/index.php/AAAI/article/download/
	16850/16657)
	,

Functions

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

Package Contents

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

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        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.

1 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() \rightarrow 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()
```

1 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 Light-ningModule and access them in this hook:

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

(continues on next page)

(continued from previous page)

```
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])

$\textbf{test_dataloader} \; () \; \rightarrow None$

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.

A Warning

do not assign state in prepare_data

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

1 Note

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

1 Note

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

$val_dataloader() \rightarrow None$

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()

1 Note

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

1 Note

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

$predict_dataloader() \rightarrow None$

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()

1 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.

$\textbf{train_dataloader} \, (\,) \, \to None$

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.



Marning

do not assign state in prepare_data

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



1 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
```

(continues on next page)

(continued from previous page)

```
"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.

1 Note

Some things to know:

- Lightning calls .backward() and .step() automatically in case of automatic optimization.
- If a learning rate scheduler is specified in <code>configure_optimizers()</code> 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.

1 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

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

byte pair encoded neural link predictors

Parameters

init_params_with_sanity_checking()

Parameters

- x
- y_idx
- ordered_bpe_entities

 $\textbf{forward_triples} \ (\textit{x: torch.LongTensor}) \ \rightarrow \textbf{torch.Tensor}$

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.

1 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)
```

Base class for all neural network modules.

Bases: BaseKGELightning

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)

(continued from previous page)

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

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

1 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()
```

Parameters

- x
- y_idx
- ordered_bpe_entities

 $\textbf{forward_triples} \ (\textit{x: torch.LongTensor}) \ \rightarrow \textbf{torch.Tensor}$

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)
     \texttt{get\_embeddings}() \rightarrow 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) \rightarrow 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) \rightarrow \text{torch.FloatTensor}
     forward_triples (x) \rightarrow \text{torch.FloatTensor}
              Parameters
                  ¥
              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.

1 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

```
\verb"init_params_with_sanity_checking" ()
```

Parameters

• x

```
y_idx
                 • ordered_bpe_entities
     forward_triples (x: torch.LongTensor) → torch.Tensor
             Parameters
     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
                 \mathbf{x} (B \times 2 \times T)
     get embeddings() → Tuple[numpy.ndarray, numpy.ndarray]
class dicee.models.ConEx (args)
     Bases: dicee.models.base_model.BaseKGE
     Convolutional ComplEx Knowledge Graph Embeddings
```

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**

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

class dicee.models.AConEx(args)

x

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:

Base class for all neural network modules.

Your models should also subclass this class.

Bases: dicee.models.base_model.BaseKGE

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 $t \circ ()$, etc.

1 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.

Parameters

emb_h

```
• emb_r
```

• emb E

 $forward_k_vs_all(x: torch.LongTensor) \rightarrow torch.FloatTensor$

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

 \rightarrow 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 $t \circ ()$, etc.

1 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
     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
                 \mathbf{x} (B \times 2 \times T)
     get embeddings() → Tuple[numpy.ndarray, numpy.ndarray]
class dicee.models.IdentityClass(args=None)
     Bases: torch.nn.Module
```

Base class for all neural network modules.

Your models should also subclass this class.

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

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

class Model(nn.Module):
    def __init__(self):
        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 $t \circ ()$, etc.



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)
```

Base class for all neural network modules.

Bases: dicee.models.base_model.BaseKGE

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.

1 Note

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

Variables

training $(b \circ \circ 1)$ – 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:

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

Parameters

 \mathbf{x} – The vector.

Returns

The normalized vector.

 $k_vs_all_score$ (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)

 $forward_triples (indexed_triple: torch.Tensor) \rightarrow torch.Tensor$

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)
```

 $\textbf{forward_triples} \ (\textit{indexed_triple: torch.Tensor}) \ \rightarrow \textbf{torch.Tensor}) \ \rightarrow \textbf{torch.Tensor}$

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.

1 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
                 ¥
     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.

1 Note

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

Variables

training $(b \circ o 1)$ – 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 $t \circ ()$, etc.

1 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.

```
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.

1 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) \rightarrow torch.Tensor
               Parameters
     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) \rightarrow torch.Tensor
               Parameters
     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.

1 Note

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

Variables

training $(b \circ o 1)$ – 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_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops

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

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

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

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

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

```
e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
```

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

$compute_sigma_qq(hq, rq)$

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

$\texttt{compute_sigma_pq} \ (\ ^*, hp, hq, rp, rq)$

```
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)

$\verb"apply_coefficients" (h0, hp, hq, r0, rp, rq)$

Multiplying a base vector with its scalar coefficient

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

Compute our CL multiplication

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

ei
$$^2 = +1$$
 for $i = < i = < p$ ej $^2 = -1$ for $p < j = < p+q$ ei ej = -eje1 for i

eq j

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

- (1) $sigma_0 = h_0 r_0 + sum_{i=1}^p (h_0 r_i) e_i sum_{j=p+1}^{p+q} (h_j r_j) e_j$
- (2) $sigma_p = sum_{i=1}^p (h_0 r_i + h_i r_0) e_i$
- (3) $sigma_q = sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j$
- (4) $sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k h_k r_i) e_i e_k$
- (5) $sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k h_k r_j) e_j e_k$
- (6) $sigma_{pq} = sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j h_j r_i) e_i e_j$

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

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

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

Parameter

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

returns

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

forward_k_vs_with_explicit(x: torch.Tensor)

k_vs_all_score (bpe_head_ent_emb, bpe_rel_ent_emb, E)

 $forward_k_vs_all(x: torch.Tensor) \rightarrow torch.FloatTensor$

Kvsall training

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

forward_k_vs_with_explicit and this funcitons are identical Parameter — x: torch.LongTensor with (n,2) shape :rtype: torch.FloatTensor with (n,|E|) 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, |E|) shape
      score(h, r, t)
      forward\_triples (x: torch.Tensor) \rightarrow torch.FloatTensor
            Parameter
            x: torch.LongTensor with (n,3) shape
                 rtype
                      torch.FloatTensor with (n) shape
class dicee.models.KeciBase(args)
      Bases: Keci
      Without learning dimension scaling
class dicee.models.CMult(args)
      Bases: dicee.models.base model.BaseKGE
      Cl(0,0) \Rightarrow 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} 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 + a_1 e_1 + a_2 e_2 + a_{12} e_1 e_2 A multivector mathbf\{b\} = b_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 e_2 A multivector mathbf\{b\} = b_0 e_1 + a_1 e_2 e_2 + a_2 e_3 e_4 A
            b_1 e_1 + b_2 e_2 + b_{12} e_1 e_2
            mathbf{a} imes mathbf{b} = a_0b_0 + a_0b_1 e_1 + a_0b_2 e_2 + a_0 b_1 e_1 e_2
                    • a_1 b_0 e_1 + a_1b_1 e_1_e1 ...
      Cl_{(0,2)} => Quaternions
      clifford_mul(x: torch.FloatTensor, y: torch.FloatTensor, p: int, q: int) \rightarrow tuple
                 Clifford multiplication Cl_{p,q} (mathbb{R})
                 ei ^2 = +1 for i = < i = < p ej ^2 = -1 for p < j = < p+q ei ej = -eje1 for i
            eq j
```

```
x: torch.FloatTensor with (n,d) shape
y: torch.FloatTensor with (n,d) shape
p: a non-negative integer p>= 0 q: a non-negative integer q>= 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

 $\textbf{forward_k_vs_all} \ (\textit{x: torch.Tensor}) \ \rightarrow \text{torch.FloatTensor}$

Compute batch KvsAll triple scores

Parameter

x: torch.LongTensor with shape n by 3

rtvpe

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 $t \circ ()$, etc.

1 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) —> 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) 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) s4 = \sum_{i=p+q+1}^{p+q+r} (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) s4 = \sum_{i=p+q+1}^{p+q+r} (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) s4 = \sum_{i=p+q+1}^{p+q+r} (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) s4 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (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) s5 = \sum_{i=p+q+1}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r} (h_0 r_i t_i + h_i r_0 t_i) s5 = \sum_{i=p+q+r}^{p+q+r}$$

and return:

$$sigma_0t = \sigma_0 \cdot t_0 = s0 + s1 - s2s3, s4ands5$$

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) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_i) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_j r_{j'} - h_{i'} r_j) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j=p+1}^{p+q} \sum_{j'=j+1}^{p+q} (h_j r_j - h_{i'} r_j) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j'=p+1}^{p+q} \sum_{j'=j+1}^{p+q} (h_j r_j - h_{j'} r_j) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j'=p+1}^{p+q} \sum_{j'=j+1}^{p+q} (h_j r_j - h_{j'} r_j) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j'=p+1}^{p+q} \sum_{j'=j+1}^{p+q} (h_j r_j - h_{j'} r_j) (models the interactions between e_i and e_i' for 1 <= i, i' <= p) \sigma_q q = \sum_{j'=p+1}^{p+q} \sum_{j'=j+1}^{p+q} (h_j r_j - h_{j'} r_j) (models the interactions between e_i and e_i' for 1 <= i, i' <= i$$

For different base vector interactions, we have

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

forward_k_vs_all (x: torch.Tensor) \rightarrow 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 funcitons are identical Parameter — x: torch.LongTensor with (n,) shape :rtype: torch.FloatTensor with (n, |E|) shape

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

Multiplying a base vector with its scalar coefficient

 $\verb|construct_cl_multivector||(x: torch.FloatTensor, re: int, p: int, q: int, r: int)|$

→ 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:

$$\label{eq:sigma_{p,p}^* = \sum_{i=1}^{p-1}\sum_{i'=i+1}^{p} (x_{i'}_{i'}-x_{i'}y_{i})} \\$$

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):

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

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

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

$compute_sigma_qq(hq, rq)$

Compute

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

sigma_{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):

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

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

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

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

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)

 $\texttt{compute_sigma_qr} \ (*, hq, hk, rq, rk)$

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

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

for i 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.

1 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

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

byte pair encoded neural link predictors

Parameters

```
init_params_with_sanity_checking()
```

$$\label{eq:condition} \begin{split} \textbf{forward} \ (x: torch.LongTensor \mid Tuple[torch.LongTensor, torch.LongTensor],} \\ y_idx: torch.LongTensor = None) \end{split}$$

Parameters

- x
- y_idx
- ordered_bpe_entities

```
forward_triples (x: torch.LongTensor) \rightarrow torch.Tensor
               Parameters
     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
                   \mathbf{x} (B \times 2 \times T)
     \texttt{get\_embeddings}() \rightarrow 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: 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) \rightarrow 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:

$$\label{eq:hammon} \begin{split} &h = h.reshape(len(x), self.embedding_dim, self.last_dim) \ r = r.reshape(len(x), self.embedding_dim, self.last_dim) \end{split}$$
 $\ & t = t.reshape(len(x), self.embedding_dim, self.last_dim) \end{split}$

(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)

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.

1 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.

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
                  \mathbf{x} (B \times 2 \times T)
     get_embeddings() → Tuple[numpy.ndarray, numpy.ndarray]
class dicee.models.FMult(args)
     Bases: dicee.models.base_model.BaseKGE
     Learning Knowledge Neural Graphs
     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
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
```

x

```
Bases: dicee.models.base_model.BaseKGE
     Learning Knowledge Neural Graphs
     build_func(Vec)
     build_chain_funcs (list_Vec)
     compute func (W, b, x) \rightarrow \text{torch.FloatTensor}
     function (list_W, list_b)
     trapezoid (list_W, list_b)
     forward_triples (idx\_triple: torch.Tensor) \rightarrow torch.Tensor
               Parameters
                   x
class dicee.models.LFMult1(args)
     Bases: dicee.models.base_model.BaseKGE
     Embedding with trigonometric functions. We represent all entities and relations in the complex number space as:
     f(x) = sum_{k=0}^{k=0}^{k=d-1}wk e^{kix}, and use the three differents scoring function as in the paper to evaluate
     the score
     forward_triples (idx_triple)
               Parameters
     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 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 = sigma(wh^T x + bh), r = sigma(wr^T x + br),
          t = sigma(wt^T x + bt)
     linear(x, w, b)
     scalar_batch_NN(a, b, c)
          element wise multiplication between a,b and c: Inputs: a, b, c ====> torch.tensor of size batch_size x m x
          d Output: a tensor of size batch_size x d
```

class dicee.models.FMult2(args)

tri_score (coeff_h, coeff_r, coeff_t)

this part implement the trilinear scoring techniques:

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

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

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

- 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 $[0][0] + \text{coeff}[0][1]x + ... + \text{coeff}[0][d]x^d$,

$$coeff[1][0] + coeff[1][1]x + ... + coeff[1][d]x^d$$

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.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) \to torch.tensor KvsAll scoring function

```
Input
            x: torch.LongTensor with (n, ) shape
            Output
            torch.FloatTensor with (n) shape
      \textbf{forward\_triples} \ (\textit{idx\_triple: torch.tensor}) \ \rightarrow \textbf{torch.tensor}) \ \rightarrow \textbf{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) \rightarrow torch.tensor
            Transpose function
            Input: Tensor with shape (nxm) Output: Tensor with shape (mxn)
dicee.read_preprocess_save_load_kg
Submodules
```

Classes

Preprocess the data in memory

dicee.read_preprocess_save_load_kg.preprocess

Module Contents

```
class dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG(kg)
     Preprocess the data in memory
     \mathtt{start}() \to \mathrm{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() \rightarrow None
     preprocess\_with\_pandas() \rightarrow 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
     {\tt preprocess\_with\_polars}\,(\,)\,\to None
     \verb"sequential_vocabulary_construction"\ () \ \to 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
```

ReadFromDisk

Read the data from disk into memory

Module Contents

```
class dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk(kg)
    Read the data from disk into memory
    start() → None
        Read a knowledge graph from disk into memory
        Data will be available at the train_set, test_set, valid_set attributes.

