
DICE Embeddings

Release 0.1.3.2

Caglar Demir

Mar 05, 2024

Contents:

1 DICE Embeddings: Hardware-agnostic Framework for Large-scale Knowledge Graph Embeddings:	1
1.1 <code>dicee</code>	1
1.2 Indices and tables	175
Python Module Index	176
Index	177

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

1.1 `dicee`

Subpackages

`dicee.models`

Submodules

`dicee.models.base_model`

Module Contents

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

Classes

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

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

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

mem_of_model () → Dict

Size of model in MB and number of params

training_step (*batch, batch_idx=None*)

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

Args:

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)

Return:

- **Tensor** - The loss tensor
- **dict** - A dictionary which can include any keys, but must include the key 'loss' in the case of automatic optimization.

- None - In automatic optimization, this will skip to the next batch (but is not supported for multi-GPU, TPU, or DeepSpeed). For manual optimization, this has no special meaning, as returning the loss is not required.

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

Example:

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

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

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

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

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

Note:

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

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

Parameters

yhat_batch y_batch

Returns

on_train_epoch_end (*args, **kwargs)

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

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

```
class MyLightningModule(L.LightningModule):
    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
self.training_step_outputs = []

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

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

test_epoch_end (outputs: List[Any])

test_dataloader () → None

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning: do not assign state in `prepare_data`

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

Note:

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

Note:

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

val_dataloader () → None

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

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

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

- `fit()`
- `validate()`

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

Note:

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

Note:

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

`predict_dataloader()` → None

An iterable or collection of iterables specifying prediction samples.

For more information about multiple dataloaders, see this section.

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

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

Note:

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

Return:

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

`train_dataloader()` → None

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

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

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning: do not assign state in `prepare_data`

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

Note:

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

`configure_optimizers` (*parameters=None*)

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

Return:

Any of these 6 options.

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

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

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

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

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

Note:

Some things to know:

- Lightning calls `.backward()` and `.step()` automatically in case of automatic optimization.
- If a learning rate scheduler is specified in `configure_optimizers()` with key "interval" (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()`

method automatically in case of automatic optimization.

- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizer.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, you will have to switch to ‘manual optimization’ mode and step them yourself.
- If you need to control how often the optimizer steps, override the `optimizer_step()` hook.

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

forward_byte_pair_encoded_k_vs_all (*x: torch.LongTensor*)

Parameters

$x : B \times 2 \times T$

Returns

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

Returns

forward_triples (x : *torch.LongTensor*) \rightarrow *torch.Tensor*

Parameters

x

Returns

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

x shape (b,3,t)

Returns

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

Parameters

x : B x 2 x T

Returns

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

Returns

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

static forward (x)

`dicee.models.clifford`

Module Contents

Classes

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

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

Bases: `dicee.models.base_model.BaseKGE`

Cl_(0,0) => Real Numbers

Cl_(0,1) =>

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

multiplication is isomorphic to the product of two complex numbers

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

Cl_(2,0) =>

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

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

Cl_(0,2) => Quaternions

clifford_mul (*x: torch.FloatTensor, y: torch.FloatTensor, p: int, q: int*) → tuple

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

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

e_j

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

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

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

score (*head_ent_emb, rel_ent_emb, tail_ent_emb*)

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

Compute batch triple scores

Parameter

x: torch.LongTensor with shape n by 3

Returns

torch.LongTensor with shape n

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

Compute batch KvsAll triple scores

Parameter

x: torch.LongTensor with shape n by 3

Returns

torch.LongTensor with shape n

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

compute_sigma_pp (*hp, rp*)

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

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

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

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

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

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

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

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

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

compute_sigma_qq (*hq, rq*)

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

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

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

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

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

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

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

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

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

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

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

for j in range(q):

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

print(sigma_pq.shape)

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

Multiplying a base vector with its scalar coefficient

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

Compute our CL multiplication

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

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

e_j

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

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

- (2) $\sigma_p = \sum_{i=1}^p (h_{0r_i} + h_{ir_0}) e_i$
- (3) $\sigma_q = \sum_{j=p+1}^{p+q} (h_{0r_j} + h_{jr_0}) e_j$
- (4) $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_{ir_k} - h_{kr_i}) e_i e_k$
- (5) $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_{jr_k} - h_{kr_j}) e_j e_k$
- (6) $\sigma_{pq} = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_{ir_j} - h_{jr_i}) e_i e_j$

construct_cl_multivector (*x: torch.FloatTensor, r: int, p: int, q: int*)
 \rightarrow 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 $Cl_{\{p,q\}}(\mathbb{R}^d)$.
- (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 Returns ——— torch.FloatTensor with (n, **IEI**) shape

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 $Cl_{\{p,q\}}(\mathbb{R}^d)$.
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

Parameter

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

Returns

torch.FloatTensor with (n, IEI) shape

score (*h, r, t*)

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

Parameter

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

Returns

torch.FloatTensor with (n) shape

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

Bases: *Keci*

Without learning dimension scaling

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

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

Parameter

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

Returns

torch.FloatTensor with (n) shape

cl_pqr (*a*)

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, 1) $s_0 = h_{Or_Ot_0}$ 2) $s_1 = \sum_{i=1}^{p-1} \{p\} h_{ir_it_0}$ 3) $s_2 = \sum_{j=p+1}^{p+q-1} \{p+q\} h_{jr_jt_0}$ 4) $s_3 = \sum_{i=1}^{p-1} \{q\} (h_{Or_it_i} + h_{ir_Ot_i})$ 5) $s_4 = \sum_{i=p+1}^{p+q-1} \{p+q\} (h_{Or_it_i} + h_{ir_Ot_i})$ 5) $s_5 = \sum_{i=p+q+1}^{p+q+r-1} \{p+q+r\} (h_{Or_it_i} + h_{ir_Ot_i})$

and return:

$\ast) \sigma_{0t} = \sigma_0 \cdot t_0 = s_0 + s_1 - s_2 \ast) s_3, s_4 \text{ and } s_5$

compute_sigmas_multivect (*list_h_emb, list_r_emb*)

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

For same bases vectors interaction we have

- 1) $\sigma_{pp} = \sum_{i=1}^{p-1} \{p-1\} \sum_{i'=i+1}^p \{p\} (h_{ir_i'} - h_{i'} r_i)$ (models the interactions between e_i and $e_{i'}$ for $1 \leq i, i' \leq p$)
- 2) $\sigma_{qq} = \sum_{j=p+1}^{p+q-1} \{p+q-1\} \sum_{j'=j+1}^{p+q} \{p+q\} (h_{jr_j'} - h_{j'} r_j)$ (models the interactions between e_j and $e_{j'}$ for $p+1 \leq j, j' \leq p+q$)
- 3) $\sigma_{rr} = \sum_{k=p+q+1}^{p+q+r-1} \{p+q+r-1\} \sum_{k'=k+1}^{p+q+r} \{p\} (h_{kr_k'} - h_{k'} r_k)$ (models the interactions between e_k and $e_{k'}$ for $p+q+1 \leq k, k' \leq p+q+r$)

For different base vector interactions, we have

- 4) $\sigma_{pq} = \sum_{i=1}^{p-1} \{p\} \sum_{j=p+1}^{p+q-1} \{p+q\} (h_{ir_j} - h_{jr_i})$ (interactionsn between e_i and e_j for $1 \leq i \leq p$ and $p+1 \leq j \leq p+q$)
- 5) $\sigma_{pr} = \sum_{i=1}^{p-1} \{p\} \sum_{k=p+q+1}^{p+q+r-1} \{p+q+r\} (h_{ir_k} - h_{kr_i})$ (interactionsn between e_i and e_k for $1 \leq i \leq p$ and $p+q+1 \leq k \leq p+q+r$)
- 6) $\sigma_{qr} = \sum_{j=p+1}^{p+q-1} \{p+q\} \sum_{k=p+q+1}^{p+q+r-1} \{p+q+r\} (h_{jr_k} - h_{kr_j})$ (interactionsn between e_j and e_k for $p+1 \leq j \leq p+q$ and $p+q+1 \leq k \leq p+q+r$)

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

Kvsall training

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

forward_k_vs_with_explicit and this functions are identical Parameter ——— *x*: torch.LongTensor with (n,2) shape Returns ——— torch.FloatTensor with (n, **IE**) shape

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

Multiplying a base vector with its scalar coefficient

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

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

Parameter

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

Returns

a0: torch.FloatTensor *ap*: torch.FloatTensor *aq*: torch.FloatTensor *ar*: torch.FloatTensor

compute_sigma_pp (*hp, rp*)

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

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

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

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

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

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

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

*e*₂*e*₁, *e*₂*e*₂, *e*₂*e*₃, *e*₃*e*₁, *e*₃*e*₂, *e*₃*e*₃

Then select the triangular matrix without diagonals: *e*₁*e*₂, *e*₁*e*₃, *e*₂*e*₃.

compute_sigma_qq (*hq, rq*)

Compute $\sigma_{\{q,q\}}^{*} = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (x_{jy_{j'}} - x_{j'y_j})$ Eq. 16
 $\sigma_{\{q\}}$ captures the interactions between along *q* bases For instance, let *q* *e*₁, *e*₂, *e*₃, we compute interactions between *e*₁ *e*₂, *e*₁ *e*₃, and *e*₂ *e*₃ This can be implemented with a nested two for loops

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

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

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

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


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

$e2e1$, $e2e2$, $e2e3$, $e3e1$, $e3e2$, $e3e3$

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

```
compute_sigma_rr (hk, rk)
    sigma_{r,r}^* = sum_{k=p+q+1}^{p+q+r-1} sum_{k'=k+1}^p (x_ky_{k'} - x_{k'}y_k)

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)

compute_sigma_pr (*, hp, hk, rp, 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)

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`

Module Contents

Classes

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

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

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

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

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

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

Parameters

x

Returns

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]*) → torch.FloatTensor

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

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

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

Parameters

x

Returns

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

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

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

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

```

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

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

Variables

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

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

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

Parameters

emb_h emb_r emb_E

Returns

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

`dicee.models.function_space`

Module Contents

Classes

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

class `dicee.models.function_space.FMult` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Learning Knowledge Neural Graphs

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

chain_func (*weights, x: torch.FloatTensor*)

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

Parameters

x

Returns

class `dicee.models.function_space.GFMult` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Learning Knowledge Neural Graphs

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

chain_func (*weights, x: torch.FloatTensor*)

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

Parameters

x

Returns

```
class dicee.models.function_space.FMult2(args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    build_func (Vec)
    build_chain_funcs (list_Vec)
    compute_func (W, b, x) → torch.FloatTensor
    function (list_W, list_b)
    trapezoid (list_W, list_b)
    forward_triples (idx_triple: torch.Tensor) → torch.Tensor
```

Parameters

x

Returns

```
class dicee.models.function_space.LFMult1(args)
    Bases: dicee.models.base_model.BaseKGE
    Embedding with trigonometric functions. We represent all entities and relations in the complex number space as:
     $f(x) = \sum_{k=0}^{d-1} w_k e^{kix}$ . and use the three differents scoring function as in the paper to evaluate
    the score
    forward_triples (idx_triple)
```

Parameters

x

Returns

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

class dicee.models.function_space.LFMult(args)
    Bases: dicee.models.base_model.BaseKGE
    Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
     $f(x) = \sum_{i=0}^{d-1} a_i x^i$  and use the three differents scoring function as in the paper to evaluate the score.
    We also consider combining with Neural Networks.
    forward_triples (idx_triple)
```

Parameters

x

Returns

construct_multi_coeff (x)

poly_NN ($x, \text{coefh}, \text{coefr}, \text{coeft}$)

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

linear (x, w, b)

scalar_batch_NN (a, b, c)

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

tri_score ($\text{coeff}_h, \text{coeff}_r, \text{coeff}_t$)

this part implement the trilinear scoring techniques:

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

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

vtp_score (h, r, t)

this part implement the vector triple product scoring techniques:

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

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

comp_func (h, r, t)

this part implement the function composition scoring techniques: i.e. $\text{score} = \langle h, r, t \rangle$

polynomial ($\text{coeff}, x, \text{degree}$)

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

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

pop ($\text{coeff}, x, \text{degree}$)

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

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

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

`dicee.models.octonion`

Module Contents

Classes

<i>OMult</i>	Base class for all neural network modules.
<i>ConvO</i>	Base class for all neural network modules.
<i>ACConvO</i>	Additive Convolutional Octonion Knowledge Graph Embeddings

Functions

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

`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.

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

Variables

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

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

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

k_vs_all_score (*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

forward_k_vs_all (*x*)

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

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

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

residual_convolution (*O_1, O_2*)

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

Parameters

x

Returns

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) → torch.Tensor

Parameters

x

Returns

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

Module Contents

Classes

PykeenKGE

A class for using knowledge graph embedding models implemented in Pykeen

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

Bases: *dicee.models.base_model.BaseKGE*

A class for using knowledge graph embedding models implemented in Pykeen

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

```

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

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

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

    else:
        t = self.entity_embeddings.weight

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

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

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

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

abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)

```

dicee.models.quaternion

Module Contents

Classes

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

Functions

```

quaternion_mul_with_unit_norm(*, Q_1,
Q_2)

```

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

class dicee.models.quaternion.QMult (args)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

quaternion_multiplication_followed_by_inner_product (*h, r, t*)

Parameters

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

Returns

Triple scores.

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

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

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

Parameters

x – The vector.

Returns

The normalized vector.

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

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

Parameters

bpe_head_ent_emb *bpe_rel_ent_emb* *E*

Returns

forward_k_vs_all (*x*)

Parameters

x

Returns

forward_k_vs_sample (*x*, *target_entity_idx*)

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

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

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings

residual_convolution (*Q_1*, *Q_2*)

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

Parameters

x

Returns

forward_k_vs_all (*x*: torch.Tensor)

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

```
class dicee.models.quaternion.AConvQ(args)
    Bases: dicee.models.base_model.BaseKGE
    Additive Convolutional Quaternion Knowledge Graph Embeddings
    residual_convolution(Q_1, Q_2)
    forward_triples(indexed_triple: torch.Tensor) → torch.Tensor
```

Parameters

x

Returns

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`

Module Contents

Classes

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

```
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

Returns

forward_k_vs_all (*x*: torch.LongTensor)

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

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

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

Bases: *dicee.models.base_model.BaseKGE*

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

score (*head_ent_emb*, *rel_ent_emb*, *tail_ent_emb*)

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

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

Bases: *dicee.models.base_model.BaseKGE*

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

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

Returns

forward_k_vs_all (*x*) → torch.FloatTensor

forward_triples (*x*) → torch.FloatTensor

Parameters

x –

Returns

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

Bases: *dicee.models.base_model.BaseKGE*

A Physical Embedding Model for Knowledge Graphs

forward_triples (*x*: torch.LongTensor)

Parameters

x

Returns

`dicee.models.static_funcs`

Module Contents

Functions

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

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

`dicee.models.transformers`

Module Contents

Classes

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

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

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

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

```

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

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

Variables

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

loss_function (*yhat_batch, y_batch*)

Parameters

yhat_batch y_batch

Returns

forward (*x: torch.LongTensor*)

Parameters

x: B by T tensor

Returns

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.