Parameter

None
        rtype
            None
        add_noisy_triples_into_training()

dicee.read_preprocess_save_load_kg.save_load_disk
```

Classes

LoadSaveToDisk

Module Contents

dicee.read_preprocess_save_load_kg.util

Functions

```
apply_reciprical_or_noise(add_reciprical,
 eval_model)
 timeit(func)
read\_with\_polars(\rightarrow polars.DataFrame)
                                                      Load and Preprocess via Polars
 read_with_pandas(data_path[, read_only_few, ...])
read_from_disk(data_path[, read_only_few, ...])
 read_from_triple_store([endpoint])
                                                      Read triples from triple store into pandas dataframe
 get_er_vocab(data[, file_path])
 get_re_vocab(data[, file_path])
 get_ee_vocab(data[, file_path])
 create_constraints(triples[, file_path])
                                                      Deserialize data
load\_with\_pandas(\rightarrow None)
 save_numpy_ndarray(*, data, file_path)
load_numpy_ndarray(*, file_path)
 save_pickle(*, data[, file_path])
load_pickle(*[, file_path])
 create_recipriocal_triples(x)
                                                      Add inverse triples into dask dataframe
 index triples with pandas(\rightarrow
                                               pan-
 das.core.frame.DataFrame)
 dataset\_sanity\_checking(\rightarrow None)
```

Module Contents

Read triples from triple store into pandas dataframe

- (1) Extract domains and ranges of relations
- (2) Store a mapping from relations to entities that are outside of the domain and range. Crete constrainted entities based on the range of relations :param triples: :return: Tuple[dict, dict]

```
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)
```

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

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

- 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) \rightarrow None
```

Parameters

- train_set
- num_entities
- num_relations

Returns

Classes

PreprocessKG	Preprocess the data in memory
LoadSaveToDisk	
ReadFromDisk	Read the data from disk into memory

Package Contents

```
class dicee.read_preprocess_save_load_kg.PreprocessKG(kg)
     Preprocess the data in memory
     \mathtt{start}() \to 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() \rightarrow None
     {\tt preprocess\_with\_pandas}\,(\,)\,\to 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
               rtvpe
                  None
     \textbf{preprocess\_with\_polars}\,(\,)\,\to None
     \verb"sequential_vocabulary_construction"\ () \ \to 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
     \mathtt{start}() \to \mathsf{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

Functions

```
get_default_arguments([description])
                                                    Extends pytorch_lightning Trainer's arguments with ours
main()
```

Module Contents

dicee.scripts.serve

Attributes

```
app
neural_searcher
```

Classes

NeuralSearcher

Functions

```
get_default_arguments()
root()
search_embeddings(q)
retrieve_embeddings(q)
main()
```

Module Contents

```
dicee.scripts.serve.app
dicee.scripts.serve.neural_searcher = None
dicee.scripts.serve.get_default_arguments()
async dicee.scripts.serve.root()
async dicee.scripts.serve.search_embeddings(q: str)
```

```
async dicee.scripts.serve.retrieve_embeddings(q: str)
class dicee.scripts.serve.NeuralSearcher(args)
    get (entity: str)
    search (entity: str)
dicee.scripts.serve.main()
```

dicee.trainer

Submodules

dicee.trainer.dice_trainer

Classes

DICE_Trainer

DICE_Trainer implement

Functions

```
initialize_trainer(args, callbacks)

get_callbacks(args)
```

Module Contents

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) \rightarrow lightning.Trainer
```

Initialize Trainer from input arguments

```
initialize_or_load_model()
```

 $initialize_dataloader$ (dataset: torch.utils.data.Dataset) \rightarrow torch.utils.data.DataLoader

```
initialize_dataset (dataset: dicee.knowledge_graph.KG, form_of_labelling)
```

 \rightarrow torch.utils.data.Dataset

 $\textbf{start} \ (\textit{knowledge_graph: dicee.knowledge_graph.KG}) \ \rightarrow \textbf{Tuple}[\textit{dicee.models.base_model.BaseKGE}, \textbf{str}]$

Train selected model via the selected training strategy

 $k_fold_cross_validation(dataset) \rightarrow Tuple[dicee.models.base_model.BaseKGE, str]$

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

TorchTrainer	TorchTrainer for using single GPU or multi CPUs on a
	single node

Module Contents

```
class dicee.trainer.torch_trainer.TorchTrainer(args, callbacks)
      Bases: dicee.abstracts.AbstractTrainer
           TorchTrainer for using single GPU or multi CPUs on a single node
           Arguments
      callbacks: list of Abstract callback instances
      fit (*args, train_dataloaders, **kwargs) \rightarrow None
                Training starts
                Arguments
           kwargs:Tuple
                empty dictionary
                Return type
                    batch loss (float)
      forward_backward_update (x_batch: torch. Tensor, y_batch: torch. Tensor) \rightarrow torch. Tensor
                Compute forward, loss, backward, and parameter update
                Arguments
                Return type
                    batch loss (float)
      \textbf{extract\_input\_outputs\_set\_device} \ (\textit{batch: list}) \ \rightarrow \textbf{Tuple}
                Construct inputs and outputs from a batch of inputs with outputs From a batch of inputs and put
                Arguments
                Return type
                    (tuple) mini-batch on select device
```

dicee.trainer.torch_trainer_ddp

Classes

TorchDDPTrainer	A Trainer based on torch.nn.parallel.DistributedDataParallel
NodeTrainer	
DDPTrainer	

Functions

```
print_peak_memory(prefix, device)
```

Module Contents

```
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 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
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()
```

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) \rightarrow lightning.Trainer
```

Initialize Trainer from input arguments

```
initialize_or_load_model()
```

 $initialize_dataloader$ (dataset: torch.utils.data.Dataset) \rightarrow torch.utils.data.DataLoader

 $\verb|start| (knowledge_graph: dicee.knowledge_graph.KG)| \rightarrow \verb|Tuple[dicee.models.base_model.BaseKGE, str]|$

Train selected model via the selected training strategy

 $k_fold_cross_validation(dataset) \rightarrow Tuple[dicee.models.base_model.BaseKGE, str]$

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

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

Module Contents

class dicee.abstracts.AbstractTrainer(args, callbacks)

A function to call callbacks before the training starts.

Abstract class for Trainer class for knowledge graph embedding models

Parameter

```
args
    [str] ?
callbacks: list
    ?
on_fit_start(*args, **kwargs)
```

```
Parameter
```

```
args
     kwargs
         rtype
             None
on_fit_end(*args, **kwargs)
     A function to call callbacks at the ned 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) \rightarrow 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
     \texttt{get\_eval\_report}() \rightarrow dict
     get_bpe_token_representation (str_entity_or_relation: List[str] | str)
                   \rightarrow 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]]) \rightarrow Tuple[List, List, List]
               Parameters
                   triples
     get\_domain\_of\_relation(rel: str) \rightarrow List[str]
     get_range_of_relation(rel: str) \rightarrow List[str]
     \verb"set_model_train_mode"() \to None
```

Setting the model into training mode

Parameter

```
\verb"set_model_eval_mode" () \to None
     Setting the model into eval mode
     Parameter
property name
sample_entity(n:int) \rightarrow List[str]
sample\_relation(n: int) \rightarrow List[str]
is_seen (entity: str = None, relation: str = None) \rightarrow bool
save() \rightarrow None
get_entity_index (x: str)
get_relation_index (x: str)
index_triple (head_entity: List[str], relation: List[str], tail_entity: List[str])
              → 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
```

```
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],
          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

```
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

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.callbacks

Classes

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

Functions

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

Module Contents

```
{\tt class \ dicee.callbacks.AccumulateEpochLossCallback}\ (\textit{path: str})
```

 $Bases: \ \textit{dicee.abstracts.} Abstract \textit{Callback}$

Abstract class for Callback class for knowledge graph embedding models

```
Parameter
```

```
on_fit_end(trainer, model) \rightarrow 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.
```

```
trainer:
           model:
               rtype
                   None
     on_train_epoch_end(*args, **kwargs)
           Call at the end of each epoch during training.
           Parameter
           trainer:
           model:
               rtype
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
     \verb"on_fit_start" (\textit{trainer}, \textit{pl}\_\textit{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

```
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
\verb|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) \rightarrow float
     get_aswa_state_dict(model)
     {\tt 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
```

Abstract class for Callback class for knowledge graph embedding models

Bases: dicee.abstracts.AbstractCallback

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 mush :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

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

Namespace

Simple object for storing attributes.

Module Contents

```
class dicee.config.Namespace(**kwargs)
     Bases: argparse.Namespace
     Simple object for storing attributes.
     Implements equality by attribute names and values, and provides a simple string representation.
     dataset_dir: str = None
          The path of a folder containing train.txt, and/or valid.txt and/or test.txt
     save_embeddings_as_csv: bool = False
          Embeddings of entities and relations are stored into CSV files to facilitate easy usage.
     storage_path: str = 'Experiments'
          A directory named with time of execution under -storage_path that contains related data about embeddings.
     path_to_store_single_run: str = None
          A single directory created that contains related data about embeddings.
     path_single_kg = None
          Path of a file corresponding to the input knowledge graph
     sparql_endpoint = None
          An endpoint of a triple store.
     model: str = 'Keci'
          KGE model
     optim: str = 'Adam'
          Optimizer
     embedding_dim: int = 64
          Size of continuous vector representation of an entity/relation
     num_epochs: int = 150
          Number of pass over the training data
     batch size: int = 1024
          Mini-batch size if it is None, an automatic batch finder technique applied
     lr: float = 0.1
          Learning rate
     add_noise_rate: float = None
          The ratio of added random triples into training dataset
     gpus = None
          Number GPUs to be used during training
     callbacks
          10}}
              Type
                  Callbacks, e.g., {"PPE"
              Type
                  { "last_percent_to_consider"
```

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

BPE_NegativeSamplingDataset	An abstract class representing a Dataset.
MultiLabelDataset	An abstract class representing a Dataset.
MultiClassClassificationDataset	Dataset for the 1vsALL training strategy
OnevsAllDataset	Dataset for the 1vsALL training strategy
KvsAll	Creates a dataset for KvsAll training by inheriting from
	torch.utils.data.Dataset.
AllvsAll	Creates a dataset for AllvsAll training by inheriting from
	torch.utils.data.Dataset.
KvsSampleDataset	KvsSample a Dataset:
NegSampleDataset	An abstract class representing a Dataset.
TriplePredictionDataset	Triple Dataset
CVDataModule	Create a Dataset for cross validation

Functions

```
\begin{tabular}{ll} reload\_dataset (path, form\_of\_labelling, ...) & Reload the files from disk to construct the Pytorch dataset \\ construct\_dataset ($\rightarrow$ torch.utils.data.Dataset) \\ \end{tabular}
```

Module Contents

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
```

Bases: torch.utils.data.Dataset

An abstract class representing a 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.

1 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
```

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.

1 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_idxs)
     Bases: torch.utils.data.Dataset
     Dataset for the 1vsALL training strategy
          Parameters
               • train_set_idx - Indexed triples for the training.
               • entity_idxs - mapping.
               • relation_idxs - mapping.
               • form - ?
               • num_workers - int for https://pytorch.org/docs/stable/data.html#torch.utils.data.
                 DataLoader
          Return type
              torch.utils.data.Dataset
      _len__()
      getitem_{(idx)}
```

class dicee.dataset_classes.KvsAll(train_set_idx: numpy.ndarray, entity_idxs, relation_idxs, 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:= $\{(x,y) \mid i \mid i \land N, \text{ where } x: (h,r) \text{ is an unique} \}$ 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 $[0,1]^{\{|\mathbf{E}|\}}$ is a binary label.

orall y i = 1 s.t. (h r E i) in KG



train_set_idx

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

entity idxs

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

relation idxs

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

self: torch.utils.data.Dataset

```
>>> a = KvsAll()
>>> a
```

```
__len__()
\_getitem\_(idx)
```

class dicee.dataset_classes.AllvsAll (train_set_idx: numpy.ndarray, entity_idxs, relation_idxs, label smoothing rate=0.0)

Bases: torch.utils.data.Dataset

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

Let D denote a dataset for AllysAll training and be defined as $D := \{(x,y)_i\}_i ^n N$, 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



1 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 idxs

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

```
relation idxs
               [dictonary] string representation of a relation to its integer id
           self: torch.utils.data.Dataset
           >>> a = AllvsAll()
           ? 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, \text{ 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
                   lnew_yl << IEI new_y contains all 1's if sum(y)< neg_sample ratio new_y contains</pre>
           train_set_idx
               Indexed triples for the training.
           entity_idxs
               mapping.
           relation_idxs
               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 <code>DataLoader</code>. Subclasses could also optionally implement <code>__getitems__</code>(), for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.



Parameters

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, \text{ where }
                    . x:(h,r,t) in KG is a unique h in E and a relation r in R and . collact_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 idxs
                mapping.
           relation_idxs
                mapping.
           form
           store
           label_smoothing_rate
           collate_fn: batch:List[torch.IntTensor] Returns ——- torch.utils.data.Dataset
      __len__()
      \underline{\phantom{a}}getitem\underline{\phantom{a}} (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
```

- 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.

dataloader will not be reloaded unless you return you set :paramref: ~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()

1 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, ...).

1 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
```

(continues on next page)

(continued from previous page)

```
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.

A Warning

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

Example:

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

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

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

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

Example:

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

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

This is called before requesting the dataloaders:

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

dicee.eval_static_funcs

Functions

```
evaluate_link_prediction_performance(→
Dict)
evaluate_link_prediction_performance_w

evaluate_link_prediction_performance_w

evaluate_link_prediction_performance_w
...)
evaluate_lp_bpe_k_vs_all(model, triples[, er_vocab, ...])
```

Module Contents

dicee.eval_static_funcs.

• re_vocab

```
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])
```

- model
- triples
- within_entities
- er_vocab
- re_vocab

dicee.evaluator

Classes

Evaluator

Evaluator class to evaluate KGE models in various downstream tasks

Module Contents

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

Evaluator class to evaluate KGE models in various downstream tasks

Arguments

 $vocab_preparation(dataset) \rightarrow None$

A function to wait future objects for the attributes of executor

Return type

None

 $\begin{tabular}{ll} \textbf{eval_with_byte} (*, raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model,\\ form_of_labelling) &\rightarrow None \end{tabular}$

Evaluate model after reciprocal triples are added

 $\begin{tabular}{ll} \textbf{eval_with_bpe_vs_all} (*, raw_train_set, raw_valid_set=None, raw_test_set=None, trained_model, \\ form_of_labelling) \rightarrow \textbf{None} \\ \end{tabular}$

Evaluate model after reciprocal triples are added

```
\begin{tabular}{ll} \textbf{eval\_with\_vs\_all} (*, train\_set, valid\_set=None, test\_set=None, trained\_model, form\_of\_labelling) \\ &\rightarrow \textbf{None} \end{tabular}
```

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)

 $\textbf{evaluate_lp_bpe_k_vs_all} \ (\textit{model}, \textit{triples: List[List[str]]}, \textit{info=None}, \textit{form_of_labelling=None})$

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

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

Module Contents

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

A class for Training, Retraining and Evaluation a model.

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

read_or_load_kg()

 ${\tt read_preprocess_index_serialize_data}\:(\:)\:\to None$

Read & Preprocess & Index & Serialize Input Data

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

rtype

None

$\textbf{load_indexed_data} \, (\,) \, \to None$

Load the indexed data from disk into memory

Parameter

rtype

None

${\tt save_trained_model}\:()\:\to None$

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 $(form_of_labelling: str) \rightarrow dict$

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

$\textbf{write_report}\:(\:)\:\to None$

Report training related information in a report.json file

$\mathtt{start}() \rightarrow \mathrm{dict}$

Start training

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

rtype

A dict containing information about the training and/or evaluation

class dicee.executer.ContinuousExecute(args)

Bases: Execute

A subclass of Execute Class for retraining

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

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

```
continual_start() → dict
```

Start Continual Training

- (1) Initialize training.
- (2) Start continual training.
- (3) Save trained model.

Parameter

rtype

A dict containing information about the training and/or evaluation

dicee.knowledge graph

Classes

KG

Knowledge Graph

Module Contents

```
property entities_str: List
```

```
property relations_str: List
```

func_triple_to_bpe_representation (triple: List[str])

dicee.knowledge_graph_embeddings

Classes

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

Module Contents

```
class dicee.knowledge_graph_embeddings.KGE (path=None, url=None,
            construct_ensemble=False, model_name=None, apply_semantic_constraint=False)
     Bases: dicee.abstracts.BaseInteractiveKGE
     Knowledge Graph Embedding Class for interactive usage of pre-trained models
      __str__()
           Return str(self).
     to (device: str) \rightarrow None
     get_transductive_entity_embeddings (indices: torch.LongTensor | List[str],
                   as_pytorch=False, as_numpy=False, as_list=True)
                   → torch.FloatTensor | numpy.ndarray | List[float]
     create_vector_database (collection_name: str, distance: str, location: str = 'localhost',
                  port: int = 6333)
     generate (h=", r=")
     eval_lp_performance (dataset=List[Tuple[str, str, str]], filtered=True)
     predict_missing_head_entity (relation: List[str] | str, tail_entity: List[str] | str, within=None)
                   \rightarrow Tuple
           Given a relation and a tail entity, return top k ranked head entity.
           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
```

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

```
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

```
\verb|predict_missing_tail_entity| (\textit{head\_entity: List[str]} \mid \textit{str}, \textit{relation: List[str]} \mid \textit{str}, \\
```

within: List[str] = None \rightarrow torch.FloatTensor

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

 $argmax_{e} in E$ 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

```
 \begin{aligned} \textbf{predict} \ (*, h: List[str] \mid str = None, r: List[str] \mid str = None, t: List[str] \mid str = None, within=None, \\ logits=True) \ \to \text{torch.FloatTensor} \end{aligned}
```

Parameters

- · logits
- h

·r

• t

• within

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

```
\label{eq:core}  \textbf{triple\_score} \ (h: List[str] \mid str = None, \, r: \, List[str] \mid str = None, \, t: \, List[str] \mid str = None, \, logits = False) \\ \rightarrow \text{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') \rightarrow torch. Tensor
tensor_t_norm (subquery\_scores: torch.FloatTensor, tnorm: str = 'min') \rightarrow torch.FloatTensor
     Compute T-norm over [0,1] ^{n imes 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') \rightarrow torch.Tensor
negnorm (tens 1: torch.Tensor, lambda: float, neg norm: str = 'standard') \rightarrow 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)
              \rightarrow 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) \rightarrow 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

Module Contents

```
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]]]) \rightarrow None
     Save Queries into Disk
static load_queries_and_answers (path: str)
             \rightarrow List[Tuple[str, Tuple[collections.defaultdict]]]
     Load Queries from Disk to Memory
```

dicee.sanity_checkers

Functions

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

Functions

<pre>create_recipriocal_triples(x) get_er_vocab(data[, file_path])</pre>	Add inverse triples into dask dataframe
<pre>get_re_vocab(data[, file_path])</pre>	
<pre>get_ee_vocab(data[, file_path])</pre>	
timeit(func)	
<pre>save_pickle(*[, data, file_path])</pre>	
load_pickle([file_path])	
<pre>select_model(args[, is_continual_training, stor- age_path])</pre>	
<pre>load_mode1(→ Tuple[object, Tuple[dict, dict]])</pre>	Load weights and initialize pytorch module from namespace arguments
load_model_ensemble()	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
save_numpy_ndarray(*, data, file_path)	
numpy_data_type_changer(→ numpy.ndarray)	Detect most efficient data type for a given triples
$save_checkpoint_model(\rightarrow None)$ $store(\rightarrow None)$	Store Pytorch model into disk Store trained_model model and save embeddings into csv
	file.
add_noisy_triples(→ pandas.DataFrame)	Add randomly constructed triples
read_or_load_kg(args, cls)	
<pre>intialize_model(→ Tuple[object, str])</pre>	
$load_json(\rightarrow dict)$	
$save_embeddings(\rightarrow None)$	Save it as CSV if memory allows.
random_prediction(pre_trained_kge)	
<pre>deploy_triple_prediction(pre_trained_kge, str_subject,)</pre>	
<pre>deploy_tail_entity_prediction(pre_trained)</pre>	
<pre>deploy_head_entity_prediction(pre_trained)</pre>	
<pre>deploy_relation_prediction(pre_trained_kge,)</pre>	
<pre>vocab_to_parquet(vocab_to_idx, name,)</pre>	
<pre>create_experiment_folder([folder_name])</pre>	

continues on next page

Table 2 - continued from previous page

```
      continual_training_setup_executor(→
      storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data

      exponential_function(→ torch.FloatTensor)
      load_numpy(→ numpy.ndarray)

      evaluate(entity_to_idx, scores, easy_answers, hard_answers)
      # @TODO: CD: Renamed this function

      download_file(url[, destination_folder])
      download_files_from_url(→ None)

      download_pretrained_model(→ str)
```

Module Contents

```
dicee.static_funcs.create_recipriocal_triples(x)
     Add inverse triples into dask dataframe :param x: :return:
dicee.static_funcs.get_er_vocab(data, file_path: str = None)
dicee.static_funcs.get_re_vocab(data, file_path: str = None)
dicee.static_funcs.get_ee_vocab(data, file_path: str = None)
dicee.static funcs.timeit(func)
dicee.static_funcs.save_pickle(*, data: object = None, file_path=str)
dicee.static_funcs.load_pickle(file_path=str)
dicee.static_funcs.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) \rightarrow 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)
            \rightarrow 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) \rightarrow 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) \rightarrow 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)
            \rightarrow 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) \rightarrow dict
dicee.static funcs.save embeddings (embeddings: numpy.ndarray, indexes, path: str) \rightarrow 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) \rightarrow 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) \rightarrow 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='.') \rightarrow 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) $\rightarrow str$

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

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.efficient_zero_grad(model)

dicee.static preprocess funcs

Attributes

enable_log

Functions

```
timeit(func)
preprocesses\_input\_args(args) \qquad Sanity Checking in input arguments
create\_constraints(\rightarrow Tuple[dict, dict, dict, dict, dict])
get\_er\_vocab(data)
get\_re\_vocab(data)
get\_ee\_vocab(data)
mapping\_from\_first\_two\_cols\_to\_third(trains)
```

Module Contents

- (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:

13.3 Attributes

__version__

13.4 Classes

CM-, 1+	C1 (0,0) => Real Numbers
CMult	- ` / /
Pyke	A Physical Embedding Model for Knowledge Graphs
DistMult	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
KeciBase	Without learning dimension scaling
Keci	Base class for all neural network modules.
TransE	Translating Embeddings for Modeling
DeCaL	Base class for all neural network modules.
DualE	Dual Quaternion Knowledge Graph Embeddings (https://ojs.aaai.org/index.php/AAAI/article/download/16850/16657)
ComplEx	Base class for all neural network modules.
AConEx	Additive Convolutional ComplEx Knowledge Graph Embeddings
AConv0	Additive Convolutional Octonion Knowledge Graph Embeddings
AConvQ	Additive Convolutional Quaternion Knowledge Graph Embeddings
ConvQ	Convolutional Quaternion Knowledge Graph Embeddings
ConvO	Base class for all neural network modules.
ConEx	Convolutional ComplEx Knowledge Graph Embeddings
OMult	Base class for all neural network modules.
OMult	Base class for all neural network modules.
Shallom	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
LFMult	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
PykeenKGE	A class for using knowledge graph embedding models implemented in Pykeen
BytE	Base class for all neural network modules.
BaseKGE	Base class for all neural network modules.
DICE_Trainer	DICE_Trainer implement
KGE	Knowledge Graph Embedding Class for interactive usage of pre-trained models
Execute	A class for Training, Retraining and Evaluation a model.
BPE_NegativeSamplingDataset	An abstract class representing a Dataset.
MultiLabelDataset	An abstract class representing a Dataset.
MultiClassClassificationDataset	Dataset for the 1vsALL training strategy
OnevsAllDataset	Dataset for the 1vsALL training strategy
KvsAll	Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.
AllvsAll	Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.
KvsSampleDataset	KvsSample a Dataset:
NegSampleDataset	An abstract class representing a Dataset.
TriplePredictionDataset	Triple Dataset

continues on next page

QueryGenerator

13.5 Functions

```
Add inverse triples into dask dataframe
create_recipriocal_triples(x)
get_er_vocab(data[, file_path])
get_re_vocab(data[, file_path])
get_ee_vocab(data[, file_path])
 timeit(func)
save_pickle(*[, data, file_path])
load_pickle([file_path])
select_model(args[, is_continual_training,
age_path])
load_model(→ Tuple[object, Tuple[dict, dict]])
                                                       Load weights and initialize pytorch module from names-
                                                       pace arguments
load_model_ensemble(...)
                                                       Construct Ensemble Of weights and initialize pytorch
                                                       module from namespace arguments
save_numpy_ndarray(*, data, file_path)
numpy_data_type_changer(\rightarrow numpy.ndarray)
                                                       Detect most efficient data type for a given triples
save\_checkpoint\_model(\rightarrow None)
                                                       Store Pytorch model into disk
store(\rightarrow None)
                                                       Store trained_model model and save embeddings into csv
                                                       file.
                                                       Add randomly constructed triples
add\_noisy\_triples(\rightarrow pandas.DataFrame)
 read_or_load_kg(args, cls)
intialize\_model(\rightarrow Tuple[object, str])
load_{json}(\rightarrow dict)
save\_embeddings(\rightarrow None)
                                                       Save it as CSV if memory allows.
 random_prediction(pre_trained_kge)
deploy_triple_prediction(pre_trained_kge,
str_subject, ...)
deploy_tail_entity_prediction(pre_trained_)
deploy_head_entity_prediction(pre_trained_)
...)
deploy_relation_prediction(pre_trained_kge,
```

continues on next page

Table 4 - continued from previous page

```
vocab_to_parquet(vocab_to_idx, name, ...)
create experiment folder([folder name])
continual_training_setup_executor(→
                                                     storage_path:str A path leading to a parent directory,
None)
                                                      where a subdirectory containing KGE related data
exponential\_function(\rightarrow torch.FloatTensor)
load_numpy(\rightarrow numpy.ndarray)
                                                     # @TODO: CD: Renamed this function
evaluate(entity_to_idx,
                            scores,
                                      easy_answers,
hard_answers)
download_file(url[, destination_folder])
download files from url(\rightarrow None)
download\_pretrained\_model(\rightarrow str)
mapping_from_first_two_cols_to_third(tra
timeit(func)
load_pickle([file_path])
                                                      Reload the files from disk to construct the Pytorch dataset
reload_dataset(path, form_of_labelling, ...)
 construct_dataset(→ torch.utils.data.Dataset)
```

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} 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} imes mathbf{b} = a_0 b_0 + a_0 b_1 e_1 + a_0 b_2 e_2 + a_0 b_1 2 e_1 e_2

• 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) \rightarrow tuple
```

```
Clifford multiplication Cl_{p,q} (mathbb\{R\})
               ei ^2 = +1 for i =< i =< p ej ^2 = -1 for p < j =< p+q ei ej = -eje1 for i
          eq j
               x: torch.FloatTensor with (n,d) shape
               y: torch.FloatTensor with (n,d) shape
               p: a non-negative integer p \ge 0 q: a non-negative integer q \ge 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.Pyke(args)
     Bases: dicee.models.base_model.BaseKGE
     A Physical Embedding Model for Knowledge Graphs
     forward_triples (x: torch.LongTensor)
               Parameters
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)
```

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 $(b \circ o 1)$ – 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_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops

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

```
for k in range(i + 1, p):
    results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])
sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))
```

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

```
e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
```

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

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

Compute sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k - h_k r_j) e_j e_k sigma_{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 i in range(q - 1):
```

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

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

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

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

```
e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
```

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

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

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

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

for j in range(q):

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

print(sigma_pq.shape)

apply_coefficients (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

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

ei
$$^2 = +1$$
 for i =< i =< p ej $^2 = -1$ for p < j =< p+q ei ej = -eje1 for i

eq j

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

- (1) $sigma_0 = h_0 r_0 + sum_{i=1}^p (h_0 r_i) e_i sum_{j=p+1}^{p+q} (h_j r_j) e_j$
- (2) sigma $p = sum \{i=1\}^p (h \ 0 \ r \ i + h \ i \ r \ 0) e i$
- (3) $sigma_q = sum_{j=p+1}^{p+q} (h_0 r_j + h_j r_0) e_j$
- (4) $sigma_{pp} = sum_{i=1}^{p-1} sum_{k=i+1}^p (h_i r_k h_k r_i) e_i e_k$
- (5) $sigma_{qq} = sum_{j=1}^{p+q-1} sum_{k=j+1}^{p+q} (h_j r_k h_k r_j) e_j e_k$
- (6) $sigma_{pq} = sum_{i=1}^{p} sum_{j=p+1}^{p+q} (h_i r_j h_j r_i) e_i e_j$

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

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

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

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)

 $forward_k_vs_all(x: torch.Tensor) \rightarrow torch.FloatTensor$

Kvsall training

- (1) Retrieve real-valued embedding vectors for heads and relations $mathbb{R}^d$.
- (2) Construct head entity and relation embeddings according to $Cl_{p,q}(\mathsf{mathbb}\{R\}^d)$.
- (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, |E|) shape

 $\label{lem:condition} \textbf{forward_k_vs_sample} \ (x: torch.LongTensor, target_entity_idx: torch.LongTensor)$

 \rightarrow 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 $Cl_{p,q}(\mathbf{mathbb}_{R}^{d})$.
- (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, |E|) shape

 $\mathtt{score}\,(h,r,t)$

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

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)
```

 $forward_k_vs_all(x: torch.Tensor) \rightarrow torch.FloatTensor$

```
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.

1 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

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) \longrightarrow 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,

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

and return:

$$sigma_0t = \sigma_0 \cdot t_0 = s0 + s1 - s2s3, s4ands5$$

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

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

For different base vector interactions, we have

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

 $forward_k_vs_all(x: torch.Tensor) \rightarrow torch.FloatTensor$

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 funcitons are identical Parameter — x: torch.LongTensor with (n,) shape :rtype: torch.FloatTensor with (n, |E|) shape

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

Multiplying a base vector with its scalar coefficient

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

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

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

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:

$$\label{eq:sigma_p} $$ \sum_{p,p}^* = \sum_{i=1}^{p-1}\sum_{i'=i+1}^{p} (x_iy_{i'}-x_{i'}y_i) $$$$

sigma_{pp} captures the interactions between along p bases For instance, let p e_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):

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

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

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

$compute_sigma_qq(hq, rq)$

Compute

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

sigma_ $\{q\}$ 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):

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

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

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_1h , e_2h , e_3h , e_4h , e_5h , e_6h , e_7h , e_8h , e_1t , e_2t , e_3t , e_4t , e_5t , e_6t , e_7t , e_8t , e_1t , e_2t , e_3t , e_4t , e_5t , e_6t , e_7t , e_8t

KvsAll scoring function

Input

x: torch.LongTensor with (n,) shape

Output

```
torch.FloatTensor with (n) shape
```

```
forward_triples (idx\_triple: torch.tensor) \rightarrow 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) \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.

1 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.

Parameters

- emb_h
- emb_r
- emb_E

 $forward_k_vs_all(x: torch.LongTensor) \rightarrow torch.FloatTensor$

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]) \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.AConvO(args: dict)

Bases: dicee.models.base_model.BaseKGE

Additive Convolutional Octonion Knowledge Graph Embeddings

residual_convolution (O_1, O_2)

```
forward_triples (x: torch. Tensor) \rightarrow 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 l)
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 l)
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__()
```

(continues on next page)

(continued from previous page)

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

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

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

1 Note

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

Variables

training $(b \circ o 1)$ – Boolean represents whether this module is in training or evaluation mode.

 $residual_convolution(O_1, O_2)$

forward_triples (x: torch.Tensor) \rightarrow 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]) \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.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.

1 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.

$\verb|static quaternion_normalizer|(x: torch.FloatTensor)| \rightarrow torch.FloatTensor|$

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

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

Parameters

 \mathbf{x} – The vector.

Returns

The normalized vector.