Args:

`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)

Return:

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

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

Example:

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

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

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

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

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

Note:

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

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

Bases: `torch.nn.Module`

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

forward (*input*)

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

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

forward (*x*)

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

Bases: `torch.nn.Module`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

forward (*x*)

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

forward (*x*)

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

block_size: int = 1024

vocab_size: int = 50304

n_layer: int = 12

n_head: int = 12

n_embd: int = 768

dropout: float = 0.0

bias: bool = False

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

get_num_params (*non_embedding=True*)

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

forward (*idx, targets=None*)

crop_block_size (*block_size*)

classmethod from_pretrained (*model_type, override_args=None*)

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

estimate_mfu (*fwd_bwd_per_iter, dt*)

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

Package Contents

Classes

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

Functions

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

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

Bases: lightning.LightningModule

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

mem_of_model () → Dict

Size of model in MB and number of params

training_step (*batch, batch_idx=None*)

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

Args:

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)

Return:

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

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

Example:

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

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

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

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

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

Note:

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

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

Parameters

yhat_batch y_batch

Returns

on_train_epoch_end (*args, **kwargs)

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

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

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

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

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

test_epoch_end (outputs: List[Any])

test_dataloader () → None

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning: do not assign state in `prepare_data`

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

Note:

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

Note:

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

val_dataloader() → None

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

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

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

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

Note:

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

Note:

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

predict_dataloader() → None

An iterable or collection of iterables specifying prediction samples.

For more information about multiple dataloaders, see this section.

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

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

Note:

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

Return:

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

train_dataloader() → None

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

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

For data processing use the following pattern:

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

However, the above are only necessary for distributed processing.

Warning: do not assign state in `prepare_data`

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

Note:

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

`configure_optimizers` (*parameters=None*)

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

Return:

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
```

(continues on next page)

```
"name": None,
}
```

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

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

Note:

Some things to know:

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

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

forward_byte_pair_encoded_k_vs_all (*x: torch.LongTensor*)

Parameters

x : B x 2 x T

Returns

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

Returns

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

Parameters

x

Returns

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

x shape (b,3,t)

Returns

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

Parameters

x : B x 2 x T

Returns

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

Returns

class dicee.models.IdentityClass (args=None)

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

```
static forward(x)
```

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

forward_byte_pair_encoded_k_vs_all (*x: torch.LongTensor*)

Parameters

x: B x 2 x T

Returns

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*

Returns

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

Parameters

x

Returns

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

x shape (b,3,t)

Returns

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

Parameters

$x : B \times 2 \times T$

Returns

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

Returns

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

Returns

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() \rightarrow Tuple[numpy.ndarray, None]

Returns

forward_k_vs_all (*x*) → torch.FloatTensor

forward_triples (*x*) → torch.FloatTensor

Parameters

x –

Returns

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

Bases: *dicee.models.base_model.BaseKGE*

A Physical Embedding Model for Knowledge Graphs

forward_triples (*x: torch.LongTensor*)

Parameters

x

Returns

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

forward_byte_pair_encoded_k_vs_all (*x: torch.LongTensor*)

Parameters

x : B x 2 x T

Returns

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

Returns

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

Parameters

x

Returns

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

x shape (b,3,t)

Returns

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

Parameters

x : B x 2 x T

Returns

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

Returns

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

Bases: *dicee.models.base_model.BaseKGE*

Convolutional ComplEx Knowledge Graph Embeddings

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

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

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

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

Parameters

x

Returns

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

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional ComplEx Knowledge Graph Embeddings

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

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

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

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

Parameters

x

Returns

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

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

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

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

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

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

Parameters

emb_h emb_r emb_E

Returns

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

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

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

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

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

Parameters

$x : B \times 2 \times T$

Returns

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

Returns

forward_triples (x : *torch.LongTensor*) \rightarrow torch.Tensor

Parameters

x

Returns

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

x shape (b,3,t)

Returns

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

Parameters

x : B x 2 x T

Returns

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

Returns

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

Bases: torch.nn.Module

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

```
static forward(x)
```

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

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

quaternion_multiplication_followed_by_inner_product (*h, r, t*)

Parameters

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

Returns

Triple scores.

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

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

Absolute value of a quaternion

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

L2 norm of quaternion vector:

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

Parameters

\mathbf{x} – The vector.

Returns

The normalized vector.

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

k_vs_all_score (*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

Parameters

bpe_head_ent_emb bpe_rel_ent_emb E

Returns

forward_k_vs_all (*x*)

Parameters

x

Returns

forward_k_vs_sample (*x, target_entity_idx*)

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

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

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings

residual_convolution (*Q_1, Q_2*)

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

Parameters

x

Returns

forward_k_vs_all (*x*: torch.Tensor)

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

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Quaternion Knowledge Graph Embeddings

residual_convolution (*Q_1, Q_2*)

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

Parameters

x

Returns

forward_k_vs_all (*x*: torch.Tensor)

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

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

Bases: *BaseKGELightning*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

forward_byte_pair_encoded_k_vs_all (*x: torch.LongTensor*)

Parameters

x : B x 2 x T

Returns

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

Returns

forward_triples (*x: torch.LongTensor*) \rightarrow torch.Tensor

Parameters

x

Returns

```
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

x shape (b,3,t)

Returns

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

Parameters

x : B x 2 x T

Returns

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

Returns

```
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)
```

(continues on next page)

(continued from previous page)

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

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

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

Variables

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

static forward (*x*)

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

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

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

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

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

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

```
k_vs_all_score (bpe_head_ent_emb, bpe_rel_ent_emb, E)
```

```
forward_k_vs_all (x)
```

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

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

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

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

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

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

Variables

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

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

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

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

Parameters

x

Returns

forward_k_vs_all (x: torch.Tensor)

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

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

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Octonion Knowledge Graph Embeddings

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

residual_convolution (O_1, O_2)

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

Parameters

x

Returns

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):
```

(continues on next 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.

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

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

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

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

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

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

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

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

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

`compute_sigma_qq` (*hq, rq*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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


apply_coefficients (*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

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

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

e_j

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

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

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

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

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

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

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

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

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

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

Parameter

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

Returns

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

forward_k_vs_with_explicit (*x: torch.Tensor*)

k_vs_all_score (*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

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

Kvsall training

(1) Retrieve real-valued embedding vectors for heads and relations \mathbb{R}^d .

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

(3) Perform Cl multiplication

(4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this functions are identical Parameter ——— x: torch.LongTensor with (n,2) shape Returns ——— torch.FloatTensor with (n, **IEI**) shape

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

Kvsall training

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

Parameter

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

Returns

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

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

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

Parameter

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

Returns

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

$Cl_{(0,1)} \Rightarrow$

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

multiplication is isomorphic to the product of two complex numbers

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

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

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

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

$Cl_{\{0,2\}} \Rightarrow \text{Quaternions}$

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

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

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

e_j

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

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

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

score (*head_ent_emb, rel_ent_emb, tail_ent_emb*)

forward_triples (*x: torch.LongTensor*) \rightarrow torch.FloatTensor

Compute batch triple scores

Parameter

x: torch.LongTensor with shape n by 3

Returns

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

Returns

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):
```

(continues on next page)

```

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

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

```

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

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

Variables

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

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

Parameter

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

Returns

torch.FloatTensor with (n) shape

cl_pqr (*a*)

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, 1) $s_0 = h_{0r_{0t_0}}$ 2) $s_1 = \sum_{i=1}^{p-1} h_{ir_{it_0}}$ 3) $s_2 = \sum_{j=p+1}^{p+q} h_{jr_{jt_0}}$ 4) $s_3 = \sum_{i=1}^{p-1} \sum_{j=p+1}^{p+q} (h_{0r_{it_i}} + h_{ir_{0t_i}})$ 5) $s_4 = \sum_{i=p+1}^{p+q} \sum_{j=p+q+1}^{p+q+r} (h_{0r_{it_i}} + h_{ir_{0t_i}})$ 5) $s_5 = \sum_{i=p+q+1}^{p+q+r} (h_{0r_{it_i}} + h_{ir_{0t_i}})$

and return:

*****) $\text{sigma}_{0t} = \text{sigma}_0 \cdot t_0 = s_0 + s_1 - s_2$ *****) s_3, s_4 and s_5

compute_sigmas_multivect (*list_h_emb, list_r_emb*)

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

For same bases vectors interaction we have

1) $\text{sigma}_{pp} = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (h_{ir_{i'}} - h_{i'r_i})$ (models the interactions between e_i and $e_{i'}$ for $1 \leq i, i' \leq p$)

- 2) $\text{sigma_qq} = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_{jr_{j'}} - h_{j'})$ (models the interactions between e_j and $e_{j'}$ for $p+1 \leq j, j' \leq p+q$)
- 3) $\text{sigma_rr} = \sum_{k=p+q+1}^{p+q+r-1} \sum_{k'=k+1}^{p+q+r} (h_{kr_{k'}} - h_{k'})$ (models the interactions between e_k and $e_{k'}$ for $p+q+1 \leq k, k' \leq p+q+r$)

For different base vector interactions, we have

- 4) $\text{sigma_pq} = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_{ir_j} - h_{jr_i})$ (interactionsn between e_i and e_j for $1 \leq i \leq p$ and $p+1 \leq j \leq p+q$)
- 5) $\text{sigma_pr} = \sum_{i=1}^p \sum_{k=p+q+1}^{p+q+r} (h_{ir_k} - h_{kr_i})$ (interactionsn between e_i and e_k for $1 \leq i \leq p$ and $p+q+1 \leq k \leq p+q+r$)
- 6) $\text{sigma_qr} = \sum_{j=p+1}^{p+q} \sum_{k=p+q+1}^{p+q+r} (h_{jr_k} - h_{kr_j})$ (interactionsn between e_j and e_k for $p+1 \leq j \leq p+q$ and $p+q+1 \leq k \leq p+q+r$)

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 $Cl_{\{p,q\}}(\mathbb{R}^d)$.
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this functons are identical Parameter ——— x : torch.LongTensor with (n,2) shape Returns ——— torch.FloatTensor with (n, **IEI**) shape

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

Multiplying a base vector with its scalar coefficient

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

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

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

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

compute_sigma_qq (hq, rq)

Compute $\sigma_{q,q}^* = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (x_{jy_{j'}} - x_{j'}y_j)$ Eq. 16
 σ_q captures the interactions between along q bases For instance, let q e_1, e_2, e_3, we compute interactions between e_1 e_2, e_1 e_3, and e_2 e_3 This can be implemented with a nested two for loops

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

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

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

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

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

e2e1, e2e2, e2e3, e3e1, e3e2, e3e3

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

compute_sigma_rr (hk, rk)

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

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)
```

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

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

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

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

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

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

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

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

Bases: [BaseKGELightning](#)

Base class for all neural network modules.

Your models should also subclass this class.

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

```

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

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

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

```

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

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

Variables

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

forward_byte_pair_encoded_k_vs_all (*x: torch.LongTensor*)

Parameters

x : B x 2 x T

Returns

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

Returns

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

Parameters

x

Returns

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

x shape (b,3,t)

Returns

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

Parameters

x : B x 2 x T

Returns

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

Returns

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

Bases: `dicee.models.base_model.BaseKGE`

A class for using knowledge graph embedding models implemented in Pykeen

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

forward_k_vs_all (*x: torch.LongTensor*)

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

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

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

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

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

else:

t = self.entity_embeddings.weight

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

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

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

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

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

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

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

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

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

Variables

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

forward_byte_pair_encoded_k_vs_all (*x: torch.LongTensor*)

Parameters

x : B x 2 x T

Returns

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

Returns

forward_triples (*x: torch.LongTensor*) \rightarrow torch.Tensor

Parameters

x

Returns

```
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

x shape (b,3,t)

Returns

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

Parameters

x : B x 2 x T

Returns

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

Returns

```
class dicee.models.FMult (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    compute_func (weights: torch.FloatTensor, x) → torch.FloatTensor
    chain_func (weights, x: torch.FloatTensor)
    forward_triples (idx_triple: torch.Tensor) → torch.Tensor
```

Parameters

x

Returns

```
class dicee.models.GFMult (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    compute_func (weights: torch.FloatTensor, x) → torch.FloatTensor
    chain_func (weights, x: torch.FloatTensor)
    forward_triples (idx_triple: torch.Tensor) → torch.Tensor
```

Parameters

x

Returns

```
class dicee.models.FMult2 (args)
    Bases: dicee.models.base_model.BaseKGE
    Learning Knowledge Neural Graphs
    build_func (Vec)
    build_chain_funcs (list_Vec)
    compute_func (W, b, x) → torch.FloatTensor
    function (list_W, list_b)
    trapezoid (list_W, list_b)
    forward_triples (idx_triple: torch.Tensor) → torch.Tensor
```

Parameters

x

Returns

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

Bases: `dicee.models.base_model.BaseKGE`

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

forward_triples (*idx_triple*)

Parameters

x

Returns

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

Returns

construct_multi_coeff (*x*)

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

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

linear (*x, w, b*)

scalar_batch_NN (*a, b, c*)

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

tri_score (*coeff_h, coeff_r, coeff_t*)

this part implement the trilinear scoring techniques:

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

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

vtp_score (*h, r, t*)

this part implement the vector triple product scoring techniques:

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

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

comp_func (*h, r, t*)

this part implement the function composition scoring techniques: i.e. $\text{score} = \langle \text{hor}, t \rangle$

polynomial (*coeff, x, degree*)

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

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

pop (*coeff, x, degree*)