```
\verb+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.

1 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.

k_vs_all_score (bpe_head_ent_emb, bpe_rel_ent_emb, E)

forward_k_vs_all (x)

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

class dicee.Shallom(args)

Bases: dicee.models.base_model.BaseKGE

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

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

forward_k_vs_all $(x) \rightarrow \text{torch.FloatTensor}$

forward_triples $(x) \rightarrow \text{torch.FloatTensor}$

Parameters

x

Returns

class dicee.LFMult(args)

Bases: dicee.models.base_model.BaseKGE

Embedding with polynomial functions. We represent all entities and relations in the polynomial space as: $f(x) = sum_{i=0}^{d-1} a_k x^{i/d}$ 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. $h = sigma(wh^T x + bh)$, $r = sigma(wr^T x + br)$, $t = sigma(wt^T x + bt)$

linear(x, w, b)

 $scalar_batch_NN(a, b, c)$

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

tri_score (coeff_h, coeff_r, coeff_t)

this part implement the trilinear scoring techniques:

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

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

```
score(h,r,t) = \inf_{0}\{1\} \quad h(x)r(x)t(x) \quad dx = \sup_{i,j,k} = 0\}^{d-1} \quad 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 $[0][0] + \text{coeff}[0][1]x + ... + \text{coeff}[0][d]x^d$,

$$coeff[1][0] + coeff[1][1]x + ... + coeff[1][d]x^d$$

```
pop(coeff, x, degree)
```

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

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

$$coeff[1][0] + coeff[1][1]x + ... + coeff[1][d]x^d$$

class dicee.PykeenKGE (args: dict)

Bases: dicee.models.base_model.BaseKGE

A class for using knowledge graph embedding models implemented in Pykeen

Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Pykeen_HolE: Pykeen_HolE:

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, $r = self.get_head_relation_representation(x) # (2) Reshape (1). if <math>self.last_dim > 0$:
 - $h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim)$
- # (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) \rightarrow 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.

1 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

```
\mathbf{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()
```



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.

1 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()
```

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

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

 $\texttt{get_embeddings} \ () \ \to Tuple[numpy.ndarray, numpy.ndarray]$

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)

 $\rightarrow Tuple[\textit{dicee.models.base_model.BaseKGE}, Tuple[pandas.DataFrame, pandas.DataFrame]]$

Construct Ensemble Of weights and initialize pytorch module from namespace arguments

- (1) Detect models under given path
- (2) Accumulate parameters of detected models

```
(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) \rightarrow 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) \rightarrow 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) \rightarrow 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) \rightarrow Tuple[object, str]
dicee.load json(p: str) \rightarrow dict
dicee.save embeddings (embeddings: numpy.ndarray, indexes, path: str) \rightarrow 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) \rightarrow 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)
             \rightarrow torch.FloatTensor
dicee.load_numpy(path) \rightarrow 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
```

(3) Normalize parameters

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

```
dicee.download files_from_url(base_url: str, destination_folder='.') \rightarrow None
           Parameters
                                                   "https://files.dice-research.org/projects/DiceEmbeddings/
                 • base url
                                  (e.g.
                   KINSHIP-Keci-dim128-epoch256-KvsAll")
                 • destination_folder(e.g. "KINSHIP-Keci-dim128-epoch256-KvsAll")
dicee.download_pretrained_model(url: str) \rightarrow 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_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) \rightarrow Tuple[dicee.models.base\_model.BaseKGE, str]
           Train selected model via the selected training strategy
     k_{fold} cross_{validation}(dataset) \rightarrow 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
     __str__()
           Return str(self).
     to (device: str) \rightarrow None
     get_transductive_entity_embeddings (indices: torch.LongTensor | List[str],
                  as_pytorch=False, as_numpy=False, as_list=True)
                   → torch.FloatTensor | numpy.ndarray | List[float]
     create_vector_database (collection_name: str, distance: str, location: str = 'localhost',
                  port: int = 6333)
     generate (h=", r=")
     eval_lp_performance (dataset=List[Tuple[str, str, str]], filtered=True)
     predict missing head entity (relation: List[str] | str, tail_entity: List[str] | str, within=None)
                   \rightarrow Tuple
           Given a relation and a tail entity, return top k ranked head entity.
           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
```

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

```
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 \rightarrow torch.FloatTensor

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

 $argmax_{e} in E$ 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

Parameters

- · logits
- h
- r

• t

• within

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

 $\label{eq:core} \textbf{triple_score} \ (h: List[str] \mid str = None, \, r: \, List[str] \mid str = None, \, t: \, List[str] \mid str = None, \, logits = False) \\ \rightarrow \text{torch.FloatTensor}$

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

@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

→ List[Tuple[str, torch.Tensor]]

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

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) \mid f(e,r,x) > \text{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) \rightarrow 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()
```

$read_preprocess_index_serialize_data() \rightarrow None$

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

 $\textbf{load_indexed_data}\,(\,)\,\to None$

Load the indexed data from disk into memory

Parameter

```
rtype
```

None

```
{\tt save\_trained\_model}\:(\:)\:\to 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 ($form_of_labelling: str$) \rightarrow 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() \rightarrow None
```

Report training related information in a report. json file

```
\mathtt{start}() \rightarrow \mathrm{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

 $\verb|class dicee.BPE_NegativeSamplingDataset|| \textit{train_set: torch.LongTensor}|,$

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.

1 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]])
```

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.

1 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)

```
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__()
      \underline{\underline{\phantom{a}}}getitem\underline{\phantom{a}} (idx)
class dicee.OnevsAllDataset (train_set_idx: numpy.ndarray, entity_idxs)
     Bases: torch.utils.data.Dataset
     Dataset for the 1vsALL training strategy
           Parameters
                 • train_set_idx - Indexed triples for the training.
                 • entity_idxs - mapping.
                 • relation_idxs - mapping.
                 • form - ?
                 • num workers - int for https://pytorch.org/docs/stable/data.html#torch.utils.data.
                   DataLoader
           Return type
               torch.utils.data.Dataset
     __len__()
     \__getitem_{\_}(idx)
class dicee. KvsAll (train_set_idx: numpy.ndarray, entity_idxs, relation_idxs, form, store=None,
            label smoothing rate: float = 0.0)
     Bases: torch.utils.data.Dataset
```

class dicee.MultiClassClassificationDataset (subword_units: numpy.ndarray,

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 an unique tuple of an entity h in E and a relation r in R that has been seed in the input graph. y: denotes a multi-label vector in $[0,1]^{\{E\}}$ is a binary label.

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

1 Note

TODO

train_set_idx

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

entity_idxs

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

relation_idxs

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

self: torch.utils.data.Dataset

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

```
__len__()
qetitem (idx)
```

class dicee. AllvsAll (train set idx: numpy.ndarray, entity idxs, relation idxs, 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

1 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_idxs

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

relation idxs

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

self: torch.utils.data.Dataset

```
>>> a = AllvsAll()
? 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]^{\{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| << |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_idxs
               mapping.
           relation idxs
               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
```

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.

1 Note

An abstract class representing a Dataset.

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, \text{ where }
                   . x:(h,r, t) in KG is a unique h in E and a relation r in R and . collact 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_idxs
               mapping.
           relation_idxs
               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.

dataloader you return will not be reloaded unless :paramyou set 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()

1 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.11 = None
   def prepare_data(self):
        download_data()
        tokenize()
```

(continues on next page)

(continued from previous page)

```
# 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, ...).

1 Note

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

Parameters

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

Returns

A reference to the data on the new device.

Example:

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

See also

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

prepare_data(*args, **kwargs)

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

⚠ Warning

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

Example:

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

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

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

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

Example:

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

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

This is called before requesting the dataloaders:

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

```
class dicee. QueryGenerator (train_path, 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 \mid Tuple) \rightarrow List \mid 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) \rightarrow 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]]]) \rightarrow 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,94
dicee.analyse_experiments, 100
dicee.callbacks, 101
dicee.config, 107
dicee.dataset_classes, 110
dicee.eval_static_funcs, 119
dicee.evaluator.120
dicee.executer, 121
dicee.knowledge_graph, 123
dicee.knowledge_graph_embeddings, 124
dicee.models, 10
dicee.models.base_model, 10
dicee.models.clifford, 18
dicee.models.complex, 26
dicee.models.dualE, 28
dicee.models.function_space, 29
dicee.models.octonion, 31
dicee.models.pykeen models, 34
dicee.models.quaternion, 35
dicee.models.real, 37
dicee.models.static_funcs, 39
dicee.models.transformers, 39
dicee.query_generator, 128
dicee.read_preprocess_save_load_kg, 81
dicee.read_preprocess_save_load_kg.preprocess,
dicee.read_preprocess_save_load_kg.read_from_disk,
dicee.read_preprocess_save_load_kg.save_load_disk,
dicee.read_preprocess_save_load_kg.util,
dicee.sanity_checkers, 129
dicee.scripts, 87
dicee.scripts.index, 87
dicee.scripts.run,87
dicee.scripts.serve, 88
dicee.static_funcs, 130
dicee.static_funcs_training, 133
dicee.static_preprocess_funcs, 133
dicee.trainer,89
dicee.trainer.dice_trainer,89
dicee.trainer.torch_trainer,90
dicee.trainer.torch_trainer_ddp,91
```

Index

Non-alphabetical

```
__call__() (dicee.models.base_model.IdentityClass method), 18
__call__() (dicee.models.IdentityClass method), 53, 61, 65
__getitem__() (dicee.AllvsAll method), 169
__getitem__() (dicee.BPE_NegativeSamplingDataset method), 166
__getitem__() (dicee.dataset_classes.AllvsAll method), 114
__getitem__() (dicee.dataset_classes.BPE_NegativeSamplingDataset method), 111
__getitem__() (dicee.dataset_classes.KvsAll method), 113
__getitem__() (dicee.dataset_classes.KvsSampleDataset method), 114
__getitem__() (dicee.dataset_classes.MultiClassClassificationDataset method), 112
__getitem__() (dicee.dataset_classes.MultiLabelDataset method), 112
__getitem__() (dicee.dataset_classes.NegSampleDataset method), 115
  _getitem__() (dicee.dataset_classes.OnevsAllDataset method), 112
__getitem__() (dicee.dataset_classes.TriplePredictionDataset method), 115
__getitem__() (dicee.KvsAll method), 168
__getitem__() (dicee.KvsSampleDataset method), 169
__getitem__() (dicee.MultiClassClassificationDataset method), 167
  _getitem__() (dicee.MultiLabelDataset method), 166
__getitem__() (dicee.NegSampleDataset method), 170
__getitem__() (dicee.OnevsAllDataset method), 167
__getitem__() (dicee.TriplePredictionDataset method), 170
  _iter___() (dicee.config.Namespace method), 110
  _len__() (dicee.AllvsAll method), 168
__len__() (dicee.BPE_NegativeSamplingDataset method), 166
__len__() (dicee.dataset_classes.AllvsAll method), 114
__len__() (dicee.dataset_classes.BPE_NegativeSamplingDataset method), 111
__len__() (dicee.dataset_classes.KvsAll method), 113
__len__() (dicee.dataset_classes.KvsSampleDataset method), 114
__len__() (dicee.dataset_classes.MultiClassClassificationDataset method), 112
__len__() (dicee.dataset_classes.MultiLabelDataset method), 112
__len__() (dicee.dataset_classes.NegSampleDataset method), 115
  _len__() (dicee.dataset_classes.OnevsAllDataset method), 112
__len__() (dicee.dataset_classes.TriplePredictionDataset method), 115
__len__() (dicee.KvsAll method), 168
__len__() (dicee.KvsSampleDataset method), 169
__len__() (dicee.MultiClassClassificationDataset method), 167
  _len__() (dicee.MultiLabelDataset method), 166
__len__() (dicee.NegSampleDataset method), 169
__len__() (dicee.OnevsAllDataset method), 167
__len__() (dicee.TriplePredictionDataset method), 170
__str__() (dicee.KGE method), 160
  _str___() (dicee.knowledge_graph_embeddings.KGE method), 124
__version__(in module dicee), 174
Α
AbstractCallback (class in dicee.abstracts), 98
AbstractPPECallback (class in dicee.abstracts), 99
AbstractTrainer (class in dicee.abstracts), 94
AccumulateEpochLossCallback (class in dicee.callbacks), 101
achieve_answer() (dicee.query_generator.QueryGenerator method), 128
achieve_answer() (dicee.QueryGenerator method), 174
AConEx (class in dicee), 147
AConEx (class in dicee.models), 57
AConEx (class in dicee.models.complex), 26
AConvO (class in dicee), 147
AConvO (class in dicee.models), 67
AConvO (class in dicee.models.octonion), 33
AConvQ (class in dicee), 148
AConvQ (class in dicee.models), 63
AConvQ (class in dicee.models.quaternion), 37
adaptive_swa (dicee.config.Namespace attribute), 110
add_new_entity_embeddings() (dicee.abstracts.BaseInteractiveKGE method), 97
add_noise_rate (dicee.config.Namespace attribute), 108
add_noisy_triples() (in module dicee), 158
```

```
add noisy triples () (in module dicee.static funcs), 132
add_noisy_triples_into_training() (dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method), 83
add_noisy_triples_into_training() (dicee.read_preprocess_save_load_kg.ReadFromDisk method), 87
AllvsAll (class in dicee), 168
AllvsAll (class in dicee.dataset_classes), 113
analyse() (in module dicee.analyse_experiments), 101
answer_multi_hop_query() (dicee.KGE method), 163
answer_multi_hop_query() (dicee.knowledge_graph_embeddings.KGE method), 127
app (in module dicee.scripts.serve), 88
apply_coefficients()(dicee.DeCaL method), 143
apply_coefficients() (dicee. Keci method), 140
{\tt apply\_coefficients()} \ \textit{(dicee.models.clifford.DeCaL method)}, 24
apply_coefficients() (dicee.models.clifford.Keci method), 21
apply_coefficients() (dicee.models.DeCaL method), 73
apply_coefficients() (dicee.models.Keci method), 68
apply_reciprical_or_noise() (in module dicee.read_preprocess_save_load_kg.util), 84
ASWA (class in dicee.callbacks), 104
backend (dicee.config.Namespace attribute), 108
BaseInteractiveKGE (class in dicee.abstracts), 96
BaseKGE (class in dicee), 156
BaseKGE (class in dicee.models), 51, 53, 56, 59, 63, 74, 77
BaseKGE (class in dicee.models.base_model), 16
BaseKGELightning (class in dicee.models), 46
BaseKGELightning (class in dicee.models.base model), 10
batch_kronecker_product() (dicee.callbacks.KronE static method), 107
batch_size (dicee.config.Namespace attribute), 108
bias (dicee.models.transformers.GPTConfig attribute), 43
Block (class in dicee.models.transformers), 42
block_size (dicee.config.Namespace attribute), 110
\verb|block_size| (\textit{dicee.models.transformers.GPTC} on fig \textit{ attribute}), 43
BPE_NegativeSamplingDataset (class in dicee), 166
BPE_NegativeSamplingDataset (class in dicee.dataset_classes), 111
build_chain_funcs() (dicee.models.FMult2 method), 79
build_chain_funcs() (dicee.models.function_space.FMult2 method), 30
build_func() (dicee.models.FMult2 method), 79
build_func() (dicee.models.function_space.FMult2 method), 30
BytE (class in dicee), 154
BytE (class in dicee.models.transformers), 39
byte_pair_encoding (dicee.config.Namespace attribute), 110
callbacks (dicee.config.Namespace attribute), 108
CausalSelfAttention (class in dicee.models.transformers), 41
chain_func() (dicee.models.FMult method), 78
chain_func() (dicee.models.function_space.FMult method), 29
chain_func() (dicee.models.function_space.GFMult method), 29
chain func() (dicee.models.GFMult method), 78
cl_pqr() (dicee.DeCaL method), 143
cl_pqr() (dicee.models.clifford.DeCaL method), 23
cl_pqr() (dicee.models.DeCaL method), 72
clifford_mul() (dicee.CMult method), 137
clifford_mul() (dicee.models.clifford.CMult method), 19
clifford_mul() (dicee.models.CMult method), 70
clifford_multiplication() (dicee.Keci method), 140
clifford_multiplication() (dicee.models.clifford.Keci method), 21
clifford_multiplication() (dicee.models.Keci method), 68
CMult (class in dicee), 137
CMult (class in dicee.models), 70
CMult (class in dicee.models.clifford), 18
collate_fn() (dicee.BPE_NegativeSamplingDataset method), 166
collate_fn() (dicee.dataset_classes.BPE_NegativeSamplingDataset method), 111
collate_fn() (dicee.dataset_classes.TriplePredictionDataset method), 115
collate_fn() (dicee.TriplePredictionDataset method), 170
comp func() (dicee.LFMult method), 153
comp_func() (dicee.models.function_space.LFMult method), 31
```

```
comp func () (dicee.models.LFMult method), 80
Complex (class in dicee), 146
Complex (class in dicee.models), 58
Complex (class in dicee.models.complex), 27
compute_convergence() (in module dicee.callbacks), 104
compute_func() (dicee.models.FMult method), 78
compute_func() (dicee.models.FMult2 method), 79
compute func() (dicee.models.function space.FMult method), 29
compute_func() (dicee.models.function_space.FMult2 method), 30
compute_func() (dicee.models.function_space.GFMult method), 29
compute_func() (dicee.models.GFMult method), 78
compute_mrr() (dicee.callbacks.ASWA static method), 105
compute_sigma_pp() (dicee.DeCaL method), 144
compute_sigma_pp() (dicee.Keci method), 139
compute_sigma_pp() (dicee.models.clifford.DeCaL method). 24
compute_sigma_pp() (dicee.models.clifford.Keci method), 20
compute_sigma_pp() (dicee.models.DeCaL method),73
compute_sigma_pp() (dicee.models.Keci method), 68
\verb|compute_sigma_pq()| \textit{(dicee.DeCaL method)}, 145
compute_sigma_pq() (dicee.Keci method), 140
compute_sigma_pq() (dicee.models.clifford.DeCaL method), 25
\verb|compute_sigma_pq()| \textit{(dicee.models.clifford.Keci method)}, 20
compute_sigma_pq() (dicee.models.DeCaL method), 74
compute_sigma_pq() (dicee.models.Keci method), 68
compute_sigma_pr() (dicee.DeCaL method), 145
compute_sigma_pr() (dicee.models.clifford.DeCaL method), 25
compute_sigma_pr() (dicee.models.DeCaL method), 74
compute_sigma_qq() (dicee.DeCaL method), 144
\verb|compute_sigma_qq()| \textit{(dicee.Keci method)}, 140
compute_sigma_gg() (dicee.models.clifford.DeCaL method), 24
compute_sigma_qq() (dicee.models.clifford.Keci method), 20
compute_sigma_qq() (dicee.models.DeCaL method), 73
compute_sigma_qq() (dicee.models.Keci method), 68
compute_sigma_qr() (dicee.DeCaL method), 145
compute_sigma_gr() (dicee.models.clifford.DeCaL method), 25
compute_sigma_qr() (dicee.models.DeCaL method), 74
compute sigma rr() (dicee.DeCaL method), 144
compute_sigma_rr() (dicee.models.clifford.DeCaL method), 25
compute_sigma_rr() (dicee.models.DeCaL method), 74
compute_sigmas_multivect() (dicee.DeCaL method), 143
compute_sigmas_multivect() (dicee.models.clifford.DeCaL method), 23
compute_sigmas_multivect() (dicee.models.DeCaL method), 72
compute_sigmas_single() (dicee.DeCaL method), 143
compute_sigmas_single() (dicee.models.clifford.DeCaL method), 23
compute_sigmas_single() (dicee.models.DeCaL method), 72
ConEx (class in dicee), 149
ConEx (class in dicee.models), 57
ConEx (class in dicee.models.complex), 26
configure_optimizers() (dicee.models.base_model.BaseKGELightning method), 15
configure_optimizers() (dicee.models.BaseKGELightning method), 50
configure optimizers () (dicee.models.transformers.GPT method), 44
construct_cl_multivector() (dicee.DeCaL method), 143
construct_cl_multivector() (dicee.Keci method), 140
construct_cl_multivector() (dicee.models.clifford.DeCaL method), 24
construct_cl_multivector() (dicee.models.clifford.Keci method), 21
construct_cl_multivector() (dicee.models.DeCaL method), 73
construct_cl_multivector() (dicee.models.Keci method), 69
\verb|construct_dataset()| \textit{(in module dicee)}, 166
construct_dataset() (in module dicee.dataset_classes), 111
construct_graph() (dicee.query_generator.QueryGenerator method), 128
construct_graph() (dicee.QueryGenerator method), 174
construct_input_and_output() (dicee.abstracts.BaseInteractiveKGE method), 98
construct_multi_coeff() (dicee.LFMult method), 152
construct_multi_coeff() (dicee.models.function_space.LFMult method), 30
construct_multi_coeff() (dicee.models.LFMult method), 79
continual_learning (dicee.config.Namespace attribute), 110
continual_start() (dicee.DICE_Trainer method), 159
continual_start() (dicee.executer.ContinuousExecute method), 123
```

```
continual start() (dicee.trainer.DICE Trainer method), 93
continual_start() (dicee.trainer.dice_trainer.DICE_Trainer method), 89
continual_training_setup_executor() (in module dicee), 158
continual_training_setup_executor() (in module dicee.static_funcs), 132
Continuous Execute (class in dicee.executer), 123
ConvO (class in dicee), 148
ConvO (class in dicee.models), 66
ConvO (class in dicee.models.octonion), 33
ConvQ (class in dicee), 148
ConvQ (class in dicee.models), 62
ConvQ (class in dicee.models.quaternion), 37
create_constraints() (in module dicee.read_preprocess_save_load_kg.util), 85
create_constraints() (in module dicee.static_preprocess_funcs), 134
create_experiment_folder() (in module dicee), 158
\verb|create_experiment_folder()| \textit{(in module dicee.static\_funcs)}, 132
create_random_data() (dicee.callbacks.PseudoLabellingCallback method), 104
create_recipriocal_triples() (in module dicee), 157
create_recipriocal_triples() (in module dicee.read_preprocess_save_load_kg.util), 85
create_recipriocal_triples() (in module dicee.static_funcs), 131
create_vector_database() (dicee.KGE method), 160
create_vector_database() (dicee.knowledge_graph_embeddings.KGE method), 124
crop_block_size() (dicee.models.transformers.GPT method), 44
CVDataModule (class in dicee), 170
{\tt CVDataModule}~({\it class~in~dicee.dataset\_classes}), 115
D
dataset_dir (dicee.config.Namespace attribute), 108
dataset_sanity_checking() (in module dicee.read_preprocess_save_load_kg.util), 85
DDPTrainer (class in dicee.trainer.torch_trainer_ddp), 92
DeCaL (class in dicee), 142
DeCaL (class in dicee.models), 71
DeCaL (class in dicee.models.clifford), 22
decide() (dicee.callbacks.ASWA method), 105
deploy() (dicee.KGE method), 164
deploy() (dicee.knowledge_graph_embeddings.KGE method), 128
deploy_head_entity_prediction() (in module dicee), 158
deploy_head_entity_prediction() (in module dicee.static_funcs), 132
deploy_relation_prediction() (in module dicee), 158
deploy_relation_prediction() (in module dicee.static_funcs), 132
deploy_tail_entity_prediction() (in module dicee), 158
deploy_tail_entity_prediction() (in module dicee.static_funcs), 132
deploy_triple_prediction() (in module dicee), 158
deploy_triple_prediction() (in module dicee.static_funcs), 132
DICE_Trainer (class in dicee), 159
DICE_Trainer (class in dicee.trainer), 93
DICE_Trainer (class in dicee.trainer.dice_trainer), 89
dicee
     module, 10
dicee.abstracts
     module. 94
dicee.analyse_experiments
    module, 100
dicee.callbacks
     module, 101
dicee.confia
     module, 107
dicee.dataset_classes
     module, 110
dicee.eval_static_funcs
    module, 119
dicee.evaluator
     module 120
dicee.executer
     module, 121
dicee.knowledge_graph
     module, 123
dicee.knowledge_graph_embeddings
```

```
module, 124
dicee.models
    module. 10
dicee.models.base_model
   module, 10
dicee.models.clifford
    module, 18
dicee.models.complex
    module, 26
dicee.models.dualE
    module, 28
dicee.models.function_space
    module, 29
dicee.models.octonion
    module 31
dicee.models.pykeen_models
   module, 34
dicee.models.quaternion
    module, 35
dicee.models.real
    module, 37
dicee.models.static_funcs
   module, 39
\verb|dicee.models.transformers|\\
    module, 39
dicee.query_generator
   module, 128
dicee.read_preprocess_save_load_kg
    module, 81
dicee.read_preprocess_save_load_kq.preprocess
    module, 81
dicee.read_preprocess_save_load_kg.read_from_disk
    module, 82
dicee.read_preprocess_save_load_kg.save_load_disk
    module, 83
dicee.read_preprocess_save_load_kg.util
   module, 83
dicee.sanity_checkers
    module, 129
dicee.scripts
    module, 87
dicee.scripts.index
   module, 87
dicee.scripts.run
   module, 87
dicee.scripts.serve
   module, 88
dicee.static_funcs
    module, 130
dicee.static_funcs_training
    module, 133
dicee.static_preprocess_funcs
    module, 133
dicee.trainer
    module, 89
dicee.trainer.dice_trainer
    module, 89
dicee.trainer.torch_trainer
    module, 90
dicee.trainer.torch_trainer_ddp
    module, 91
DistMult (class in dicee), 138
DistMult (class in dicee.models), 55