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

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

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

`dicee.read_preprocess_save_load_kg`

Submodules

`dicee.read_preprocess_save_load_kg.preprocess`

Module Contents

Classes

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

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

Preprocess the data in memory

start () → None

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

Parameter

Returns

None

preprocess_with_byte_pair_encoding ()

preprocess_with_byte_pair_encoding_with_padding () → None

Returns

preprocess_with_pandas () → None

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

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

Parameter

Returns

None

preprocess_with_polars () → None

Returns

sequential_vocabulary_construction () → None

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

remove_triples_from_train_with_condition ()

`dicee.read_preprocess_save_load_kg.read_from_disk`

Module Contents

Classes

<i>ReadFromDisk</i>	Read the data from disk into memory
---------------------	-------------------------------------

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

Returns

None

add_noisy_triples_into_training ()

`dicee.read_preprocess_save_load_kg.save_load_disk`

Module Contents

Classes

<i>LoadSaveToDisk</i>

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

save ()

load ()

dicee.read_preprocess_save_load_kg.util

Module Contents

Functions

<code>apply_reciprical_or_noise(add_reciprical, eval_model)</code>	(1) Add reciprocal triples (2) Add noisy triples
<code>timeit(func)</code>	
<code>read_with_polars(→ polars.DataFrame)</code>	Load and Preprocess via Polars
<code>read_with_pandas(data_path[, read_only_few, ...])</code>	
<code>read_from_disk(data_path[, read_only_few, ...])</code>	
<code>read_from_triple_store([endpoint])</code>	Read triples from triple store into pandas dataframe
<code>get_er_vocab(data[, file_path])</code>	
<code>get_re_vocab(data[, file_path])</code>	
<code>get_ee_vocab(data[, file_path])</code>	
<code>create_constraints(triples[, file_path])</code>	(1) Extract domains and ranges of relations
<code>load_with_pandas(→ None)</code>	Deserialize data
<code>save_numpy_ndarray(*, data, file_path)</code>	
<code>load_numpy_ndarray(*, file_path)</code>	
<code>save_pickle(*, data[, file_path])</code>	
<code>load_pickle(*[, file_path])</code>	
<code>create_reciprical_triples(x)</code>	Add inverse triples into dask dataframe
<code>index_triples_with_pandas(→ das.core.frame.DataFrame)</code>	pan- param train_set pandas dataframe
<code>dataset_sanity_checking(→ None)</code>	param train_set

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

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

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

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

Load and Preprocess via Polars

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

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

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

Read triples from triple store into pandas dataframe

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

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

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

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

(1) Extract domains and ranges of relations

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

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

Deserialize data

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

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

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

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

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

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

```
dicee.read_preprocess_save_load_kg.util.index_triples_with_pandas(train_set,  
    entity_to_idx: dict, relation_to_idx: dict) → pandas.core.frame.DataFrame
```

Parameters

- **train_set** – pandas dataframe
- **entity_to_idx** – a mapping from str to integer index
- **relation_to_idx** – a mapping from str to integer index
- **num_core** – number of cores to be used

Returns

indexed triples, i.e., pandas dataframe

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

Parameters

- **train_set** –
- **num_entities** –
- **num_relations** –

Returns

Package Contents

Classes

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

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

Preprocess the data in memory

start () → None

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

Parameter

Returns

None

preprocess_with_byte_pair_encoding ()

preprocess_with_byte_pair_encoding_with_padding () → None

Returns

preprocess_with_pandas () → None

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

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

Parameter

Returns

None

`preprocess_with_polars()` → None

Returns

`sequential_vocabulary_construction()` → None

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

`remove_triples_from_train_with_condition()`

`class dicee.read_preprocess_save_load_kg.LoadSaveToDisk(kg)`

`save()`

`load()`

`class dicee.read_preprocess_save_load_kg.ReadFromDisk(kg)`

Read the data from disk into memory

`start()` → None

Read a knowledge graph from disk into memory

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

Parameter

None

Returns

None

`add_noisy_triples_into_training()`

`dicee.scripts`

Submodules

`dicee.scripts.index`

Module Contents

Functions

```
get_default_arguments()
```

```
main()
```

`dicee.scripts.index.get_default_arguments()`

`dicee.scripts.index.main()`

`dicee.scripts.run`

Module Contents

Functions

```
get_default_arguments([description])
```

Extends pytorch_lightning Trainer's arguments with ours

```
main()
```

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

Extends pytorch_lightning Trainer's arguments with ours

`dicee.scripts.run.main()`

`dicee.scripts.serve`

Module Contents

Classes

```
NeuralSearcher
```

Functions

get_default_arguments()

root()

search_embeddings(q)

retrieve_embeddings(q)

main()

Attributes

app

neural_searcher

`dicee.scripts.serve.app`

`dicee.scripts.serve.neural_searcher`

`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`

Module Contents

Classes

<code>DICE_Trainer</code>	DICE_Trainer implement
---------------------------	------------------------

Functions

<code>initialize_trainer</code> (args, callbacks)
<code>get_callbacks</code> (args)

`dicee.trainer.dice_trainer.initialize_trainer` (args, callbacks)

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

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

DICE_Trainer implement

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

args

is_continual_training:bool

storage_path:str

evaluator:

report:dict

continual_start ()

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

Parameter

Returns

model: form_of_labelling: str

initialize_trainer (callbacks: List) → lightning.Trainer

Initialize Trainer from input arguments

initialize_or_load_model ()

initialize_dataloader (dataset: torch.utils.data.Dataset) → torch.utils.data.DataLoader

initialize_dataset (*dataset: dicee.knowledge_graph.KG, form_of_labelling*)
→ torch.utils.data.Dataset

start (*knowledge_graph: dicee.knowledge_graph.KG*) → Tuple[dicee.models.base_model.BaseKGE, str]
Train selected model via the selected training strategy

k_fold_cross_validation (*dataset*) → Tuple[dicee.models.base_model.BaseKGE, str]
Perform K-fold Cross-Validation

1. Obtain K train and test splits.
2. **For each split,**
 - 2.1 initialize trainer and model
 - 2.2. Train model with configuration provided in args.
 - 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.
3. Report the mean and average MRR .

Parameters

- **self** –
- **dataset** –

Returns
model

`dicee.trainer.torch_trainer`

Module Contents

Classes

<i>TorchTrainer</i>	TorchTrainer for using single GPU or multi CPUs on a single node
---------------------	--

class `dicee.trainer.torch_trainer.TorchTrainer` (*args, callbacks*)

Bases: `dicee.abstracts.AbstractTrainer`

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

Arguments

callbacks: list of Abstract callback instances

fit (**args, train_data loaders, **kwargs*) → None

Training starts

Arguments

kwargs: Tuple

empty dictionary

Returns

batch loss (float)

forward_backward_update (*x_batch: torch.Tensor, y_batch: torch.Tensor*) → torch.Tensor

Compute forward, loss, backward, and parameter update

Arguments

Returns

batch loss (float)

extract_input_outputs_set_device (*batch: list*) → Tuple

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

Arguments

Returns

(tuple) mini-batch on select device

`dicee.trainer.torch_trainer_ddp`

Module Contents

Classes

<i>TorchDDPTrainer</i>	A Trainer based on torch.nn.parallel.DistributedDataParallel
<i>NodeTrainer</i>	
<i>DDPTrainer</i>	

Functions

<i>print_peak_memory</i> (prefix, device)

`dicee.trainer.torch_trainer_ddp.print_peak_memory` (*prefix, device*)

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

Bases: `dicee.abstracts.AbstractTrainer`

A Trainer based on torch.nn.parallel.DistributedDataParallel

Arguments

entity_idx
mapping.

relation_idx
mapping.

form
?

store
?

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

Returns

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

Returns

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

Classes

DICE_Trainer

DICE_Trainer implement

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

DICE_Trainer implement

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

args
 is_continual_training:bool
 storage_path:str
 evaluator:
 report:dict

continual_start ()

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

Parameter

Returns

model: form_of_labelling: str

initialize_trainer (*callbacks: List*) → lightning.Trainer

Initialize Trainer from input arguments

initialize_or_load_model ()

initialize_dataloader (*dataset: torch.utils.data.Dataset*) → torch.utils.data.DataLoader

initialize_dataset (*dataset: dicee.knowledge_graph.KG, form_of_labelling*)
 → torch.utils.data.Dataset

start (*knowledge_graph: dicee.knowledge_graph.KG*) → Tuple[dicee.models.base_model.BaseKGE, str]

Train selected model via the selected training strategy

k_fold_cross_validation (*dataset*) → Tuple[dicee.models.base_model.BaseKGE, str]

Perform K-fold Cross-Validation

1. Obtain K train and test splits.
2. **For each split,**
 - 2.1 initialize trainer and model
 - 2.2. Train model with configuration provided in args.
 - 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.
3. Report the mean and average MRR .

Parameters

- **self** –
- **dataset** –

Returns

model

Submodules

`dicee.abstracts`

Module Contents

Classes

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

class `dicee.abstracts.AbstractTrainer` (*args*, *callbacks*)
Abstract class for Trainer class for knowledge graph embedding models

Parameter

args
[str] ?

callbacks: list
?

on_fit_start (**args*, ***kwargs*)
A function to call callbacks before the training starts.

Parameter

args
kwargs

Returns

None

on_fit_end (**args*, ***kwargs*)
A function to call callbacks at the end of the training.

Parameter

args

kwargs

Returns

None

on_train_epoch_end (*args, **kwargs)

A function to call callbacks at the end of an epoch.

Parameter

args

kwargs

Returns

None

on_train_batch_end (*args, **kwargs)

A function to call callbacks at the end of each mini-batch during training.

Parameter

args

kwargs

Returns

None

static save_checkpoint (full_path: str, model) → None

A static function to save a model into disk

Parameter

full_path : str

model:

Returns

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

property name

get_eval_report () → dict

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

Parameters

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

Returns

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

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

Parameters

triples

Returns

get_domain_of_relation (*rel: str*) → List[str]

get_range_of_relation (*rel: str*) → List[str]

set_model_train_mode () → None
Setting the model into training mode

Parameter

Returns

set_model_eval_mode () → None

Setting the model into eval mode

Parameter

Returns

sample_entity (*n: int*) → List[str]

sample_relation (*n: int*) → List[str]

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

save () → None

get_entity_index (*x: str*)

get_relation_index (*x: str*)

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

Index Triple

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

pytorch tensor of triple score

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

get_entity_embeddings (*items: List[str]*)

Return embedding of an entity given its string representation

Parameter

items:
entities

Returns

get_relation_embeddings (*items: List[str]*)
Return embedding of a relation given its string representation

Parameter

items:
relations

Returns

construct_input_and_output (*head_entity: List[str], relation: List[str], tail_entity: List[str], labels*)
Construct a data point :param head_entity: :param relation: :param tail_entity: :param labels: :return:
parameters ()

class dicee.abstracts.**AbstractCallback**
Bases: abc.ABC, lightning.pytorch.callbacks.Callback
Abstract class for Callback class for knowledge graph embedding models

Parameter

on_init_start (**args, **kwargs*)

Parameter

trainer:
model:

Returns

None
on_init_end (**args, **kwargs*)
Call at the beginning of the training.

Parameter

trainer:

model:

Returns

None

on_fit_start (*trainer, model*)

Call at the beginning of the training.

Parameter

trainer:

model:

Returns

None

on_train_epoch_end (*trainer, model*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

Returns

None

on_train_batch_end (**args, **kwargs*)

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

Parameter

trainer:

model:

Returns

None

on_fit_end (*args, **kwargs)
Call at the end of the training.

Parameter

trainer:

model:

Returns

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:

Returns

None

on_fit_end (trainer, model)
Call at the end of the training.

Parameter

trainer:

model:

Returns

None

store_ensemble (*param_ensemble*) → None

`dicee.analyse_experiments`

This script should be moved to `dicee/scripts`

Module Contents

Classes

Experiment

Functions

get_default_arguments()

analyse(args)

`dicee.analyse_experiments.get_default_arguments()`

class `dicee.analyse_experiments.Experiment`

save_experiment (*x*)

to_df ()

`dicee.analyse_experiments.analyse(args)`

dicee.callbacks

Module Contents

Classes

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

Functions

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

class `dicee.callbacks.AccumulateEpochLossCallback` (*path: str*)

Bases: `dicee.abstracts.AbstractCallback`

Abstract class for Callback class for knowledge graph embedding models

Parameter

on_fit_end (*trainer, model*) → None

Store epoch loss

Parameter

trainer:

model:

Returns

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:

Returns

None

on_fit_end (*trainer, pl_module*)

Call at the end of the training.

Parameter

trainer:

model:

Returns

None

on_train_batch_end (**args, **kwargs*)

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

Parameter

trainer:

model:

Returns

None

on_train_epoch_end (*args, **kwargs)

Call at the end of each epoch during training.

Parameter

trainer:

model:

Returns

None

class dicee.callbacks.KGESaveCallback (every_x_epoch: int, max_epochs: int, path: str)

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

on_train_batch_end (*args, **kwargs)

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

Parameter

trainer:

model:

Returns

None

on_fit_start (trainer, pl_module)

Call at the beginning of the training.

Parameter

trainer:

model:

Returns

None

on_train_epoch_end (*args, **kwargs)

Call at the end of each epoch during training.

Parameter

trainer:

model:

Returns

None

on_fit_end (*args, **kwargs)

Call at the end of the training.

Parameter

trainer:

model:

Returns

None

on_epoch_end (model, trainer, **kwargs)

class dicee.callbacks.**PseudoLabellingCallback** (data_module, kg, batch_size)

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

create_random_data ()

on_epoch_end (trainer, model)

dicee.callbacks.estimate_q (eps)

estimate rate of convergence q from sequence esp

dicee.callbacks.compute_convergence (seq, i)

class `dicee.callbacks.ASWA` (*num_epochs, path*)

Bases: `dicee.abstracts.AbstractCallback`

Adaptive stochastic weight averaging ASWE keeps track of the validation performance and updates the ensemble model accordingly.

on_fit_end (*trainer, model*)

Call at the end of the training.

Parameter

trainer:

model:

Returns

None

static compute_mrr (*trainer, model*) → float

get_aswa_state_dict (*model*)

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

Perform Hard Update, software or rejection

Parameters

running_model_state_dict ensemble_state_dict val_running_model mrr_updated_ensemble_model

Returns

on_train_epoch_end (*trainer, model*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

Returns

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:

Returns

None

on_fit_end (*trainer, model*)

Call at the end of the training.