DistMult (class in dicee.models.real), 38
download_file() (in module dicee), 158
download_file() (in module dicee.static_funcs), 132
download_files_from_url() (in module dicee), 158
download_files_from_url() (in module dicee.static_funcs), 132
```

```
download_pretrained_model() (in module dicee), 159
download_pretrained_model() (in module dicee.static_funcs), 133
dropout (dicee.models.transformers.GPTConfig attribute), 43
DualE (class in dicee), 145
DualE (class in dicee.models), 80
DualE (class in dicee.models.dualE), 28
dummy_eval() (dicee.evaluator.Evaluator method), 121
F
efficient_zero_grad() (in module dicee.static_funcs_training), 133
embedding_dim (dicee.config.Namespace attribute), 108
enable_log (in module dicee.static_preprocess_funcs), 134
end() (dicee.Execute method), 165
end() (dicee.executer.Execute method), 122
entities_str (dicee.knowledge_graph.KG property), 123
estimate_mfu() (dicee.models.transformers.GPT method), 44
estimate_q() (in module dicee.callbacks), 104
Eval (class in dicee.callbacks), 105
eval () (dicee.evaluator.Evaluator method), 120
eval_lp_performance() (dicee.KGE method), 160
eval_lp_performance() (dicee.knowledge_graph_embeddings.KGE method), 124
eval_model (dicee.config.Namespace attribute), 109
eval_rank_of_head_and_tail_byte_pair_encoded_entity() (dicee.evaluator.Evaluator method), 120
eval_rank_of_head_and_tail_entity() (dicee.evaluator.Evaluator method), 120
eval_with_bpe_vs_all() (dicee.evaluator.Evaluator method), 120
eval_with_byte() (dicee.evaluator.Evaluator method), 120
eval_with_data() (dicee.evaluator.Evaluator method), 121
eval_with_vs_all() (dicee.evaluator.Evaluator method), 120
evaluate() (in module dicee), 158
evaluate() (in module dicee.static_funcs), 132
evaluate_bpe_lp() (in module dicee.static_funcs_training), 133
evaluate_link_prediction_performance() (in module dicee.eval_static_funcs), 119
evaluate_link_prediction_performance_with_bpe() (in module dicee.eval_static_funcs), 119
evaluate_link_prediction_performance_with_bpe_reciprocals() (in module dicee.eval_static_funcs), 119
evaluate_link_prediction_performance_with_reciprocals() (in module dicee.eval_static_funcs), 119
evaluate_lp() (dicee.evaluator.Evaluator method), 121
evaluate_lp() (in module dicee.static_funcs_training), 133
evaluate_lp_bpe_k_vs_all() (dicee.evaluator.Evaluator method), 121
evaluate_lp_bpe_k_vs_all() (in module dicee.eval_static_funcs), 120
evaluate_lp_k_vs_all() (dicee.evaluator.Evaluator method), 121
evaluate_lp_with_byte() (dicee.evaluator.Evaluator method), 121
Evaluator (class in dicee.evaluator), 120
Execute (class in dicee), 164
Execute (class in dicee.executer), 121
Experiment (class in dicee.analyse_experiments), 101
exponential_function() (in module dicee), 158
{\tt exponential\_function()} \ \textit{(in module dicee.static\_funcs)}, 132
extract input outputs() (dicee.trainer.torch trainer ddp.DDPTrainer method), 92
extract_input_outputs() (dicee.trainer.torch_trainer_ddp.NodeTrainer method), 92
extract_input_outputs_set_device() (dicee.trainer.torch_trainer.TorchTrainer method), 91
feature_map_dropout_rate (dicee.config.Namespace attribute), 110
fill_query() (dicee.query_generator.QueryGenerator method), 128
fill_query() (dicee.QueryGenerator method), 174
find_missing_triples() (dicee.KGE method), 163
find_missing_triples() (dicee.knowledge_graph_embeddings.KGE method), 127
fit () (dicee.trainer.torch_trainer_ddp.TorchDDPTrainer method), 92
fit () (dicee.trainer.torch_trainer.TorchTrainer method), 91
FMult (class in dicee.models), 78
FMult (class in dicee.models.function_space), 29
FMult2 (class in dicee.models), 78
FMult2 (class in dicee.models.function_space), 30
forward() (dicee.BaseKGE method), 156
forward() (dicee.BytE method), 154
forward() (dicee.models.base_model.BaseKGE method), 17
forward() (dicee.models.base_model.IdentityClass static method), 18
```

```
forward() (dicee.models.BaseKGE method), 52, 54, 56, 59, 64, 75, 77
forward() (dicee.models.IdentityClass static method), 53, 61, 65
forward() (dicee.models.transformers.Block method), 43
forward() (dicee.models.transformers.BytE method), 40
forward() (dicee.models.transformers.CausalSelfAttention method), 42
forward() (dicee.models.transformers.GPT method), 44
forward() (dicee.models.transformers.LayerNorm method), 41
forward() (dicee.models.transformers.MLP method), 42
forward_backward_update() (dicee.trainer.torch_trainer.TorchTrainer method), 91
forward_byte_pair_encoded_k_vs_all() (dicee.BaseKGE method), 156
forward_byte_pair_encoded_k_vs_all() (dicee.models.base_model.BaseKGE method), 17
forward_byte_pair_encoded_k_vs_all() (dicee.models.BaseKGE method), 52, 54, 56, 59, 63, 75, 77
forward_byte_pair_encoded_triple() (dicee.BaseKGE method), 156
forward_byte_pair_encoded_triple() (dicee.models.base_model.BaseKGE method), 17
forward_byte_pair_encoded_triple() (dicee.models.BaseKGE method), 52, 54, 56, 59, 64, 75, 77
forward_k_vs_all() (dicee.AConEx method), 147
forward_k_vs_all() (dicee.AConvO method), 148
forward_k_vs_all() (dicee.AConvQ method), 148
forward_k_vs_all() (dicee.BaseKGE method), 157
forward_k_vs_all() (dicee.CMult method), 138
forward_k_vs_all() (dicee.ComplEx method), 147
forward_k_vs_all() (dicee.ConEx method), 149
forward_k_vs_all() (dicee.ConvO method), 149
forward_k_vs_all() (dicee.ConvQ method), 148
forward_k_vs_all() (dicee.DeCaL method), 143
forward_k_vs_all() (dicee.DistMult method), 138
{\tt forward\_k\_vs\_all}~(\textit{)}~(\textit{dicee.DualE method}),~146
forward_k_vs_all() (dicee.Keci method), 141
forward_k_vs_all() (dicee.models.AConEx method), 58
forward_k_vs_all() (dicee.models.AConvO method), 67
forward_k_vs_all() (dicee.models.AConvQ method), 63
forward_k_vs_all() (dicee.models.base_model.BaseKGE method), 17
forward_k_vs_all() (dicee.models.BaseKGE method), 52, 54, 57, 60, 64, 76, 78
forward_k_vs_all() (dicee.models.clifford.CMult method), 19
forward_k_vs_all() (dicee.models.clifford.DeCaL method), 24
forward_k_vs_all() (dicee.models.clifford.Keci method), 21
forward k vs all() (dicee.models.CMult method), 71
forward_k_vs_all() (dicee.models.ComplEx method), 59
forward_k_vs_all() (dicee.models.complex.AConEx method), 26
forward_k_vs_all() (dicee.models.complex.ComplEx method), 27
forward_k_vs_all() (dicee.models.complex.ConEx method), 26
forward_k_vs_all() (dicee.models.ConEx method), 57
forward_k_vs_all() (dicee.models.ConvO method), 67
forward_k_vs_all() (dicee.models.ConvQ method), 62
forward_k_vs_all() (dicee.models.DeCaL method), 72
forward_k_vs_all() (dicee.models.DistMult method), 55
forward_k_vs_all() (dicee.models.DualE method), 81
forward_k_vs_all() (dicee.models.dualE.DualE method), 28
forward_k_vs_all() (dicee.models.Keci method), 69
forward_k_vs_all() (dicee.models.octonion.AConvO method), 34
forward_k_vs_all() (dicee.models.octonion.ConvO method), 33
forward_k_vs_all() (dicee.models.octonion.OMult method), 33
forward_k_vs_all() (dicee.models.OMult method), 66
forward_k_vs_all() (dicee.models.pykeen_models.PykeenKGE method), 34
forward_k_vs_all() (dicee.models.PykeenKGE method), 76
forward_k_vs_all() (dicee.models.QMult method), 62
forward_k_vs_all() (dicee.models.quaternion.AConvQ method), 37
{\tt forward\_k\_vs\_all()} \ (\textit{dicee.models.quaternion.ConvQ method}), 37
forward_k_vs_all() (dicee.models.quaternion.QMult method), 36
forward_k_vs_all() (dicee.models.real.DistMult method), 38
forward_k_vs_all() (dicee.models.real.Shallom method), 38
forward_k_vs_all() (dicee.models.real.TransE method), 38
forward_k_vs_all() (dicee.models.Shallom method), 55
forward_k_vs_all() (dicee.models.TransE method), 55
forward_k_vs_all() (dicee.OMult method), 152
forward_k_vs_all() (dicee.PykeenKGE method), 153
forward_k_vs_all() (dicee.QMult method), 151
forward_k_vs_all() (dicee.Shallom method), 152
```

```
forward k vs all() (dicee. Trans E method), 142
forward_k_vs_sample() (dicee.AConEx method), 147
forward_k_vs_sample() (dicee.BaseKGE method), 157
forward_k_vs_sample() (dicee.ConEx method), 149
forward_k_vs_sample() (dicee.DistMult method), 138
forward_k_vs_sample() (dicee.Keci method), 141
forward_k_vs_sample() (dicee.models.AConEx method), 58
forward k vs sample() (dicee.models.base model.BaseKGE method), 17
forward_k_vs_sample() (dicee.models.BaseKGE method), 52, 54, 57, 60, 64, 76, 78
forward_k_vs_sample() (dicee.models.clifford.Keci method), 22
forward_k_vs_sample() (dicee.models.complex.AConEx method), 26
forward_k_vs_sample() (dicee.models.complex.ConEx method), 26
forward_k_vs_sample() (dicee.models.ConEx method), 57
forward_k_vs_sample() (dicee.models.DistMult method), 55
forward_k_vs_sample() (dicee.models.Keci method), 69
forward_k_vs_sample() (dicee.models.pykeen_models.PykeenKGE method), 35
forward_k_vs_sample() (dicee.models.PykeenKGE method),77
forward_k_vs_sample() (dicee.models.QMult method), 62
forward_k_vs_sample() (dicee.models.quaternion.QMult method), 37
forward_k_vs_sample() (dicee.models.real.DistMult method), 38
forward_k_vs_sample() (dicee.PykeenKGE method), 154
forward_k_vs_sample() (dicee.QMult method), 151
forward_k_vs_with_explicit() (dicee.Keci method), 141
forward_k_vs_with_explicit() (dicee.models.clifford.Keci method), 21
forward_k_vs_with_explicit() (dicee.models.Keci method), 69
forward_triples() (dicee.AConEx method), 147
forward_triples() (dicee.AConvO method), 147
forward_triples() (dicee.AConvQ method), 148
forward_triples() (dicee.BaseKGE method), 157
forward_triples() (dicee.CMult method), 138
forward_triples() (dicee.ConEx method), 149
forward triples () (dicee. ConvO method), 149
forward_triples() (dicee.ConvQ method), 148
forward_triples() (dicee.DeCaL method), 142
forward_triples() (dicee.DualE method), 146
forward_triples() (dicee.Keci method), 141
forward triples() (dicee.LFMult method), 152
forward_triples() (dicee.models.AConEx method), 58
forward_triples() (dicee.models.AConvO method), 67
forward_triples() (dicee.models.AConvQ method), 63
forward_triples() (dicee.models.base_model.BaseKGE method), 17
forward_triples() (dicee.models.BaseKGE method), 52, 54, 57, 60, 64, 75, 78
forward_triples() (dicee.models.clifford.CMult method), 19
forward_triples() (dicee.models.clifford.DeCaL method), 23
forward_triples() (dicee.models.clifford.Keci method), 22
forward_triples() (dicee.models.CMult method), 71
forward_triples() (dicee.models.complex.AConEx method), 26
forward_triples() (dicee.models.complex.ConEx method), 26
forward_triples() (dicee.models.ConEx method), 57
forward_triples() (dicee.models.ConvO method), 67
forward_triples() (dicee.models.ConvQ method), 62
forward_triples() (dicee.models.DeCaL method), 72
forward_triples() (dicee.models.DualE method), 81
forward_triples() (dicee.models.dualE.DualE method), 28
forward_triples() (dicee.models.FMult method), 78
forward_triples() (dicee.models.FMult2 method), 79
forward_triples() (dicee.models.function_space.FMult method), 29
forward_triples() (dicee.models.function_space.FMult2 method), 30
forward_triples() (dicee.models.function_space.GFMult method), 29
forward_triples() (dicee.models.function_space.LFMult method), 30
forward triples() (dicee.models.function space.LFMult1 method), 30
forward_triples() (dicee.models.GFMult method), 78
forward_triples() (dicee.models.Keci method), 70
forward_triples() (dicee.models.LFMult method), 79
forward_triples() (dicee.models.LFMult1 method), 79
forward_triples() (dicee.models.octonion.AConvO method), 34
forward_triples() (dicee.models.octonion.ConvO method), 33
forward_triples() (dicee.models.Pyke method), 55
```

```
forward triples() (dicee.models.pykeen models.PykeenKGE method), 34
forward_triples() (dicee.models.PykeenKGE method), 76
forward_triples() (dicee.models.quaternion.AConvQ method), 37
forward_triples() (dicee.models.quaternion.ConvQ method), 37
forward_triples() (dicee.models.real.Pyke method), 38
forward_triples() (dicee.models.real.Shallom method), 38
forward_triples() (dicee.models.Shallom method), 55
forward triples() (dicee.Pyke method), 138
forward_triples() (dicee.PykeenKGE method), 153
forward_triples() (dicee.Shallom method), 152
from_pretrained() (dicee.models.transformers.GPT class method), 44
func_triple_to_bpe_representation() (dicee.knowledge_graph.KG method), 123
function() (dicee.models.FMult2 method), 79
function() (dicee.models.function_space.FMult2 method), 30
G
generate() (dicee.BytE method), 154
generate() (dicee.KGE method), 160
generate() (dicee.knowledge_graph_embeddings.KGE method), 124
generate() (dicee.models.transformers.BytE method), 40
generate_queries() (dicee.query_generator.QueryGenerator method), 129
generate_queries() (dicee.QueryGenerator method), 174
get () (dicee.scripts.serve.NeuralSearcher method), 89
get_aswa_state_dict() (dicee.callbacks.ASWA method), 105
get_bpe_head_and_relation_representation() (dicee.BaseKGE method), 157
get_bpe_head_and_relation_representation() (dicee.models.base_model.BaseKGE method), 17
get_bpe_head_and_relation_representation() (dicee.models.BaseKGE method), 52, 55, 57, 60, 64, 76, 78
get_bpe_token_representation() (dicee.abstracts.BaseInteractiveKGE method), 96
get_callbacks() (in module dicee.trainer.dice_trainer), 89
get_default_arguments() (in module dicee.analyse_experiments), 101
get_default_arguments() (in module dicee.scripts.index), 87
get_default_arguments() (in module dicee.scripts.run), 88
\verb"get_default_arguments"() \textit{ (in module dicee.scripts.serve)}, 88
get_domain_of_relation() (dicee.abstracts.BaseInteractiveKGE method), 96
get_ee_vocab() (in module dicee), 157
get_ee_vocab() (in module dicee.read_preprocess_save_load_kg.util), 85
get_ee_vocab() (in module dicee.static_funcs), 131
get_ee_vocab() (in module dicee.static_preprocess_funcs), 134
get_embeddings() (dicee.BaseKGE method), 157
get_embeddings() (dicee.models.base_model.BaseKGE method), 17
get embeddings () (dicee.models.BaseKGE method), 53, 55, 57, 60, 64, 76, 78
get_embeddings() (dicee.models.real.Shallom method), 38
get_embeddings() (dicee.models.Shallom method), 55
get_embeddings() (dicee.Shallom method), 152
get_entity_embeddings() (dicee.abstracts.BaseInteractiveKGE method), 97
get_entity_index() (dicee.abstracts.BaseInteractiveKGE method), 97
get_er_vocab() (in module dicee), 157
get_er_vocab() (in module dicee.read_preprocess_save_load_kg.util), 85
get_er_vocab() (in module dicee.static_funcs), 131
get_er_vocab() (in module dicee.static_preprocess_funcs), 134
get_eval_report() (dicee.abstracts.BaseInteractiveKGE method), 96
get_head_relation_representation() (dicee.BaseKGE method), 157
get_head_relation_representation() (dicee.models.base_model.BaseKGE method), 17