Parameter

trainer:

model:

Returns

None

on_train_epoch_end (*trainer, model*)

Call at the end of each epoch during training.

Parameter

trainer:

model:

Returns

None

on_train_batch_end (*args, **kwargs)

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

Parameter

trainer:

model:

Returns

None

class dicee.callbacks.**KronE**

Bases: *dicee.abstracts.AbstractCallback*

Abstract class for Callback class for knowledge graph embedding models

Parameter

static batch_kronecker_product (a, b)

Kronecker product of matrices a and b with leading batch dimensions. Batch dimensions are broadcast. The number of them must match :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:

Returns

None

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

Bases: *dicee.abstracts.AbstractCallback*

A callback for a three-Level Perturbation

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

Parameter Perturbation:

Output Perturbation:

on_train_batch_start (*trainer, model, batch, batch_idx*)

Called when the train batch begins.

`dicee.config`

Module Contents

Classes

Namespace

Simple object for storing attributes.

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

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

A single directory created that contains related data about embeddings.

path_single_kg

Path of a file corresponding to the input knowledge graph

sparql_endpoint

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

The ratio of added random triples into training dataset

gpus

Number GPUs to be used during training

callbacks

Callbacks, e.g., {"PPE": {"last_percent_to_consider": 10}}

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

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'

Evaluate trained model choices:["None", "train", "train_val", "train_val_test", "test"]

save_model_at_every_epoch: int

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

Read some triples that are uniformly at random sampled. Ratio being between 0 and 1

read_only_few: int
 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
 WIP: Byte pair encoding

adaptive_swa: bool = False
 Adaptive stochastic weight averaging

swa: bool = False
 Stochastic weight averaging

block_size: int
 block size of LLM

__iter__()

dicee.dataset_classes

Module Contents

Classes

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

Functions

<code>reload_dataset(path, form_of_labelling, ...)</code>	Reload the files from disk to construct the Pytorch dataset
<code>construct_dataset(→ torch.utils.data.Dataset)</code>	

`dicee.dataset_classes.reload_dataset (path: str, form_of_labelling, scoring_technique, neg_ratio, label_smoothing_rate)`

Reload the files from disk to construct the Pytorch dataset

`dicee.dataset_classes.construct_dataset (*, train_set: numpy.ndarray | list, valid_set=None, test_set=None, ordered_bpe_entities=None, train_target_indices=None, target_dim: int = None, entity_to_idx: dict, relation_to_idx: dict, form_of_labelling: str, scoring_technique: str, neg_ratio: int, label_smoothing_rate: float, byte_pair_encoding=None, block_size: int = None)`
→ `torch.utils.data.Dataset`

class `dicee.dataset_classes.BPE_NegativeSamplingDataset (`
 `train_set: torch.LongTensor, ordered_shaped_bpe_entities: torch.LongTensor, neg_ratio: int)`

Bases: `torch.utils.data.Dataset`

An abstract class representing a Dataset.

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

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

`__len__()`

`__getitem__(idx)`

```
collate_fn (batch_shaped_bpe_triples: List[Tuple[torch.Tensor, torch.Tensor]])
```

```
class dicee.dataset_classes.MultiLabelDataset (train_set: torch.LongTensor,  
        train_indices_target: torch.LongTensor, target_dim: int,  
        torch_ordered_shaped_bpe_entities: torch.LongTensor)
```

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

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

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

```
__len__ ()
```

```
__getitem__ (idx)
```

```
class dicee.dataset_classes.MultiClassClassificationDataset (  
        subword_units: numpy.ndarray, block_size: int = 8)
```

Bases: torch.utils.data.Dataset

Dataset for the 1vsALL training strategy

Parameters

train_set_idx

Indexed triples for the training.

entity_idx

mapping.

relation_idx

mapping.

form

?

num_workers

int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

Returns

torch.utils.data.Dataset

```
__len__ ()
```

```
__getitem__ (idx)
```

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

Bases: torch.utils.data.Dataset

Dataset for the 1vsALL training strategy

Parameters

train_set_idx

Indexed triples for the training.

entity_idxxs

mapping.

relation_idxxs

mapping.

form

?

num_workers

int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

Returns

torch.utils.data.Dataset

__len__ ()

__getitem__ (idx)

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

Bases: torch.utils.data.Dataset

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

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

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

Note: TODO

train_set_idx

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

entity_idxxs

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

relation_idxxs

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

self : torch.utils.data.Dataset

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

__len__ ()

__getitem__ (idx)


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

Bases: torch.utils.data.Dataset

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

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

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

Note:

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

train_set_idx

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

entity_idxxs

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

relation_idxxs

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

self : torch.utils.data.Dataset

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

__len__()

__getitem__(idx)

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

Bases: torch.utils.data.Dataset

KvsSample a Dataset:

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

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

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

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

new_y! $<< |E|$ new_y contains all 1's if $\text{sum}(y) < \text{neg_sample_ratio}$ new_y contains

train_set_idx

Indexed triples for the training.

entity_idxxs

mapping.

relation_idxxs

mapping.

form
?
store
?
label_smoothing_rate
?

torch.utils.data.Dataset

__len__()
__getitem__(idx)

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

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

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

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

__len__()
__getitem__(idx)

class dicee.dataset_classes.**TriplePredictionDataset** (train_set: numpy.ndarray, num_entities: int, num_relations: int, neg_sample_ratio: int = 1, label_smoothing_rate: float = 0.0)

Bases: torch.utils.data.Dataset

Triple Dataset

D:= {(x)_i}_i ^N, where
 . x:(h,r, t) in KG is a unique h in E and a relation r in R and . collect_fn => Generates negative triples

collect_fn:

orall (h,r,t) in G obtain, create negative triples{(h,r,x),(r,t),(h,m,t)}

y:labels are represented in torch.float16

train_set_idx
Indexed triples for the training.

entity_idx
mapping.

relation_idx
mapping.

```

    form
        ?

    store
        ?

    label_smoothing_rate

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

__len__ ()

__getitem__ (idx)

collate_fn (batch: List[torch.Tensor])

```

```

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

```

Parameters

```

train_set_idx
    Indexed triples for the training.

num_entities
    entity to index mapping.

num_relations
    relation to index mapping.

batch_size
    int

form
    ?

num_workers
    int for https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

```

Returns

```

?

train_dataloader () → torch.utils.data.DataLoader
    An iterable or collection of iterables specifying training samples.

    For more information about multiple dataloaders, see this section.

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

    For data processing use the following pattern:
    
```

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

However, the above are only necessary for distributed processing.

Warning: do not assign state in `prepare_data`

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

Note:

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

setup (*args, **kwargs)

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

Args:

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

Example:

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

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

        # don't do this
        self.something = else

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

transfer_batch_to_device (*args, **kwargs)

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

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

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

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

Note:

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

Args:

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
```

Raises:**MisconfigurationException:**

If using IPUs, Trainer(accelerator='ipu').

See Also:

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

prepare_data (*args, **kwargs)

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

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

Example:

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

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

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

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

Example:

```

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

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

```

This is called before requesting the dataloaders:

```

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

```

`dicee.eval_static_funcs`

Module Contents

Functions

```

evaluate_link_prediction_performance(→ Parameters
Dict)
evaluate_link_prediction_performance_w

evaluate_link_prediction_performance_w

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

```

```

dicee.eval_static_funcs.evaluate_link_prediction_performance(
    model: dicee.knowledge_graph_embeddings.KGE, triples, er_vocab: Dict[Tuple, List],
    re_vocab: Dict[Tuple, List]) → Dict

```

Parameters

model triples er_vocab re_vocab

Returns

```
dicee.eval_static_funcs.  
    evaluate_link_prediction_performance_with_reciprocals (  
        model: dicee.knowledge_graph_embeddings.KGE, triples, er_vocab: Dict[Tuple, List])  
  
dicee.eval_static_funcs.  
    evaluate_link_prediction_performance_with_bpe_reciprocals (  
        model: dicee.knowledge_graph_embeddings.KGE, within_entities: List[str], triples: List[List[str]],  
        er_vocab: Dict[Tuple, List])  
  
dicee.eval_static_funcs.evaluate_link_prediction_performance_with_bpe (  
    model: dicee.knowledge_graph_embeddings.KGE, within_entities: List[str], triples: List[Tuple[str]],  
    er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List])
```

Parameters

model triples within_entities er_vocab re_vocab

Returns

```
dicee.eval_static_funcs.evaluate_lp_bpe_k_vs_all (model, triples: List[List[str]],  
    er_vocab=None, batch_size=None, func_triple_to_bpe_representation: Callable = None,  
    str_to_bpe_entity_to_idx=None)
```

`dicee.evaluator`

Module Contents

Classes

<i>Evaluator</i>	Evaluator class to evaluate KGE models in various downstream tasks
------------------	--

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

Evaluator class to evaluate KGE models in various downstream tasks

Arguments

vocab_preparation (*dataset*) → None

A function to wait future objects for the attributes of executor

Arguments

Return

None

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

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

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

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

Evaluate model after reciprocal triples are added

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

Evaluate model after reciprocal triples are added

eval_with_vs_all (*, *train_set, valid_set=None, test_set=None, trained_model, form_of_labelling*)
→ None

Evaluate model after reciprocal triples are added

evaluate_lp_k_vs_all (*model, triple_idx, info=None, form_of_labelling=None*)

Filtered link prediction evaluation. :param model: :param triple_idx: test triples :param info: :param form_of_labelling: :return:

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

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

Parameters

model triples: List of lists info form_of_labelling

Returns

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`

Module Contents

Classes

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

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

A class for Training, Retraining and Evaluation a model.

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

read_or_load_kg ()

read_preprocess_index_serialize_data () → None

Read & Preprocess & Index & Serialize Input Data

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

Parameter

Return

None

load_indexed_data () → None

Load the indexed data from disk into memory

Parameter

Return

None

save_trained_model () → None

Save a knowledge graph embedding model

- (1) Send model to eval mode and cpu.
- (2) Store the memory footprint of the model.
- (3) Save the model into disk.
- (4) Update the stats of KG again ?

Parameter

Return

None

end (*form_of_labelling: str*) → dict

End training

- (1) Store trained model.
- (2) Report runtimes.
- (3) Eval model if required.

Parameter

Returns

A dict containing information about the training and/or evaluation

write_report () → None

Report training related information in a report.json file

start () → dict

Start training

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

Parameter

Returns

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

continual_start () → dict

Start Continual Training

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

Parameter

Returns

A dict containing information about the training and/or evaluation

`dicee.knowledge_graph`

Module Contents

Classes

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

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

Knowledge Graph

```
property entities_str: List
```

```
property relations_str: List
```

```
func triple_to_bpe_representation (triple: List[str])
```

`dicee.knowledge_graph_embeddings`

Module Contents

Classes

<i>KGE</i>	Knowledge Graph Embedding Class for interactive usage of pre-trained models
------------	---

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

Bases: `dicee.abstracts.BaseInteractiveKGE`

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

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

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

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

__str__ ()
Return str(self).

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

predict_missing_head_entity (*relation: List[str] | str, tail_entity: List[str] | str, within=None*)
→ Tuple
Given a relation and a tail entity, return top k ranked head entity.
 $\text{argmax}_{\{e \in E\}} f(e, r, t)$, where $r \in R, t \in E$.

Parameter

relation: Union[List[str], str]
String representation of selected relations.

tail_entity: Union[List[str], str]
String representation of selected entities.

k: int
Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_relations (*head_entity: List[str] | str, tail_entity: List[str] | str, within=None*)
→ Tuple
Given a head entity and a tail entity, return top k ranked relations.
 $\text{argmax}_{\{r \in R\}} f(h, r, t)$, where $h, t \in E$.

Parameter

head_entity: List[str]
String representation of selected entities.

tail_entity: List[str]
String representation of selected entities.

k: int
Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_tail_entity (*head_entity: List[str] | str, relation: List[str] | str, within: List[str] = None*) → torch.FloatTensor

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

argmax__{e in E } f(h,r,e), where h in E and r in R.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

scores

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

Parameters

logits h r t within

Returns

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

Predict missing item in a given triple.

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k item.

Returns: Tuple

Highest K scores and items

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

Predict triple score

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

Returns: Tuple

pytorch tensor of triple score

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

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

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

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

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

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

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

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

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

Find an answer set for EPFO queries including negation and disjunction

Parameter

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

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

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

tnorm: str The t-norm operator.

neg_norm: str The negation norm.

lambda_: float lambda parameter for sugeno and yager negation norms

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

Returns

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

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

Find missing triples

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

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

Return (e,r,x)

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

confidence: float

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

topk: int

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

at_most: int

Stop after finding at_most missing triples

{(e,r,x) | f(e,r,x) > confidence and (e,r,x)}

return G

deploy (*share: bool = False, top_k: int = 10*)

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

Module Contents

Classes

QueryGenerator

```
class dicee.query_generator.QueryGenerator (train_path: str, val_path: str, test_path: str,
      ent2id: Dict = None, rel2id: Dict = None, seed: int = 1, gen_valid: bool = False,
      gen_test: bool = True)

    list2tuple (list_data)

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

    set_global_seed (seed: int)
        Set seed

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

    fill_query (query_structure: List[str | List], ent_in: Dict, ent_out: Dict, answer: int) → bool
        Private method for fill_query logic.

    achieve_answer (query: List[str | List], ent_in: Dict, ent_out: Dict) → set
        Private method for achieve_answer logic. @TODO: Document the code

    write_links (ent_out, small_ent_out)

    ground_queries (query_structure: List[str | List], ent_in: Dict, ent_out: Dict, small_ent_in: Dict,
        small_ent_out: Dict, gen_num: int, query_name: str)
        Generating queries and achieving answers

    unmap (query_type, queries, tp_answers, fp_answers, fn_answers)

    unmap_query (query_structure, query, id2ent, id2rel)

    generate_queries (query_struct: List, gen_num: int, query_type: str)
        Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting
        queries and answers in return @ TODO: create a class for each single query struct

    save_queries (query_type: str, gen_num: int, save_path: str)

    abstract load_queries (path)

    get_queries (query_type: str, gen_num: int)

    static save_queries_and_answers (path: str,
        data: List[Tuple[str, Tuple[collections.defaultdict]]]) → None
        Save Queries into Disk

    static load_queries_and_answers (path: str)
        → List[Tuple[str, Tuple[collections.defaultdict]]]
        Load Queries from Disk to Memory
```


`dicee.sanity_checkers`

Module Contents

Functions

<code>is_sparql_endpoint_alive(sparql_endpoint)</code>	
<code>validate_knowledge_graph(args)</code>	Validating the source of knowledge graph
<code>sanity_checking_with_arguments(args)</code>	

`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`

Module Contents

Functions

<code>create_recipriocal_triples(x)</code>	Add inverse triples into dask dataframe
<code>get_er_vocab(data[, file_path])</code>	
<code>get_re_vocab(data[, file_path])</code>	
<code>get_ee_vocab(data[, file_path])</code>	
<code>timeit(func)</code>	
<code>save_pickle(*[, data, file_path])</code>	
<code>load_pickle([file_path])</code>	
<code>select_model(args[, is_continual_training, storage_path])</code>	
<code>load_model(→ Tuple[object, Tuple[dict, dict]])</code>	Load weights and initialize pytorch module from namespace arguments
<code>load_model_ensemble(...)</code>	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
<code>save_numpy_ndarray(*, data, file_path)</code>	
<code>numpy_data_type_changer(→ numpy.ndarray)</code>	Detect most efficient data type for a given triples
<code>save_checkpoint_model(→ None)</code>	Store Pytorch model into disk

continues on next page

Table 2 – continued from previous page

<code>store(→ None)</code>	Store trained_model model and save embeddings into csv file.
<code>add_noisy_triples(→ pandas.DataFrame)</code>	Add randomly constructed triples
<code>read_or_load_kg(args, cls)</code>	
<code>intialize_model(→ Tuple[object, str])</code>	
<code>load_json(→ dict)</code>	
<code>save_embeddings(→ None)</code>	Save it as CSV if memory allows.
<code>random_prediction(pre_trained_kge)</code>	
<code>deploy_triple_prediction(pre_trained_kge, str_subject, ...)</code>	
<code>deploy_tail_entity_prediction(pre_trained_]</code>	
<code>...)</code>	
<code>deploy_head_entity_prediction(pre_trained_]</code>	
<code>...)</code>	
<code>deploy_relation_prediction(pre_trained_kge, ...)</code>	
<code>vocab_to_parquet(vocab_to_idx, name, ...)</code>	
<code>create_experiment_folder([folder_name])</code>	
<code>continual_training_setup_executor(→ None)</code>	storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data
<code>exponential_function(→ torch.FloatTensor)</code>	
<code>load_numpy(→ numpy.ndarray)</code>	
<code>evaluate(entity_to_idx, scores, easy_answers, hard_answers)</code>	# @TODO: CD: Renamed this function
<code>download_file(url[, destination_folder])</code>	
<code>download_files_from_url(base_url[, destination_folder])</code>	
<code>download_pretrained_model(→ str)</code>	

`dicee.static_funcs.create_recipriocal_triples(x)`

Add inverse triples into dask dataframe :param x: :return:

`dicee.static_funcs.get_er_vocab(data, file_path: str = None)`

`dicee.static_funcs.get_re_vocab(data, file_path: str = None)`

`dicee.static_funcs.get_ee_vocab(data, file_path: str = None)`

`dicee.static_funcs.timeit(func)`

`dicee.static_funcs.save_pickle(*, data: object = None, file_path=str)`

`dicee.static_funcs.load_pickle(file_path=str)`

`dicee.static_funcs.select_model` (*args: dict, is_continual_training: bool = None, storage_path: str = None*)

`dicee.static_funcs.load_model` (*path_of_experiment_folder: str, model_name='model.pt', verbose=0*) → Tuple[object, Tuple[dict, dict]]

Load weights and initialize pytorch module from namespace arguments

`dicee.static_funcs.load_model_ensemble` (*path_of_experiment_folder: str*) → Tuple[*dicee.models.base_model.BaseKGE*, Tuple[pandas.DataFrame, pandas.DataFrame]]

Construct Ensemble Of weights and initialize pytorch module from namespace arguments

- (1) Detect models under given path
- (2) Accumulate parameters of detected models
- (3) Normalize parameters
- (4) Insert (3) into model.

`dicee.static_funcs.save_numpy_ndarray` (*, *data: numpy.ndarray, file_path: str*)

`dicee.static_funcs.numpy_data_type_changer` (*train_set: numpy.ndarray, num: int*) → numpy.ndarray

Detect most efficient data type for a given triples :param train_set: :param num: :return:

`dicee.static_funcs.save_checkpoint_model` (*model, path: str*) → None

Store Pytorch model into disk

`dicee.static_funcs.store` (*trainer, trained_model, model_name: str = 'model', full_storage_path: str = None, save_embeddings_as_csv=False*) → None

Store trained_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param full_storage_path: path to save parameters. :param model_name: string representation of the name of the model. :param trained_model: an instance of BaseKGE see core.models.base_model . :param save_embeddings_as_csv: for easy access of embeddings. :return:

`dicee.static_funcs.add_noisy_triples` (*train_set: pandas.DataFrame, add_noise_rate: float*) → pandas.DataFrame

Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

`dicee.static_funcs.read_or_load_kg` (*args, cls*)

`dicee.static_funcs.intialize_model` (*args: dict, verbose=0*) → Tuple[object, str]

`dicee.static_funcs.load_json` (*p: str*) → dict

`dicee.static_funcs.save_embeddings` (*embeddings: numpy.ndarray, indexes, path: str*) → None

Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:

`dicee.static_funcs.random_prediction` (*pre_trained_kge*)

`dicee.static_funcs.deploy_triple_prediction` (*pre_trained_kge, str_subject, str_predicate, str_object*)

`dicee.static_funcs.deploy_tail_entity_prediction` (*pre_trained_kge, str_subject, str_predicate, top_k*)

`dicee.static_funcs.deploy_head_entity_prediction` (*pre_trained_kge, str_object, str_predicate, top_k*)

```

dicee.static_funcs.deploy_relation_prediction (pre_trained_kge, str_subject, str_object, top_k)

dicee.static_funcs.vocab_to_parquet (vocab_to_idx, name, path_for_serialization, print_into)

dicee.static_funcs.create_experiment_folder (folder_name='Experiments')

dicee.static_funcs.continual_training_setup_executor (executor) → None
    storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data
    full_storage_path:str A path leading to a subdirectory containing KGE related data

dicee.static_funcs.exponential_function (x: numpy.ndarray, lam: float, ascending_order=True) → torch.FloatTensor

dicee.static_funcs.load_numpy (path) → numpy.ndarray

dicee.static_funcs.evaluate (entity_to_idx, scores, easy_answers, hard_answers)
    # @TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

dicee.static_funcs.download_file (url, destination_folder='.')

dicee.static_funcs.download_files_from_url (base_url, destination_folder='.')

dicee.static_funcs.download_pretrained_model (url: str) → str

```

`dicee.static_funcs_training`

Module Contents

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)

```

```

dicee.static_funcs_training.evaluate_lp (model, triple_idx, num_entities, er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List], info='Eval Starts')

```

Evaluate model in a standard link prediction task

for each triple the rank is computed by taking the mean of the filtered missing head entity rank and the filtered missing tail entity rank :param model: :param triple_idx: :param info: :return:

```

dicee.static_funcs_training.evaluate_bpe_lp (model, triple_idx: List[Tuple], all_bpe_shaped_entities, er_vocab: Dict[Tuple, List], re_vocab: Dict[Tuple, List], info='Eval Starts')

```

```

dicee.static_funcs_training.efficient_zero_grad (model)

```

dicee.static_preprocess_funcs

Module Contents

Functions

<code>timeit(func)</code>	
<code>preprocesses_input_args(args)</code>	Sanity Checking in input arguments
<code>create_constraints(→ Tuple[dict, dict, dict, dict])</code>	(1) Extract domains and ranges of relations
<code>get_er_vocab(data)</code>	
<code>get_re_vocab(data)</code>	
<code>get_ee_vocab(data)</code>	
<code>mapping_from_first_two_cols_to_third(triples)</code>	

Attributes

<code>enable_log</code>

`dicee.static_preprocess_funcs.enable_log = False`

`dicee.static_preprocess_funcs.timeit(func)`

`dicee.static_preprocess_funcs.preprocesses_input_args(args)`
Sanity Checking in input arguments

`dicee.static_preprocess_funcs.create_constraints(triples: numpy.ndarray)`
→ Tuple[dict, dict, dict, dict]

(1) Extract domains and ranges of relations

(2) Store a mapping from relations to entities that are outside of the domain and range. Create constraints entities based on the range of relations :param triples: :return:

`dicee.static_preprocess_funcs.get_er_vocab(data)`

`dicee.static_preprocess_funcs.get_re_vocab(data)`

`dicee.static_preprocess_funcs.get_ee_vocab(data)`

`dicee.static_preprocess_funcs.mapping_from_first_two_cols_to_third(train_set_idx)`

Package Contents

Classes

<i>CMult</i>	Cl_(0,0) => Real Numbers
<i>Pyke</i>	A Physical Embedding Model for Knowledge Graphs
<i>DistMult</i>	Embedding Entities and Relations for Learning and Inference in Knowledge Bases
<i>KeciBase</i>	Without learning dimension scaling
<i>Keci</i>	Base class for all neural network modules.
<i>TransE</i>	Translating Embeddings for Modeling
<i>DeCaL</i>	Base class for all neural network modules.
<i>ComplEx</i>	Base class for all neural network modules.
<i>AConEx</i>	Additive Convolutional ComplEx Knowledge Graph Embeddings
<i>AConvO</i>	Additive Convolutional Octonion Knowledge Graph Embeddings
<i>AConvQ</i>	Additive Convolutional Quaternion Knowledge Graph Embeddings
<i>ConvQ</i>	Convolutional Quaternion Knowledge Graph Embeddings
<i>ConvO</i>	Base class for all neural network modules.
<i>ConEx</i>	Convolutional ComplEx Knowledge Graph Embeddings
<i>QMult</i>	Base class for all neural network modules.
<i>OMult</i>	Base class for all neural network modules.
<i>Shallom</i>	A shallow neural model for relation prediction (https://arxiv.org/abs/2101.09090)
<i>LFMult</i>	Embedding with polynomial functions. We represent all entities and relations in the polynomial space as:
<i>PykeenKGE</i>	A class for using knowledge graph embedding models implemented in Pykeen
<i>Byte</i>	Base class for all neural network modules.
<i>BaseKGE</i>	Base class for all neural network modules.
<i>DICE_Trainer</i>	DICE_Trainer implement
<i>KGE</i>	Knowledge Graph Embedding Class for interactive usage of pre-trained models
<i>Execute</i>	A class for Training, Retraining and Evaluation a model.
<i>BPE_NegativeSamplingDataset</i>	An abstract class representing a Dataset.
<i>MultiLabelDataset</i>	An abstract class representing a Dataset.
<i>MultiClassClassificationDataset</i>	Dataset for the 1vsALL training strategy
<i>OnevsAllDataset</i>	Dataset for the 1vsALL training strategy
<i>KvsAll</i>	Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.
<i>AllvsAll</i>	Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.
<i>KvsSampleDataset</i>	KvsSample a Dataset:
<i>NegSampleDataset</i>	An abstract class representing a Dataset.
<i>TriplePredictionDataset</i>	Triple Dataset
<i>CVDataModule</i>	Create a Dataset for cross validation
<i>QueryGenerator</i>	

Functions

<code>create_recipriocal_triples(x)</code>	Add inverse triples into dask dataframe
<code>get_er_vocab(data[, file_path])</code>	
<code>get_re_vocab(data[, file_path])</code>	
<code>get_ee_vocab(data[, file_path])</code>	
<code>timeit(func)</code>	
<code>save_pickle(*[, data, file_path])</code>	
<code>load_pickle([file_path])</code>	
<code>select_model(args[, is_continual_training, storage_path])</code>	
<code>load_model(→ Tuple[object, Tuple[dict, dict]])</code>	Load weights and initialize pytorch module from namespace arguments
<code>load_model_ensemble(...)</code>	Construct Ensemble Of weights and initialize pytorch module from namespace arguments
<code>save_numpy_ndarray(*, data, file_path)</code>	
<code>numpy_data_type_changer(→ numpy.ndarray)</code>	Detect most efficient data type for a given triples
<code>save_checkpoint_model(→ None)</code>	Store Pytorch model into disk
<code>store(→ None)</code>	Store trained_model model and save embeddings into csv file.
<code>add_noisy_triples(→ pandas.DataFrame)</code>	Add randomly constructed triples
<code>read_or_load_kg(args, cls)</code>	
<code>intialize_model(→ Tuple[object, str])</code>	
<code>load_json(→ dict)</code>	
<code>save_embeddings(→ None)</code>	Save it as CSV if memory allows.
<code>random_prediction(pre_trained_kge)</code>	
<code>deploy_triple_prediction(pre_trained_kge, str_subject, ...)</code>	
<code>deploy_tail_entity_prediction(pre_trained_</code> <code>...)</code>	
<code>deploy_head_entity_prediction(pre_trained_</code> <code>...)</code>	
<code>deploy_relation_prediction(pre_trained_kge,</code> <code>...)</code>	
<code>vocab_to_parquet(vocab_to_idx, name, ...)</code>	
<code>create_experiment_folder([folder_name])</code>	

continues on next page

Table 4 – continued from previous page

<code>continual_training_setup_executor(→ None)</code>	storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data
<code>exponential_function(→ torch.FloatTensor)</code>	
<code>load_numpy(→ numpy.ndarray)</code>	
<code>evaluate(entity_to_idx, scores, easy_answers, hard_answers)</code>	# @TODO: CD: Renamed this function
<code>download_file(url[, destination_folder])</code>	
<code>download_files_from_url(base_url[, destination_folder])</code>	
<code>download_pretrained_model(→ str)</code>	
<code>mapping_from_first_two_cols_to_third(tr</code>	
<code>timeit(func)</code>	
<code>load_pickle([file_path])</code>	
<code>reload_dataset(path, form_of_labelling, ...)</code>	Reload the files from disk to construct the Pytorch dataset
<code>construct_dataset(→ torch.utils.data.Dataset)</code>	

Attributes

<code>__version__</code>

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

$$\mathbf{a} \text{ imes } \mathbf{b} = a_0 b_0 + a_0 b_1 e_1 + a_1 b_1 e_1 e_1 \\ = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) e_1$$