get_head_relation_representation() (dicee.models.BaseKGE method), 52, 54, 57, 60, 64, 76, 78
get_kronecker_triple_representation() (dicee.callbacks.KronE method), 107
get_num_params() (dicee.models.transformers.GPT method), 44
get_padded_bpe_triple_representation() (dicee.abstracts.BaseInteractiveKGE method), 96
get_queries() (dicee.query_generator.QueryGenerator method), 129
get_queries() (dicee.QueryGenerator method), 174
get_range_of_relation() (dicee.abstracts.BaseInteractiveKGE method), 96
get_re_vocab() (in module dicee), 157
get_re_vocab() (in module dicee.read_preprocess_save_load_kg.util), 85
get_re_vocab() (in module dicee.static_funcs), 131
get_re_vocab() (in module dicee.static_preprocess_funcs), 134
get_relation_embeddings() (dicee.abstracts.BaseInteractiveKGE method), 98
get_relation_index() (dicee.abstracts.BaseInteractiveKGE method), 97
get_sentence_representation() (dicee.BaseKGE method), 157
```

```
get sentence representation() (dicee.models.base model.BaseKGE method), 17
get_sentence_representation() (dicee.models.BaseKGE method), 52, 54, 57, 60, 64, 76, 78
get_transductive_entity_embeddings() (dicee.KGE method), 160
get_transductive_entity_embeddings() (dicee.knowledge_graph_embeddings.KGE method), 124
get_triple_representation() (dicee.BaseKGE method), 157
get_triple_representation() (dicee.models.base_model.BaseKGE method), 17
get_triple_representation() (dicee.models.BaseKGE method), 52, 54, 57, 60, 64, 76, 78
GFMult (class in dicee.models), 78
GFMult (class in dicee.models.function_space), 29
GPT (class in dicee.models.transformers), 43
GPTConfig (class in dicee.models.transformers), 43
gpus (dicee.config.Namespace attribute), 108
gradient_accumulation_steps (dicee.config.Namespace attribute), 109
ground_queries() (dicee.query_generator.QueryGenerator method), 129
ground_queries() (dicee.QueryGenerator method), 174
Н
hidden_dropout_rate (dicee.config.Namespace attribute), 110
IdentityClass (class in dicee.models), 53, 60, 64
IdentityClass (class in dicee.models.base_model), 17
index_triple() (dicee.abstracts.BaseInteractiveKGE method), 97
index_triples_with_pandas() (in module dicee.read_preprocess_save_load_kg.util), 85
\verb"init_param" (\textit{dicee.config.Namespace attribute}), 109
init_params_with_sanity_checking() (dicee.BaseKGE method), 156
\verb|init_params_with_sanity_checking()| \textit{(dicee.models.base\_model.BaseKGE method)}, 17
init_params_with_sanity_checking() (dicee.models.BaseKGE method), 52, 54, 56, 59, 64, 75, 77
initialize_dataloader() (dicee.DICE_Trainer method), 159
initialize_dataloader() (dicee.trainer.DICE_Trainer method), 93
initialize_dataloader() (dicee.trainer.dice_trainer.DICE_Trainer method), 90
initialize_dataset() (dicee.DICE_Trainer method), 159
initialize_dataset() (dicee.trainer.DICE_Trainer method), 93
initialize_dataset() (dicee.trainer.dice_trainer.DICE_Trainer method), 90
initialize_or_load_model() (dicee.DICE_Trainer method), 159
initialize_or_load_model() (dicee.trainer.DICE_Trainer method), 93
initialize_or_load_model() (dicee.trainer.dice_trainer.DICE_Trainer method), 90
initialize_trainer() (dicee.DICE_Trainer method), 159
initialize_trainer() (dicee.trainer.DICE_Trainer method), 93
initialize_trainer() (dicee.trainer.dice_trainer.DICE_Trainer method), 90
initialize_trainer() (in module dicee.trainer.dice_trainer), 89
input_dropout_rate (dicee.config.Namespace attribute), 110
intialize_model() (in module dicee), 158
\verb|intialize_model()| \textit{(in module dicee.static\_funcs)}, 132
is_seen() (dicee.abstracts.BaseInteractiveKGE method), 97
is_sparql_endpoint_alive() (in module dicee.sanity_checkers), 129
K
k_fold_cross_validation() (dicee.DICE_Trainer method), 159
k_fold_cross_validation() (dicee.trainer.DICE_Trainer method), 93
k_fold_cross_validation() (dicee.trainer.dice_trainer.DICE_Trainer method), 90
k_vs_all_score() (dicee.ComplEx static method), 147
k_vs_all_score() (dicee.DistMult method), 138
k_vs_all_score() (dicee.Keci method), 141
k\_vs\_all\_score() (dicee.models.clifford.Keci method), 21
k_vs_all_score() (dicee.models.ComplEx static method), 58
k_vs_all_score() (dicee.models.complex.ComplEx static method), 27
k_vs_all_score() (dicee.models.DistMult method), 55
k_vs_all_score() (dicee.models.Keci method), 69
k_vs_all_score() (dicee.models.octonion.OMult method), 32
k vs all score() (dicee.models.OMult method), 66
k_vs_all_score() (dicee.models.QMult method), 62
\verb|k_vs_all_score|() \textit{ (dicee.models.quaternion.QMult method)}, 36
k_vs_all_score() (dicee.models.real.DistMult method), 38
k_vs_all_score() (dicee.OMult method), 152
k_vs_all_score() (dicee.QMult method), 151
```

```
Keci (class in dicee), 139
Keci (class in dicee.models), 67
Keci (class in dicee.models.clifford), 19
KeciBase (class in dicee), 139
KeciBase (class in dicee.models), 70
KeciBase (class in dicee.models.clifford), 22
kernel_size (dicee.config.Namespace attribute), 109
KG (class in dicee.knowledge_graph), 123
KGE (class in dicee), 160
KGE (class in dicee.knowledge_graph_embeddings), 124
KGESaveCallback (class in dicee.callbacks), 103
KronE (class in dicee.callbacks), 106
KvsAll (class in dicee), 167
KvsAll (class in dicee.dataset_classes), 112
kvsall_score() (dicee.DualE method), 145
kvsall_score() (dicee.models.DualE method), 80
kvsall_score() (dicee.models.dualE.DualE method), 28
KvsSampleDataset (class in dicee), 169
KvsSampleDataset (class in dicee.dataset_classes), 114
LayerNorm (class in dicee.models.transformers), 41
LFMult (class in dicee), 152
LFMult (class in dicee.models), 79
LFMult (class in dicee.models.function_space), 30
LFMult1 (class in dicee.models), 79
LFMult1 (class in dicee.models.function_space), 30
linear() (dicee.LFMult method), 152
linear() (dicee.models.function_space.LFMult method), 30
linear() (dicee.models.LFMult method), 79
list2tuple() (dicee.query_generator.QueryGenerator method), 128
list2tuple() (dicee.QueryGenerator method), 174
load() (dicee.read_preprocess_save_load_kg.LoadSaveToDisk method), 87
load() (dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk method), 83
load_indexed_data() (dicee.Execute method), 164
load_indexed_data() (dicee.executer.Execute method), 122
load_json() (in module dicee), 158
load_json() (in module dicee.static_funcs), 132
load_model() (in module dicee), 157
load_model() (in module dicee.static_funcs), 131
load_model_ensemble() (in module dicee), 157
load_model_ensemble() (in module dicee.static_funcs), 131
load_numpy() (in module dicee), 158
load_numpy() (in module dicee.static_funcs), 132
load_numpy_ndarray() (in module dicee.read_preprocess_save_load_kg.util), 85
load_pickle() (in module dicee), 157, 165
load_pickle() (in module dicee.read_preprocess_save_load_kg.util), 85
load_pickle() (in module dicee.static_funcs), 131
load_queries() (dicee.query_generator.QueryGenerator method), 129
load_queries() (dicee.QueryGenerator method), 174
load_queries_and_answers() (dicee.query_generator.QueryGenerator static method), 129
load_queries_and_answers() (dicee.QueryGenerator static method), 174
load_with_pandas() (in module dicee.read_preprocess_save_load_kg.util), 85
LoadSaveToDisk (class in dicee.read_preprocess_save_load_kg), 86
LoadSaveToDisk (class in dicee.read_preprocess_save_load_kg.save_load_disk), 83
loss_function() (dicee.BytE method), 154
loss_function() (dicee.models.base_model.BaseKGELightning method), 12
loss_function() (dicee.models.BaseKGELightning method), 47
loss_function() (dicee.models.transformers.BytE method), 40
1r (dicee.config.Namespace attribute), 108
main() (in module dicee.scripts.index), 87
main() (in module dicee.scripts.run), 88
main() (in module dicee.scripts.serve), 89
mapping_from_first_two_cols_to_third() (in module dicee), 165
mapping_from_first_two_cols_to_third() (in module dicee.static_preprocess_funcs), 134
```

```
mem of model() (dicee.models.base model.BaseKGELightning method), 11
mem_of_model() (dicee.models.BaseKGELightning method), 46
MLP (class in dicee.models.transformers), 42
model (dicee.config.Namespace attribute), 108
module
     dicee, 10
    {\tt dicee.abstracts}, 94
     dicee.analyse experiments, 100
     dicee.callbacks, 101
     dicee.config, 107
     dicee.dataset_classes, 110
     dicee.eval_static_funcs, 119
     dicee.evaluator, 120
     dicee.executer, 121
     dicee.knowledge_graph, 123
     dicee.knowledge_graph_embeddings, 124
    dicee.models, 10
     dicee.models.base_model, 10
     dicee.models.clifford, 18
     dicee.models.complex, 26
     dicee.models.dualE, 28
     dicee.models.function_space, 29
     dicee.models.octonion, 31
     dicee.models.pykeen_models,34
     dicee.models.quaternion, 35
     dicee.models.real, 37
     dicee.models.static_funcs, 39
     dicee.models.transformers, 39
     dicee.query_generator, 128
     dicee.read_preprocess_save_load_kg,81
     dicee.read_preprocess_save_load_kg.preprocess, 81
     dicee.read_preprocess_save_load_kg.read_from_disk,82
     dicee.read_preprocess_save_load_kg.save_load_disk,83
     dicee.read_preprocess_save_load_kg.util,83
     dicee.sanity_checkers, 129
     dicee.scripts,87
     dicee.scripts.index,87
     dicee.scripts.run,87
     dicee.scripts.serve,88
     dicee.static_funcs, 130
     dicee.static_funcs_training, 133
     dicee.static_preprocess_funcs, 133
     dicee.trainer,89
     dicee.trainer.dice_trainer,89
     {\tt dicee.trainer.torch\_trainer,90}
     dicee.trainer.torch_trainer_ddp,91
MultiClassClassificationDataset (class in dicee), 166
MultiClassClassificationDataset (class in dicee.dataset_classes), 112
MultiLabelDataset (class in dicee), 166
MultiLabelDataset (class in dicee.dataset_classes), 111
n_embd (dicee.models.transformers.GPTConfig attribute), 43
n_head (dicee.models.transformers.GPTConfig attribute), 43
n_layer (dicee.models.transformers.GPTConfig attribute), 43
name (dicee.abstracts.BaseInteractiveKGE property), 97
Namespace (class in dicee.config), 108
neg_ratio (dicee.config.Namespace attribute), 109
negnorm() (dicee.KGE method), 163
negnorm() (dicee.knowledge_graph_embeddings.KGE method), 127
NegSampleDataset (class in dicee), 169
NegSampleDataset (class in dicee.dataset_classes), 114
neural_searcher (in module dicee.scripts.serve), 88
Neural Searcher (class in dicee.scripts.serve), 89
NodeTrainer (class in dicee.trainer.torch_trainer_ddp), 92
normalization (dicee.config.Namespace attribute), 109
num_core (dicee.config.Namespace attribute), 109
```

```
num epochs (dicee.config.Namespace attribute), 108
num_folds_for_cv (dicee.config.Namespace attribute), 109
num_of_output_channels (dicee.config.Namespace attribute), 109
numpy_data_type_changer() (in module dicee), 158
numpy_data_type_changer() (in module dicee.static_funcs), 131
O
octonion_mul() (in module dicee.models), 65
octonion mul () (in module dicee.models.octonion), 32
octonion_mul_norm() (in module dicee.models), 65
\verb|octonion_mul_norm()| \textit{ (in module dicee.models.octonion)}, 32
octonion_normalizer() (dicee.AConvO static method), 147
octonion_normalizer() (dicee.ConvO static method), 149
octonion_normalizer() (dicee.models.AConvO static method), 67
octonion_normalizer() (dicee.models.ConvO static method), 66
octonion_normalizer() (dicee.models.octonion.AConvO static method), 34
octonion_normalizer() (dicee.models.octonion.ConvO static method), 33
octonion_normalizer() (dicee.models.octonion.OMult static method), 32
octonion normalizer() (dicee.models.OMult static method), 66
octonion_normalizer() (dicee.OMult static method), 152
OMult (class in dicee), 151
OMult (class in dicee.models), 65
OMult (class in dicee.models.octonion), 32
on_epoch_end() (dicee.callbacks.KGESaveCallback method), 104
on_epoch_end() (dicee.callbacks.PseudoLabellingCallback method), 104
on_fit_end() (dicee.abstracts.AbstractCallback method), 99
on_fit_end() (dicee.abstracts.AbstractPPECallback method), 100
on_fit_end() (dicee.abstracts.AbstractTrainer method), 95
on_fit_end() (dicee.callbacks.AccumulateEpochLossCallback method), 102
on_fit_end() (dicee.callbacks.ASWA method), 104
on_fit_end() (dicee.callbacks.Eval method), 106
on_fit_end() (dicee.callbacks.KGESaveCallback method), 104
on_fit_end() (dicee.callbacks.PrintCallback method), 102
on_fit_start() (dicee.abstracts.AbstractCallback method), 98
on_fit_start() (dicee.abstracts.AbstractPPECallback method), 100
on_fit_start() (dicee.abstracts.AbstractTrainer method), 94
on_fit_start() (dicee.callbacks.Eval method), 105
on_fit_start() (dicee.callbacks.KGESaveCallback method), 103
on_fit_start() (dicee.callbacks.KronE method), 107
on_fit_start() (dicee.callbacks.PrintCallback method), 102
on_init_end() (dicee.abstracts.AbstractCallback method), 98
on_init_start() (dicee.abstracts.AbstractCallback method), 98
on_train_batch_end() (dicee.abstracts.AbstractCallback method), 99
on_train_batch_end() (dicee.abstracts.AbstractTrainer method), 95
\verb"on_train_batch_end()" (\textit{dicee.callbacks.Eval method}), 106
on_train_batch_end() (dicee.callbacks.KGESaveCallback method), 103
on_train_batch_end() (dicee.callbacks.PrintCallback method), 102
on train batch start () (dicee.callbacks.Perturb method), 107
on_train_epoch_end() (dicee.abstracts.AbstractCallback method), 99
on_train_epoch_end() (dicee.abstracts.AbstractTrainer method), 95
on_train_epoch_end() (dicee.callbacks.ASWA method), 105
on_train_epoch_end() (dicee.callbacks.Eval method), 106
on_train_epoch_end() (dicee.callbacks.KGESaveCallback method), 103
on_train_epoch_end() (dicee.callbacks.PrintCallback method), 103
on_train_epoch_end() (dicee.models.base_model.BaseKGELightning method), 12
on_train_epoch_end() (dicee.models.BaseKGELightning method), 47
OnevsAllDataset (class in dicee), 167
OnevsAllDataset (class in dicee.dataset_classes), 112
optim (dicee.config.Namespace attribute), 108
Р
p (dicee.config.Namespace attribute), 109
parameters () (dicee.abstracts.BaseInteractiveKGE method), 98
path_single_kg (dicee.config.Namespace attribute), 108
path_to_store_single_run (dicee.config.Namespace attribute), 108
Perturb (class in dicee.callbacks), 107
poly_NN() (dicee.LFMult method), 152
```

```
poly NN() (dicee.models.function space.LFMult method), 30
poly_NN() (dicee.models.LFMult method), 79
polynomial() (dicee.LFMult method), 153
polynomial() (dicee.models.function_space.LFMult method), 31
polynomial() (dicee.models.LFMult method), 80
pop() (dicee.LFMult method), 153
pop() (dicee.models.function_space.LFMult method), 31
pop() (dicee.models.LFMult method), 80
predict() (dicee.KGE method), 161
predict() (dicee.knowledge_graph_embeddings.KGE method), 125
predict_dataloader() (dicee.models.base_model.BaseKGELightning method), 14
predict_dataloader() (dicee.models.BaseKGELightning method), 49
predict_missing_head_entity() (dicee.KGE method), 160
predict_missing_head_entity() (dicee.knowledge_graph_embeddings.KGE method), 124
predict_missing_relations() (dicee.KGE method), 161
predict_missing_relations() (dicee.knowledge_graph_embeddings.KGE method), 125
predict_missing_tail_entity() (dicee.KGE method), 161
predict_missing_tail_entity() (dicee.knowledge_graph_embeddings.KGE method), 125
predict_topk() (dicee.KGE method), 162
predict_topk() (dicee.knowledge_graph_embeddings.KGE method), 126
prepare_data() (dicee.CVDataModule method), 173
prepare_data() (dicee.dataset_classes.CVDataModule method), 118
preprocess_with_byte_pair_encoding() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 86
preprocess_with_byte_pair_encoding() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 82
preprocess_with_byte_pair_encoding_with_padding() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 86
preprocess_with_byte_pair_encoding_with_padding() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method),
          82
preprocess_with_pandas() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 86
preprocess_with_pandas() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 82
preprocess_with_polars() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 86
preprocess_with_polars() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 82
preprocesses_input_args() (in module dicee.static_preprocess_funcs), 134
PreprocessKG (class in dicee.read_preprocess_save_load_kg), 86
PreprocessKG (class in dicee.read_preprocess_save_load_kg.preprocess), 82
print_peak_memory() (in module dicee.trainer.torch_trainer_ddp), 92
PrintCallback (class in dicee.callbacks), 102
PseudoLabellingCallback (class in dicee.callbacks), 104
Pyke (class in dicee), 138
Pyke (class in dicee.models), 55
Pyke (class in dicee.models.real), 38
pykeen_model_kwargs (dicee.config.Namespace attribute), 109
PykeenKGE (class in dicee), 153
PykeenKGE (class in dicee.models), 76
PykeenKGE (class in dicee.models.pykeen_models), 34
Q
q (dicee.config.Namespace attribute), 109
OMult (class in dicee), 149
QMult (class in dicee.models), 61
QMult (class in dicee.models.quaternion), 35
quaternion_mul() (in module dicee.models), 59
quaternion_mul() (in module dicee.models.static_funcs), 39
quaternion_mul_with_unit_norm() (in module dicee.models), 61
quaternion_mul_with_unit_norm() (in module dicee.models.quaternion), 35
quaternion_multiplication_followed_by_inner_product() (dicee.models.QMult method), 61
quaternion_multiplication_followed_by_inner_product() (dicee.models.quaternion.QMult method), 36
\verb"quaternion_multiplication_followed_by_inner_product() (\textit{dicee.QMult method}), 150
quaternion_normalizer() (dicee.models.QMult static method), 62
quaternion_normalizer() (dicee.models.quaternion.QMult static method), 36
quaternion_normalizer() (dicee.QMult static method), 150
QueryGenerator (class in dicee), 173
QueryGenerator (class in dicee.query_generator), 128
R
random_prediction() (in module dicee), 158
random_prediction() (in module dicee.static_funcs), 132
random_seed (dicee.config.Namespace attribute), 109
```

```
read_from_disk() (in module dicee.read_preprocess_save_load_kg.util), 84
read_from_triple_store() (in module dicee.read_preprocess_save_load_kg.util), 85
read_only_few (dicee.config.Namespace attribute), 109
read_or_load_kg() (dicee.Execute method), 164
read_or_load_kg() (dicee.executer.Execute method), 121
read_or_load_kg() (in module dicee), 158
read_or_load_kg() (in module dicee.static_funcs), 132
read_preprocess_index_serialize_data() (dicee. Execute method), 164
read_preprocess_index_serialize_data() (dicee.executer.Execute method), 121
read_with_pandas() (in module dicee.read_preprocess_save_load_kg.util), 84
read_with_polars() (in module dicee.read_preprocess_save_load_kg.util), 84
ReadFromDisk (class in dicee.read_preprocess_save_load_kg), 87
ReadFromDisk (class in dicee.read_preprocess_save_load_kg.read_from_disk), 83
relations_str (dicee.knowledge_graph.KG property), 123
reload_dataset() (in module dicee), 165
reload_dataset() (in module dicee.dataset_classes), 111
{\tt remove\_triples\_from\_train\_with\_condition()} \ (\textit{dicee.read\_preprocess\_save\_load\_kg.PreprocessKG method}), 86
remove_triples_from_train_with_condition() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 82
residual_convolution() (dicee.AConEx method), 147
residual_convolution() (dicee.AConvO method), 147
residual_convolution() (dicee.AConvQ method), 148
residual_convolution() (dicee.ConEx method), 149
residual_convolution() (dicee.ConvO method), 149
residual_convolution() (dicee.ConvQ method), 148
residual_convolution() (dicee.models.AConEx method), 57
residual_convolution() (dicee.models.AConvO method), 67
residual_convolution() (dicee.models.AConvQ method), 63
residual_convolution() (dicee.models.complex.AConEx method), 26
\verb"residual_convolution" () \textit{ (dicee.models.complex.ConEx method)}, 26
residual_convolution() (dicee.models.ConEx method), 57
residual_convolution() (dicee.models.ConvO method), 67
residual_convolution() (dicee.models.ConvQ method), 62
residual_convolution() (dicee.models.octonion.AConvO method), 34
residual_convolution() (dicee.models.octonion.ConvO method), 33
residual_convolution() (dicee.models.quaternion.AConvQ method), 37
residual_convolution() (dicee.models.quaternion.ConvQ method), 37
retrieve embeddings () (in module dicee.scripts.serve), 88
return_multi_hop_query_results() (dicee.KGE method), 163
\verb|return_multi_hop_query_results|()| \textit{(dicee.knowledge\_graph\_embeddings.KGE method)}, 127|
root () (in module dicee.scripts.serve), 88
S
sample_entity() (dicee.abstracts.BaseInteractiveKGE method), 97
sample_relation() (dicee.abstracts.BaseInteractiveKGE method), 97
sample_triples_ratio (dicee.config.Namespace attribute), 109
sanity_checking_with_arguments() (in module dicee.sanity_checkers), 129
save () (dicee.abstracts.BaseInteractiveKGE method), 97
save() (dicee.read_preprocess_save_load_kg.LoadSaveToDisk method), 86
save() (dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk method), 83
save_checkpoint() (dicee.abstracts.AbstractTrainer static method), 95
save_checkpoint_model() (in module dicee), 158
save_checkpoint_model() (in module dicee.static_funcs), 131
save_embeddings() (in module dicee), 158
save_embeddings() (in module dicee.static_funcs), 132
save_embeddings_as_csv (dicee.config.Namespace attribute), 108
save_experiment() (dicee.analyse_experiments.Experiment method), 101
save_model_at_every_epoch (dicee.config.Namespace attribute), 109
save_numpy_ndarray() (in module dicee), 158
save_numpy_ndarray() (in module dicee.read_preprocess_save_load_kg.util), 85
save_numpy_ndarray() (in module dicee.static_funcs), 131
save_pickle() (in module dicee), 157
save_pickle() (in module dicee.read_preprocess_save_load_kg.util), 85
save_pickle() (in module dicee.static_funcs), 131
save_queries() (dicee.query_generator.QueryGenerator method), 129
save\_queries() (dicee.QueryGenerator method), 174
save_queries_and_answers() (dicee.query_generator.QueryGenerator static method), 129
save_queries_and_answers() (dicee.QueryGenerator static method), 174
```

```
save trained model () (dicee. Execute method), 165
save_trained_model() (dicee.executer.Execute method), 122
scalar_batch_NN() (dicee.LFMult method), 152
scalar_batch_NN() (dicee.models.function_space.LFMult method), 30
scalar_batch_NN() (dicee.models.LFMult method), 79
score() (dicee.CMult method), 138
score() (dicee.ComplEx static method), 147
score () (dicee.DistMult method), 139
score () (dicee. Keci method), 141
score() (dicee.models.clifford.CMult method), 19
score() (dicee.models.clifford.Keci method), 22
score() (dicee.models.CMult method), 71
score() (dicee.models.ComplEx static method), 58
score () (dicee.models.complex.ComplEx static method), 27
score() (dicee.models.DistMult method), 55
score() (dicee.models.Keci method), 70
score() (dicee.models.octonion.OMult method), 32
score () (dicee.models.OMult method), 66
score () (dicee.models.QMult method), 62
score () (dicee.models.quaternion.QMult method), 36
score() (dicee.models.real.DistMult method), 38
score() (dicee.models.real.TransE method), 38
score() (dicee.models.TransE method), 55
score() (dicee.OMult method), 152
score () (dicee.QMult method), 151
score() (dicee. TransE method), 142
scoring_technique (dicee.config.Namespace attribute), 109
search() (dicee.scripts.serve.NeuralSearcher method), 89
search_embeddings() (in module dicee.scripts.serve), 88
select_model() (in module dicee), 157
select_model() (in module dicee.static_funcs), 131
sequential_vocabulary_construction() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 86
sequential_vocabulary_construction() (dicee.read_preprocess_save_load_kg.preprocess.Preprocess.PreprocessKG method), 82
set_global_seed() (dicee.query_generator.QueryGenerator method), 128
set_global_seed() (dicee.QueryGenerator method), 174
\verb|set_model_eval_mode()| (dicee.abstracts.BaseInteractiveKGE method), 97|
set model train mode() (dicee.abstracts.BaseInteractiveKGE method), 96
setup() (dicee.CVDataModule method), 171
setup() (dicee.dataset_classes.CVDataModule method), 116
Shallom (class in dicee), 152
Shallom (class in dicee.models), 55
Shallom (class in dicee.models.real), 38
single_hop_query_answering() (dicee.KGE method), 163
\verb|single_hop_query_answering()| \textit{(dicee.knowledge\_graph\_embeddings.KGE method)}, 127
sparql_endpoint (dicee.config.Namespace attribute), 108
start() (dicee.DICE_Trainer method), 159
start () (dicee. Execute method), 165
start() (dicee.executer.Execute method), 122
start() (dicee.read_preprocess_save_load_kg.PreprocessKG method), 86
start() (dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method), 82
start() (dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method), 83
start() (dicee.read_preprocess_save_load_kg.ReadFromDisk method), 87
start() (dicee.trainer.DICE_Trainer method), 93
start() (dicee.trainer.dice_trainer.DICE_Trainer method), 90
storage_path (dicee.config.Namespace attribute), 108
store() (in module dicee), 158
store() (in module dicee.static_funcs), 132
\verb|store_ensemble()| \textit{(dicee.abstracts.AbstractPPECallback method)}, 100
swa (dicee.config.Namespace attribute), 110
T() (dicee.DualE method), 146
T() (dicee.models.DualE method), 81
T () (dicee.models.dualE.DualE method), 29
t_conorm() (dicee.KGE method), 163
t_conorm() (dicee.knowledge_graph_embeddings.KGE method), 127
t_norm() (dicee.KGE method), 163
```

```
t norm() (dicee.knowledge graph embeddings.KGE method), 127
tensor_t_norm() (dicee.KGE method), 163
tensor_t_norm() (dicee.knowledge_graph_embeddings.KGE method), 127
test_dataloader() (dicee.models.base_model.BaseKGELightning method), 13
test_dataloader() (dicee.models.BaseKGELightning method), 48
test_epoch_end() (dicee.models.base_model.BaseKGELightning method), 12
test_epoch_end() (dicee.models.BaseKGELightning method), 48
timeit() (in module dicee), 157, 165
timeit() (in module dicee.read_preprocess_save_load_kg.util), 84
timeit() (in module dicee.static_funcs), 131
timeit() (in module dicee.static_preprocess_funcs), 134
to () (dicee.KGE method), 160
to () (dicee.knowledge_graph_embeddings.KGE method), 124
to_df() (dicee.analyse_experiments.Experiment method), 101
TorchDDPTrainer (class in dicee.trainer.torch_trainer_ddp), 92
TorchTrainer (class in dicee.trainer.torch_trainer), 91
train() (dicee.KGE method), 164
train() (dicee.knowledge_graph_embeddings.KGE method), 128
train() (dicee.trainer.torch_trainer_ddp.DDPTrainer method), 92
train() (dicee.trainer.torch_trainer_ddp.NodeTrainer method), 92
train_dataloader() (dicee.CVDataModule method), 171
train_dataloader() (dicee.dataset_classes.CVDataModule method), 116
train_dataloader() (dicee.models.base_model.BaseKGELightning method), 14
train_dataloader() (dicee.models.BaseKGELightning method), 49
train_k_vs_all() (dicee.KGE method), 164
train_k_vs_all() (dicee.knowledge_graph_embeddings.KGE method), 128
train_triples() (dicee.KGE method), 164
train_triples() (dicee.knowledge_graph_embeddings.KGE method), 128
trainer (dicee.config.Namespace attribute), 109
training_step() (dicee.BytE method), 155
training_step() (dicee.models.base_model.BaseKGELightning method), 11
training_step() (dicee.models.BaseKGELightning method), 46
\verb|training_step()| \textit{(dicee.models.transformers.BytE method)}, 40
TransE (class in dicee), 142
TransE (class in dicee.models), 55
TransE (class in dicee, models, real), 38
transfer batch to device() (dicee.CVDataModule method), 172
transfer_batch_to_device() (dicee.dataset_classes.CVDataModule method), 117
trapezoid() (dicee.models.FMult2 method), 79
trapezoid() (dicee.models.function_space.FMult2 method), 30
tri_score() (dicee.LFMult method), 152
tri_score() (dicee.models.function_space.LFMult method), 31
tri_score() (dicee.models.function_space.LFMult1 method), 30
tri_score() (dicee.models.LFMult method), 79
tri_score() (dicee.models.LFMult1 method), 79
triple_score() (dicee.KGE method), 162
triple_score() (dicee.knowledge_graph_embeddings.KGE method), 126
TriplePredictionDataset (class in dicee), 170
TriplePredictionDataset (class in dicee.dataset_classes), 115
tuple2list() (dicee.query_generator.QueryGenerator method), 128
tuple2list() (dicee.QueryGenerator method), 174
U
unmap() (dicee.query_generator.QueryGenerator method), 129
unmap() (dicee.QueryGenerator method), 174
unmap_query() (dicee.query_generator.QueryGenerator method), 129
unmap_query() (dicee.QueryGenerator method), 174
val_dataloader() (dicee.models.base_model.BaseKGELightning method), 13
val_dataloader() (dicee.models.BaseKGELightning method), 48
validate_knowledge_graph() (in module dicee.sanity_checkers), 129
vocab_preparation() (dicee.evaluator.Evaluator method), 120
vocab_size (dicee.models.transformers.GPTConfig attribute), 43
vocab_to_parquet() (in module dicee), 158
vocab_to_parquet() (in module dicee.static_funcs), 132
vtp_score() (dicee.LFMult method), 153
```

```
vtp_score() (dicee.models.function_space.LFMult method), 31 vtp_score() (dicee.models.function_space.LFMult1 method), 30 vtp_score() (dicee.models.LFMult method), 80 vtp_score() (dicee.models.LFMult1 method), 79
```

W

```
weight_decay (dicee.config.Namespace attribute), 109
write_links() (dicee.query_generator.QueryGenerator method), 128
write_links() (dicee.QueryGenerator method), 174
write_report() (dicee.Execute method), 165
write_report() (dicee.executer.Execute method), 122
```