`Cl_(2,0)` =>

A multivector $\mathbf{a} = a_0 + a_1 e_1 + a_2 e_2 + a_{12} e_1 e_2$ A multivector $\mathbf{b} = b_0 + b_1 e_1 + b_2 e_2 + b_{12} e_1 e_2$

$$\mathbf{a} \text{ imes } \mathbf{b} = a_0 b_0 + a_0 b_1 e_1 + a_0 b_2 e_2 + a_0 b_{12} e_1 e_2 \\ + a_1 b_0 e_1 + a_1 b_1 e_1 e_1 + \dots$$

`Cl_(0,2)` => Quaternions

`clifford_mul` (*x: torch.FloatTensor, y: torch.FloatTensor, p: int, q: int*) → tuple

Clifford multiplication $Cl_{\{p,q\}}(\mathbb{R})$

$e_i^2 = +1$ for $i \leq p$ $e_j^2 = -1$ for $p < j \leq p+q$ $e_i e_j = -e_j e_i$ for i

e_j

x: torch.FloatTensor with (n,d) shape

y: torch.FloatTensor with (n,d) shape

p: a non-negative integer $p \geq 0$ q: a non-negative integer $q \geq 0$

score (*head_ent_emb*, *rel_ent_emb*, *tail_ent_emb*)

forward_triples (*x*: torch.LongTensor) \rightarrow torch.FloatTensor

Compute batch triple scores

Parameter

x: torch.LongTensor with shape n by 3

Returns

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

Returns

torch.LongTensor with shape n

class `dicce.Pyke` (*args*)

Bases: `dicce.models.base_model.BaseKGE`

A Physical Embedding Model for Knowledge Graphs

forward_triples (*x*: torch.LongTensor)

Parameters

x

Returns

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

Returns

forward_k_vs_all (*x: torch.LongTensor*)

forward_k_vs_sample (*x: torch.LongTensor, target_entity_idx: torch.LongTensor*)

score (*h, r, t*)

class `dicee.KeciBase` (*args*)

Bases: `Keci`

Without learning dimension scaling

class `dicee.Keci` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

compute_sigma_pp (*hp, rp*)

Compute $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_{i,r_k} - h_{k,r_i}) e_i e_k$

σ_{pp} captures the interactions between along p bases. For instance, let $p = e_1, e_2, e_3$, we compute interactions between $e_1 e_2, e_1 e_3$, and $e_2 e_3$. This can be implemented with a nested two for loops

```
results = [] for i in range(p - 1):
```

```
    for k in range(i + 1, p):
```

```
        results.append(hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i])
```

```
sigma_pp = torch.stack(results, dim=2) assert sigma_pp.shape == (b, r, int((p * (p - 1)) / 2))
```

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., $e_1 e_1, e_1 e_2, e_1 e_3$,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals: $e_1 e_2, e_1 e_3, e_2 e_3$.

compute_sigma_qq (*hq, rq*)

Compute $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_{j,r_k} - h_{k,r_j}) e_j e_k$. σ_{qq} captures the interactions between along q bases. For instance, let $q = e_1, e_2, e_3$, we compute interactions between $e_1 e_2, e_1 e_3$, and $e_2 e_3$. This can be implemented with a nested two for loops

```
results = [] for j in range(q - 1):
```

```
    for k in range(j + 1, q):
```

```
        results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])
```

```
sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))
```

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., $e_1 e_1, e_1 e_2, e_1 e_3$,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals: $e_1 e_2, e_1 e_3, e_2 e_3$.

compute_sigma_pq (**, hp, hq, rp, rq*)

$\sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_{i,r_j} - h_{j,r_i}) e_i e_j$

```
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

$$h = h_0 + \sum_{i=1}^p h_i e_i + \sum_{j=p+1}^{p+q} h_j e_j \quad r = r_0 + \sum_{i=1}^p r_i e_i + \sum_{j=p+1}^{p+q} r_j e_j$$
$$e_i^2 = +1 \text{ for } i \leq p, e_j^2 = -1 \text{ for } p < j \leq p+q, e_i e_j = -e_j e_i \text{ for } i$$

$e_i e_j$

$h_r = \sigma_0 + \sigma_p + \sigma_q + \sigma_{pp} + \sigma_q + \sigma_{pq}$ where

(1) $\sigma_0 = h_{r_0} + \sum_{i=1}^p (h_{r_i} e_i - \sum_{j=p+1}^{p+q} (h_{r_j} e_j$

(2) $\sigma_p = \sum_{i=1}^p (h_{r_i} + h_{r_0}) e_i$

(3) $\sigma_q = \sum_{j=p+1}^{p+q} (h_{r_j} + h_{r_0}) e_j$

(4) $\sigma_{pp} = \sum_{i=1}^{p-1} \sum_{k=i+1}^p (h_{r_k} - h_{r_i}) e_i e_k$

(5) $\sigma_{qq} = \sum_{j=1}^{p+q-1} \sum_{k=j+1}^{p+q} (h_{r_k} - h_{r_j}) e_j e_k$

(6) $\sigma_{pq} = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_{r_j} - h_{r_i}) e_i e_j$

construct_cl_multivector (*x: torch.FloatTensor, r: int, p: int, q: int*)

→ tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]

Construct a batch of multivectors $Cl_{\{p,q\}}(\mathbb{R}^d)$

Parameter

x: torch.FloatTensor with (n,d) shape

Returns

a0: torch.FloatTensor with (n,r) shape ap: torch.FloatTensor with (n,r,p) shape aq: torch.FloatTensor with (n,r,q) shape

forward_k_vs_with_explicit (*x: torch.Tensor*)

k_vs_all_score (*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

Kvsall training

(1) Retrieve real-valued embedding vectors for heads and relations \mathbb{R}^d .

(2) Construct head entity and relation embeddings according to $Cl_{\{p,q\}}(\mathbb{R}^d)$.

(3) Perform Cl multiplication

(4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this functions are identical Parameter ——— x: torch.LongTensor with (n,2) shape Returns ——— torch.FloatTensor with (n, **IEI**) shape

forward_k_vs_sample (*x: torch.LongTensor, target_entity_idx: torch.LongTensor*)

→ torch.FloatTensor

Kvsall training

(1) Retrieve real-valued embedding vectors for heads and relations \mathbb{R}^d .

(2) Construct head entity and relation embeddings according to $Cl_{\{p,q\}}(\mathbb{R}^d)$.

(3) Perform Cl multiplication

(4) Inner product of (3) and all entity embeddings

Parameter

x: torch.LongTensor with (n,2) shape

Returns

torch.FloatTensor with (n, IEI) shape

score (*h, r, t*)

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Parameter

x: torch.LongTensor with (n,3) shape

Returns

torch.FloatTensor with (n) shape

class `dicee.TransE` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Translating Embeddings for Modeling Multi-relational Data <https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf>

score (*head_ent_emb, rel_ent_emb, tail_ent_emb*)

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

class `dicee.DeCaL` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_triples (*x: torch.Tensor*) \rightarrow torch.FloatTensor

Parameter

x: torch.LongTensor with (n,3) shape

Returns

torch.FloatTensor with (n) shape

cl_pqr (*a*)

Input: tensor(batch_size, emb_dim) \rightarrow output: tensor with 1+p+q+r components with size (batch_size, emb_dim/(1+p+q+r)) each.

1) takes a tensor of size (batch_size, emb_dim), split it into 1 + p + q + r components, hence 1+p+q+r must be a divisor of the emb_dim. 2) Return a list of the 1+p+q+r components vectors, each are tensors of size (batch_size, emb_dim/(1+p+q+r))

compute_sigmas_single (*list_h_emb, list_r_emb, list_t_emb*)

here we compute all the sums with no others vectors interaction taken with the scalar product with t, that is, 1) $s_0 = h_{0r_{0t_0}}$ 2) $s_1 = \sum_{i=1}^p h_{ir_{it_0}}$ 3) $s_2 = \sum_{j=p+1}^{p+q} h_{jr_{jt_0}}$ 4) $s_3 = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_{0r_{it_i}} + h_{ir_{0t_i}})$ 5) $s_4 = \sum_{i=p+1}^{p+q} \sum_{j=p+1}^{p+q} (h_{0r_{it_i}} + h_{ir_{0t_i}})$ 5) $s_5 = \sum_{i=p+q+1}^{p+q+r} \sum_{j=p+q+1}^{p+q+r} (h_{0r_{it_i}} + h_{ir_{0t_i}})$

and return:

*****) $\text{sigma}_{0t} = \text{sigma}_0 \cdot t_0 = s_0 + s_1 - s_2$ *****) s_3, s_4 and s_5

compute_sigmas_multivect (*list_h_emb, list_r_emb*)

Here we compute and return all the sums with vectors interaction for the same and different bases.

For same bases vectors interaction we have

- 1) $\text{sigma}_{pp} = \sum_{i=1}^{p-1} \sum_{i'=i+1}^p (h_{ir_{i'}} - h_{i'r_i})$ (models the interactions between e_i and $e_{i'}$ for $1 \leq i, i' \leq p$)
- 2) $\text{sigma}_{qq} = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (h_{jr_{j'}} - h_{j'r_j})$ (models the interactions between e_j and $e_{j'}$ for $p+1 \leq j, j' \leq p+q$)
- 3) $\text{sigma}_{rr} = \sum_{k=p+q+1}^{p+q+r-1} \sum_{k'=k+1}^{p+q+r} (h_{kr_{k'}} - h_{k'r_k})$ (models the interactions between e_k and $e_{k'}$ for $p+q+1 \leq k, k' \leq p+q+r$)

For different base vector interactions, we have

- 4) $\text{sigma}_{pq} = \sum_{i=1}^p \sum_{j=p+1}^{p+q} (h_{ir_j} - h_{jr_i})$ (interactionsn between e_i and e_j for $1 \leq i \leq p$ and $p+1 \leq j \leq p+q$)

- 5) $\sigma_{pr} = \sum_{i=1}^p \sum_{k=p+q+1}^{p+q+r} (h_{ir_k} - h_{kr_i})$ (interactions between e_i and e_k for $1 \leq i \leq p$ and $p+q+1 \leq k \leq p+q+r$)
- 6) $\sigma_{qr} = \sum_{j=p+1}^{p+q} \sum_{j=p+q+1}^{p+q+r} (h_{jr_k} - h_{kr_j})$ (interactions between e_j and e_k for $p+1 \leq j \leq p+q$ and $p+q+1 \leq j \leq p+q+r$)

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 $Cl_{\{p,q\}}(\mathbb{R}^d)$.
- (3) Perform Cl multiplication
- (4) Inner product of (3) and all entity embeddings

forward_k_vs_with_explicit and this function are identical Parameter ——— x : torch.LongTensor with (n,2) shape Returns ——— torch.FloatTensor with (n, **IE**) shape

apply_coefficients ($h0, hp, hq, hk, r0, rp, rq, rk$)

Multiplying a base vector with its scalar coefficient

construct_cl_multivector (x : torch.FloatTensor, re : int, p : int, q : int, r : int)
 \rightarrow 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)

$\sigma_{\{p,p\}}^{**} = \sum_{i=1}^p \sum_{i'=i+1}^p (x_{iy_{i'}} - x_{i'y_i})$

$\sigma_{\{pp\}}$ captures the interactions between along p bases For instance, let p e_1, e_2, e_3 , we compute interactions between $e_1 e_2, e_1 e_3$, and $e_2 e_3$ This can be implemented with a nested two for loops

results = [] for i in range($p - 1$):

for k in range($i + 1, p$):

 results.append($hp[:, :, i] * rp[:, :, k] - hp[:, :, k] * rp[:, :, i]$)

$\sigma_{pp} = \text{torch.stack}(\text{results}, \text{dim}=2)$ assert $\sigma_{pp}.\text{shape} == (b, r, \text{int}((p * (p - 1)) / 2))$

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., $e_1 e_1, e_1 e_2, e_1 e_3$,

$e_2 e_1, e_2 e_2, e_2 e_3, e_3 e_1, e_3 e_2, e_3 e_3$

Then select the triangular matrix without diagonals: $e_1 e_2, e_1 e_3, e_2 e_3$.

compute_sigma_qq (hq, rq)

Compute $\sigma_{\{q,q\}}^{**} = \sum_{j=p+1}^{p+q-1} \sum_{j'=j+1}^{p+q} (x_{jy_{j'}} - x_{j'y_j})$ Eq. 16
 $\sigma_{\{q\}}$ captures the interactions between along q bases For instance, let q e_1, e_2, e_3 , we compute interactions between $e_1 e_2, e_1 e_3$, and $e_2 e_3$ This can be implemented with a nested two for loops

```
results = [] for j in range(q - 1):
```

```
    for k in range(j + 1, q):
```

```
        results.append(hq[:, :, j] * rq[:, :, k] - hq[:, :, k] * rq[:, :, j])
```

```
sigma_qq = torch.stack(results, dim=2) assert sigma_qq.shape == (b, r, int((q * (q - 1)) / 2))
```

Yet, this computation would be quite inefficient. Instead, we compute interactions along all p , e.g., $e1e1$, $e1e2$, $e1e3$,

```
e2e1, e2e2, e2e3, e3e1, e3e2, e3e3
```

Then select the triangular matrix without diagonals: $e1e2$, $e1e3$, $e2e3$.

```
compute_sigma_rr (hk, rk)
```

```
sigma_{r,r}^* = sum_{k=p+q+1}^{p+q+r-1} sum_{k'=k+1}^p (x_{ky_{k'}} - x_{k'} y_k)
```

```
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)
```

```
compute_sigma_pr (*, hp, hk, rp, 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)
```

```
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.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)
```

(continues on next page)

(continued from previous page)

```
self.conv2 = nn.Conv2d(20, 20, 5)

def forward(self, x):
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

static score (*head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor, tail_ent_emb: torch.FloatTensor*)

static k_vs_all_score (*emb_h: torch.FloatTensor, emb_r: torch.FloatTensor, emb_E: torch.FloatTensor*)

Parameters

emb_h emb_r emb_E

Returns

forward_k_vs_all (*x: torch.LongTensor*) → torch.FloatTensor

class `dicee.AConEx` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Additive Convolutional ComplEx Knowledge Graph Embeddings

residual_convolution (*C_1: Tuple[torch.Tensor, torch.Tensor], C_2: Tuple[torch.Tensor, torch.Tensor]*) → torch.FloatTensor

Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Parameters

x

Returns

forward_k_vs_sample (x: torch.Tensor, target_entity_idx: torch.Tensor)

class dicee.AConvO (args: dict)

Bases: *dicee.models.base_model.BaseKGE*

Additive Convolutional Octonion Knowledge Graph Embeddings

static octonion_normalizer (emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4, emb_rel_e5, emb_rel_e6, emb_rel_e7)

residual_convolution (O_1, O_2)

forward_triples (x: torch.Tensor) → torch.Tensor

Parameters

x

Returns

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

Returns

forward_k_vs_all (*x*: torch.Tensor)

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **Entities**) Given a batch of head entities and relations => shape (size of batch,|Entities|)

class dicee.ConvQ(*args*)

Bases: *dicee.models.base_model.BaseKGE*

Convolutional Quaternion Knowledge Graph Embeddings

residual_convolution (*Q_1, Q_2*)

forward_triples (*indexed_triple*: torch.Tensor) → torch.Tensor

Parameters

x

Returns

forward_k_vs_all (*x*: torch.Tensor)

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **Entities**) Given a batch of head entities and relations => shape (size of batch,|Entities|)

class dicee.ConvO(*args*: dict)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

static octonion_normalizer (*emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4, emb_rel_e5, emb_rel_e6, emb_rel_e7*)

residual_convolution (*O_1, O_2*)

forward_triples (*x: torch.Tensor*) → torch.Tensor

Parameters

x

Returns

forward_k_vs_all (*x: torch.Tensor*)

Given a head entity and a relation (h,r), we compute scores for all entities. [score(h,r,x)|x in Entities] => [0.0,0.1,...,0.8], shape=> (1, **Entities**) Given a batch of head entities and relations => shape (size of batch,|Entities|)

class dicee.**ConEx** (*args*)

Bases: *dicee.models.base_model.BaseKGE*

Convolutional ComplEx Knowledge Graph Embeddings

residual_convolution (*C_1: Tuple[torch.Tensor, torch.Tensor], C_2: Tuple[torch.Tensor, torch.Tensor]*) → torch.FloatTensor

Compute residual score of two complex-valued embeddings. :param C_1: a tuple of two pytorch tensors that corresponds complex-valued embeddings :param C_2: a tuple of two pytorch tensors that corresponds complex-valued embeddings :return:

forward_k_vs_all (*x: torch.Tensor*) → torch.FloatTensor

forward_triples (*x: torch.Tensor*) → torch.FloatTensor

Parameters

x

Returns

forward_k_vs_sample (*x*: torch.Tensor, *target_entity_idx*: torch.Tensor)

class dicee.QMult (*args*)

Bases: *dicee.models.base_model.BaseKGE*

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

quaternion_multiplication_followed_by_inner_product (*h, r, t*)

Parameters

- **h** – shape: (**batch_dims*, dim) The head representations.
- **r** – shape: (**batch_dims*, dim) The head representations.
- **t** – shape: (**batch_dims*, dim) The tail representations.

Returns

Triple scores.

static quaternion_normalizer (*x*: torch.FloatTensor) → torch.FloatTensor

Normalize the length of relation vectors, if the forward constraint has not been applied yet.

Absolute value of a quaternion

$$|a + bi + cj + dk| = \sqrt{a^2 + b^2 + c^2 + d^2}$$

L2 norm of quaternion vector:

$$\|x\|^2 = \sum_{i=1}^d |x_i|^2 = \sum_{i=1}^d (x_i.re^2 + x_i.im_1^2 + x_i.im_2^2 + x_i.im_3^2)$$

Parameters

x – The vector.

Returns

The normalized vector.

score (*head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor, tail_ent_emb: torch.FloatTensor*)

k_vs_all_score (*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

Parameters

bpe_head_ent_emb bpe_rel_ent_emb E

Returns

forward_k_vs_all (*x*)

Parameters

x

Returns

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)
```

(continues on next page)

(continued from previous page)

```
self.conv2 = nn.Conv2d(20, 20, 5)

def forward(self, x):
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

static octonion_normalizer (*emb_rel_e0, emb_rel_e1, emb_rel_e2, emb_rel_e3, emb_rel_e4, emb_rel_e5, emb_rel_e6, emb_rel_e7*)

score (*head_ent_emb: torch.FloatTensor, rel_ent_emb: torch.FloatTensor, tail_ent_emb: torch.FloatTensor*)

k_vs_all_score (*bpe_head_ent_emb, bpe_rel_ent_emb, E*)

forward_k_vs_all (*x*)

Completed. Given a head entity and a relation (h,r), we compute scores for all possible triples, i.e., $[\text{score}(h,r,x) | x \text{ in Entities}] \Rightarrow [0.0, 0.1, \dots, 0.8]$, shape $\Rightarrow (1, \text{Entities})$ Given a batch of head entities and relations $\Rightarrow \text{shape}(\text{size of batch}, |\text{Entities}|)$

class `dicee.Shallom` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

A shallow neural model for relation prediction (<https://arxiv.org/abs/2101.09090>)

get_embeddings () \rightarrow Tuple[numpy.ndarray, None]

Returns

forward_k_vs_all (*x*) \rightarrow torch.FloatTensor

forward_triples (*x*) \rightarrow torch.FloatTensor

Parameters

x –

Returns

class `dicee.LFMult` (*args*)

Bases: `dicee.models.base_model.BaseKGE`

Embedding with polynomial functions. We represent all entities and relations in the polynomial space as: $f(x) = \sum_{i=0}^{d-1} a_k x^{i \% d}$ and use the three different scoring function as in the paper to evaluate the score. We also consider combining with Neural Networks.

forward_triples (*idx_triple*)

Parameters

x

Returns

construct_multi_coeff (x)

poly_NN ($x, \text{coefh}, \text{coefr}, \text{coeft}$)

Constructing a 2 layers NN to represent the embeddings. $h = \text{sigma}(w_h^T x + b_h)$, $r = \text{sigma}(w_r^T x + b_r)$,
 $t = \text{sigma}(w_t^T x + b_t)$

linear (x, w, b)

scalar_batch_NN (a, b, c)

element wise multiplication between a,b and c: Inputs : a, b, c ==> torch.tensor of size batch_size x m x d
Output : a tensor of size batch_size x d

tri_score ($\text{coeff}_h, \text{coeff}_r, \text{coeff}_t$)

this part implement the trilinear scoring techniques:

$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*b_j*c_k\}\{1+(i+j+k)\%d\}$

1. generate the range for i,j and k from [0 d-1]
2. perform $\text{dfrac}\{a_i*b_j*c_k\}\{1+(i+j+k)\%d\}$ in parallel for every batch
3. take the sum over each batch

vtp_score (h, r, t)

this part implement the vector triple product scoring techniques:

$\text{score}(h,r,t) = \int_0^1 h(x)r(x)t(x) dx = \sum_{i,j,k=0}^{d-1} \text{dfrac}\{a_i*c_j*b_k - b_i*c_j*a_k\}\{(1+(i+j)\%d)(1+k)\}$

1. generate the range for i,j and k from [0 d-1]
2. Compute the first and second terms of the sum
3. Multiply with then denominator and take the sum
4. take the sum over each batch

comp_func (h, r, t)

this part implement the function composition scoring techniques: i.e. $\text{score} = \langle h, r, t \rangle$

polynomial ($\text{coeff}, x, \text{degree}$)

This function takes a matrix tensor of coefficients (coeff), a tensor vector of points x and range of integer [0,1,...d] and return a vector tensor ($\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d$,

$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d$)

pop ($\text{coeff}, x, \text{degree}$)

This function allow us to evaluate the composition of two polynomes without for loops :) it takes a matrix tensor of coefficients (coeff), a matrix tensor of points x and range of integer [0,1,...d]

and return a tensor ($\text{coeff}[0][0] + \text{coeff}[0][1]x + \dots + \text{coeff}[0][d]x^d$,

$\text{coeff}[1][0] + \text{coeff}[1][1]x + \dots + \text{coeff}[1][d]x^d$)


```
class dicee.PykeenKGE (args: dict)
```

Bases: `dicee.models.base_model.BaseKGE`

A class for using knowledge graph embedding models implemented in Pykeen

Notes: Pykeen_DistMult: C Pykeen_ComplEx: Pykeen_QuatE: Pykeen_MuRE: Pykeen_CP: Pykeen_HolE: Pykeen_HolE:

```
forward_k_vs_all (x: torch.LongTensor)
```

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads and relations + apply Dropout & Normalization if given. h, r = self.get_head_relation_representation(x) # (2) Reshape (1). if self.last_dim > 0:

```
h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim)
```

(3) Reshape all entities. if self.last_dim > 0:

```
t = self.entity_embeddings.weight.reshape(self.num_entities, self.embedding_dim, self.last_dim)
```

else:

```
t = self.entity_embeddings.weight
```

(4) Call the score_t from interactions to generate triple scores. return self.interaction.score_t(h=h, r=r, all_entities=t, slice_size=1)

```
forward_triples (x: torch.LongTensor) → torch.FloatTensor
```

=> Explicit version by this we can apply bn and dropout

(1) Retrieve embeddings of heads, relations and tails and apply Dropout & Normalization if given. h, r, t = self.get_triple_representation(x) # (2) Reshape (1). if self.last_dim > 0:

```
h = h.reshape(len(x), self.embedding_dim, self.last_dim) r = r.reshape(len(x), self.embedding_dim, self.last_dim) t = t.reshape(len(x), self.embedding_dim, self.last_dim)
```

(3) Compute the triple score return self.interaction.score(h=h, r=r, t=t, slice_size=None, slice_dim=0)

```
abstract forward_k_vs_sample (x: torch.LongTensor, target_entity_idx)
```

```
class dicee.ByteE (*args, **kwargs)
```

Bases: `dicee.models.base_model.BaseKGE`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

loss_function (*yhat_batch, y_batch*)

Parameters

yhat_batch y_batch

Returns

forward (*x: torch.LongTensor*)

Parameters

x: B by T tensor

Returns

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.

Args:

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)

Return:

- `Tensor` - The loss tensor
- `dict` - A dictionary which can include any keys, but must include the key `'loss'` in the case of automatic optimization.
- `None` - In automatic optimization, this will skip to the next batch (but is not supported for multi-GPU, TPU, or DeepSpeed). For manual optimization, this has no special meaning, as returning the loss is not required.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

To use multiple optimizers, you can switch to 'manual optimization' and control their stepping:

```
def __init__(self):
    super().__init__()
    self.automatic_optimization = False

# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

    # do training_step with encoder
    ...
    opt1.step()
    # do training_step with decoder
    ...
    opt2.step()
```

Note:

When `accumulate_grad_batches > 1`, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

class `dicee.BaseKGE` (*args: dict*)

Bases: `BaseKGELightning`

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

Note: As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Variables

training (*bool*) – Boolean represents whether this module is in training or evaluation mode.

forward_byte_pair_encoded_k_vs_all (*x: torch.LongTensor*)

Parameters

x : B x 2 x T

Returns

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

Returns

forward_triples (*x: torch.LongTensor*) \rightarrow torch.Tensor

Parameters

x

Returns

```
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

x shape (b,3,t)

Returns

```
get_bpe_head_and_relation_representation (x: torch.LongTensor)
    → Tuple[torch.FloatTensor, torch.FloatTensor]
```

Parameters

x : B x 2 x T

Returns

```
get_embeddings () → Tuple[numpy.ndarray, numpy.ndarray]
```

Returns

```
dicee.create_recipriocal_triples (x)
    Add inverse triples into dask dataframe :param x: :return:
dicee.get_er_vocab (data, file_path: str = None)
dicee.get_re_vocab (data, file_path: str = None)
dicee.get_ee_vocab (data, file_path: str = None)
dicee.timeit (func)
dicee.save_pickle (*, data: object = None, file_path=str)
dicee.load_pickle (file_path=str)
dicee.select_model (args: dict, is_continual_training: bool = None, storage_path: str = None)
```

`dicee.load_model` (*path_of_experiment_folder: str, model_name='model.pt', verbose=0*)
→ Tuple[object, Tuple[dict, dict]]
Load weights and initialize pytorch module from namespace arguments

`dicee.load_model_ensemble` (*path_of_experiment_folder: str*)
→ Tuple[dicee.models.base_model.BaseKGE, Tuple[pandas.DataFrame, pandas.DataFrame]]
Construct Ensemble Of weights and initialize pytorch module from namespace arguments

- (1) Detect models under given path
- (2) Accumulate parameters of detected models
- (3) Normalize parameters
- (4) Insert (3) into model.

`dicee.save_numpy_ndarray` (*, *data: numpy.ndarray, file_path: str*)

`dicee.numpy_data_type_changer` (*train_set: numpy.ndarray, num: int*) → numpy.ndarray
Detect most efficient data type for a given triples :param train_set: :param num: :return:

`dicee.save_checkpoint_model` (*model, path: str*) → None
Store Pytorch model into disk

`dicee.store` (*trainer, trained_model, model_name: str = 'model', full_storage_path: str = None, save_embeddings_as_csv=False*) → None
Store trained_model model and save embeddings into csv file. :param trainer: an instance of trainer class :param full_storage_path: path to save parameters. :param model_name: string representation of the name of the model. :param trained_model: an instance of BaseKGE see core.models.base_model . :param save_embeddings_as_csv: for easy access of embeddings. :return:

`dicee.add_noisy_triples` (*train_set: pandas.DataFrame, add_noise_rate: float*) → pandas.DataFrame
Add randomly constructed triples :param train_set: :param add_noise_rate: :return:

`dicee.read_or_load_kg` (*args, cls*)

`dicee.intialize_model` (*args: dict, verbose=0*) → Tuple[object, str]

`dicee.load_json` (*p: str*) → dict

`dicee.save_embeddings` (*embeddings: numpy.ndarray, indexes, path: str*) → None
Save it as CSV if memory allows. :param embeddings: :param indexes: :param path: :return:

`dicee.random_prediction` (*pre_trained_kge*)

`dicee.deploy_triple_prediction` (*pre_trained_kge, str_subject, str_predicate, str_object*)

`dicee.deploy_tail_entity_prediction` (*pre_trained_kge, str_subject, str_predicate, top_k*)

`dicee.deploy_head_entity_prediction` (*pre_trained_kge, str_object, str_predicate, top_k*)

`dicee.deploy_relation_prediction` (*pre_trained_kge, str_subject, str_object, top_k*)

`dicee.vocab_to_parquet` (*vocab_to_idx, name, path_for_serialization, print_into*)

`dicee.create_experiment_folder` (*folder_name='Experiments'*)

`dicee.continual_training_setup_executor` (*executor*) → None
storage_path:str A path leading to a parent directory, where a subdirectory containing KGE related data
full_storage_path:str A path leading to a subdirectory containing KGE related data

```

dicee.exponential_function (x: numpy.ndarray, lam: float, ascending_order=True)
    → torch.FloatTensor

dicee.load_numpy (path) → numpy.ndarray

dicee.evaluate (entity_to_idx, scores, easy_answers, hard_answers)
    # @TODO: CD: Renamed this function Evaluate multi hop query answering on different query types

dicee.download_file (url, destination_folder='.')

dicee.download_files_from_url (base_url, destination_folder='.')

dicee.download_pretrained_model (url: str) → str

class dicee.DICE_Trainer (args, is_continual_training, storage_path, evaluator=None)

    DICE_Trainer implement
        1- Pytorch Lightning trainer (https://pytorch-lightning.readthedocs.io/en/stable/common/trainer.html)
        2- Multi-GPU Trainer(https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html)
        3- CPU Trainer

        args
        is_continual_training:bool
        storage_path:str
        evaluator:
        report:dict

    continual_start ()

        (1) Initialize training.
        (2) Load model
        (3) Load trainer (3) Fit model

    Parameter

    Returns

        model: form_of_labelling: str

    initialize_trainer (callbacks: List) → lightning.Trainer
        Initialize Trainer from input arguments

    initialize_or_load_model ()

    initialize_dataloader (dataset: torch.utils.data.Dataset) → torch.utils.data.DataLoader

    initialize_dataset (dataset: dicee.knowledge\_graph.KG, form_of_labelling)
        → torch.utils.data.Dataset

    start (knowledge_graph: dicee.knowledge\_graph.KG) → Tuple[dicee.models.base\_model.BaseKGE, str]
        Train selected model via the selected training strategy

```

k_fold_cross_validation (*dataset*) → Tuple[*dicee.models.base_model.BaseKGE*, str]

Perform K-fold Cross-Validation

1. Obtain K train and test splits.
2. **For each split,**
 - 2.1 initialize trainer and model
 - 2.2. Train model with configuration provided in args.
 - 2.3. Compute the mean reciprocal rank (MRR) score of the model on the test respective split.
3. Report the mean and average MRR .

Parameters

- **self** –
- **dataset** –

Returns

model

class *dicee.KGE* (*path=None, url=None, construct_ensemble=False, model_name=None, apply_semantic_constraint=False*)

Bases: *dicee.abstracts.BaseInteractiveKGE*

Knowledge Graph Embedding Class for interactive usage of pre-trained models

get_transductive_entity_embeddings (*indices: torch.LongTensor | List[str], as_pytorch=False, as_numpy=False, as_list=True*)
→ torch.FloatTensor | numpy.ndarray | List[float]

create_vector_database (*collection_name: str, distance: str, location: str = 'localhost', port: int = 6333*)

generate (*h="", r=""*)

__str__ ()

Return str(self).

eval_lp_performance (*dataset=List[Tuple[str, str, str]], filtered=True*)

predict_missing_head_entity (*relation: List[str] | str, tail_entity: List[str] | str, within=None*)
→ Tuple

Given a relation and a tail entity, return top k ranked head entity.

$\text{argmax}_{\{e \in E\}} f(e, r, t)$, where $r \in R, t \in E$.

Parameter

relation: Union[List[str], str]

String representation of selected relations.

tail_entity: Union[List[str], str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_relations (*head_entity: List[str] | str, tail_entity: List[str] | str, within=None*)
→ Tuple

Given a head entity and a tail entity, return top k ranked relations.

$\text{argmax}_{\{r \in R\}} f(h,r,t)$, where $h, t \in E$.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k entities.

Returns: Tuple

Highest K scores and entities

predict_missing_tail_entity (*head_entity: List[str] | str, relation: List[str] | str, within: List[str] = None*) → torch.FloatTensor

Given a head entity and a relation, return top k ranked entities

$\text{argmax}_{\{e \in E\}} f(h,r,e)$, where $h \in E$ and $r \in R$.

Parameter

head_entity: List[str]

String representation of selected entities.

tail_entity: List[str]

String representation of selected entities.

Returns: Tuple

scores

predict (*, *h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, within=None, logits=True*) → torch.FloatTensor

Parameters

logits h r t within

Returns

predict_topk (*, h: List[str] = None, r: List[str] = None, t: List[str] = None, topk: int = 10, within: List[str] = None)

Predict missing item in a given triple.

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

k: int

Highest ranked k item.

Returns: Tuple

Highest K scores and items

triple_score (h: List[str] | str = None, r: List[str] | str = None, t: List[str] | str = None, logits=False)
→ torch.FloatTensor

Predict triple score

Parameter

head_entity: List[str]

String representation of selected entities.

relation: List[str]

String representation of selected relations.

tail_entity: List[str]

String representation of selected entities.

logits: bool

If logits is True, unnormalized score returned

Returns: Tuple

pytorch tensor of triple score

t_norm (*tens_1*: torch.Tensor, *tens_2*: torch.Tensor, *tnorm*: str = 'min') → torch.Tensor

tensor_t_norm (*subquery_scores*: torch.FloatTensor, *tnorm*: str = 'min') → torch.FloatTensor

Compute T-norm over $[0,1]^{n \times d}$ where n denotes the number of hops and d denotes number of entities

t_conorm (*tens_1*: torch.Tensor, *tens_2*: torch.Tensor, *tconorm*: str = 'min') → torch.Tensor

negnorm (*tens_1*: torch.Tensor, *lambda_*: float, *neg_norm*: str = 'standard') → torch.Tensor

return_multi_hop_query_results (*aggregated_query_for_all_entities*, *k*: int, *only_scores*)

single_hop_query_answering (*query*: tuple, *only_scores*: bool = True, *k*: int = None)

answer_multi_hop_query (*query_type*: str = None,
 query: Tuple[str | Tuple[str, str], Ellipsis] = None,
 queries: List[Tuple[str | Tuple[str, str], Ellipsis]] = None, *tnorm*: str = 'prod',
 neg_norm: str = 'standard', *lambda_*: float = 0.0, *k*: int = 10, *only_scores*=False)
 → List[Tuple[str, torch.Tensor]]

@TODO: Refactoring is needed # @TODO: Score computation for each query type should be done in a static function

Find an answer set for EPFO queries including negation and disjunction

Parameter

query_type: str The type of the query, e.g., “2p”.

query: Union[str, Tuple[str, Tuple[str, str]]] The query itself, either a string or a nested tuple.

queries: List of Tuple[Union[str, Tuple[str, str]], ...]

tnorm: str The t-norm operator.

neg_norm: str The negation norm.

lambda_: float lambda parameter for sugeno and yager negation norms

k: int The top-k substitutions for intermediate variables.

Returns

List[Tuple[str, torch.Tensor]] Entities and corresponding scores sorted in the descening order of scores

find_missing_triples (*confidence*: float, *entities*: List[str] = None, *relations*: List[str] = None,
 topk: int = 10, *at_most*: int = sys.maxsize) → Set

Find missing triples

Iterative over a set of entities E and a set of relation R :

orall e in E and orall r in R f(e,r,x)

Return (e,r,x)

otin G and f(e,r,x) > confidence

confidence: float

A threshold for an output of a sigmoid function given a triple.

topk: int

Highest ranked k item to select triples with $f(e,r,x) > \text{confidence}$.

at_most: int

Stop after finding at_most missing triples

$\{(e,r,x) \mid f(e,r,x) > \text{confidence} \text{ and } (e,r,x)$

otin G

deploy (*share: bool = False, top_k: int = 10*)

train triples (*h: List[str], r: List[str], t: List[str], labels: List[float], iteration=2, optimizer=None*)

train_k_vs_all (*h, r, iteration=1, lr=0.001*)

Train k vs all :param head_entity: :param relation: :param iteration: :param lr: :return:

train (*kg, lr=0.1, epoch=10, batch_size=32, neg_sample_ratio=10, num_workers=1*) → None

Retrained a pretrain model on an input KG via negative sampling.

class dicee.**Execute** (*args, continuous_training=False*)

A class for Training, Retraining and Evaluation a model.

- (1) Loading & Preprocessing & Serializing input data.
- (2) Training & Validation & Testing
- (3) Storing all necessary info

read_or_load_kg ()

read_preprocess_index_serialize_data () → None

Read & Preprocess & Index & Serialize Input Data

- (1) Read or load the data from disk into memory.
- (2) Store the statistics of the data.

Parameter

Return

None

load_indexed_data () → None

Load the indexed data from disk into memory

Parameter

Return

None

save_trained_model () → None

Save a knowledge graph embedding model

- (1) Send model to eval mode and cpu.
- (2) Store the memory footprint of the model.
- (3) Save the model into disk.
- (4) Update the stats of KG again ?

Parameter

Return

None

end (*form_of_labelling: str*) → dict

End training

- (1) Store trained model.
- (2) Report runtimes.
- (3) Eval model if required.

Parameter

Returns

A dict containing information about the training and/or evaluation

write_report () → None

Report training related information in a report.json file

start () → dict

Start training

(1) Loading the Data # (2) Create an evaluator object. # (3) Create a trainer object. # (4) Start the training

Parameter

Returns

A dict containing information about the training and/or evaluation

```
dicee.mapping_from_first_two_cols_to_third(train_set_idx)
```

```
dicee.timeit(func)
```

```
dicee.load_pickle(file_path=str)
```

```
dicee.reload_dataset(path: str, form_of_labelling, scoring_technique, neg_ratio, label_smoothing_rate)
```

Reload the files from disk to construct the Pytorch dataset

```
dicee.construct_dataset(*, train_set: numpy.ndarray | list, valid_set=None, test_set=None,
                        ordered_bpe_entities=None, train_target_indices=None, target_dim: int = None, entity_to_idx: dict,
                        relation_to_idx: dict, form_of_labelling: str, scoring_technique: str, neg_ratio: int,
                        label_smoothing_rate: float, byte_pair_encoding=None, block_size: int = None)
→ torch.utils.data.Dataset
```

```
class dicee.BPENegativeSamplingDataset(train_set: torch.LongTensor,
                                       ordered_shaped_bpe_entities: torch.LongTensor, neg_ratio: int)
```

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note: `DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

```
__len__()
```

```
__getitem__(idx)
```

```
collate_fn(batch_shaped_bpe_triples: List[Tuple[torch.Tensor, torch.Tensor]])
```

```
class dicee.MultiLabelDataset(train_set: torch.LongTensor, train_indices_target: torch.LongTensor,
                              target_dim: int, torch_ordered_shaped_bpe_entities: torch.LongTensor)
```

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note: `DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

`__len__()`

`__getitem__(idx)`

```
class dicee.MultiClassClassificationDataset (subword_units: numpy.ndarray,  
      block_size: int = 8)
```

Bases: `torch.utils.data.Dataset`

Dataset for the 1vsALL training strategy

Parameters

train_set_idx

Indexed triples for the training.

entity_idx

mapping.

relation_idx

mapping.

form

?

num_workers

int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

Returns

`torch.utils.data.Dataset`

`__len__()`

`__getitem__(idx)`

```
class dicee.OnevsAllDataset (train_set_idx: numpy.ndarray, entity_idx)
```

Bases: `torch.utils.data.Dataset`

Dataset for the 1vsALL training strategy

Parameters

train_set_idx

Indexed triples for the training.

entity_idx

mapping.

relation_idx

mapping.

form

?

num_workers

int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

Returns

torch.utils.data.Dataset

`__len__()`

`__getitem__(idx)`

```
class dicee.KvsAll (train_set_idx: numpy.ndarray, entity_idxxs, relation_idxxs, form, store=None,
                    label_smoothing_rate: float = 0.0)
```

Bases: torch.utils.data.Dataset

Creates a dataset for KvsAll training by inheriting from torch.utils.data.Dataset.

Let D denote a dataset for KvsAll training and be defined as $D := \{(x, y)_i\}_i^N$, where $x: (h, r)$ is a unique tuple of an entity h in E and a relation r in R that has been seed in the input graph. y : denotes a multi-label vector in $[0, 1]^{|E|}$ **|E|** is a binary label.

orall $y_i = 1$ s.t. $(h \ r \ E_i)$ in KG

Note: TODO

train_set_idx

[numpy.ndarray] n by 3 array representing n triples

entity_idxxs

[dictionary] string representation of an entity to its integer id

relation_idxxs

[dictionary] string representation of a relation to its integer id

self : torch.utils.data.Dataset

```
>>> a = KvsAll()
>>> a
? array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

`__len__()`

`__getitem__(idx)`

```
class dicee.AllvsAll (train_set_idx: numpy.ndarray, entity_idxxs, relation_idxxs,
                      label_smoothing_rate=0.0)
```

Bases: torch.utils.data.Dataset

Creates a dataset for AllvsAll training by inheriting from torch.utils.data.Dataset.

Let D denote a dataset for AllvsAll training and be defined as $D := \{(x, y)_i\}_i^N$, where $x: (h, r)$ is a possible unique tuple of an entity h in E and a relation r in R . Hence $N = |E| \times |R|$ y : denotes a multi-label vector in $[0, 1]^{|E|}$ **|E|** is a binary label.

orall $y_i = 1$ s.t. $(h \ r \ E_i)$ in KG

Note:

AllvsAll extends KvsAll via none existing (h,r). Hence, it adds data points that are labelled without 1s, only with 0s.

train_set_idx
[numpy.ndarray] n by 3 array representing n triples

entity_idxes
[dictionary] string representation of an entity to its integer id

relation_idxes
[dictionary] string representation of a relation to its integer id

self : torch.utils.data.Dataset

```
>>> a = AllvsAll()
>>> a
? array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

__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.

forall y_i=1 s.t. (h r E_i) in KG

At each mini-batch construction, we subsample(y), hence n
new_y! << **|E|** new_y contains all 1's if sum(y)< neg_sample ratio new_y contains

train_set_idx
Indexed triples for the training.

entity_idxes
mapping.

relation_idxes
mapping.

form
?

store
?

label_smoothing_rate
?

torch.utils.data.Dataset

__len__()

__getitem__(idx)

```
class dicee.NegSampleDataset (train_set: numpy.ndarray, num_entities: int, num_relations: int,
                             neg_sample_ratio: int = 1)
```

Bases: torch.utils.data.Dataset

An abstract class representing a Dataset.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Note: `DataLoader` by default constructs an index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

```
__len__()
```

```
__getitem__(idx)
```

```
class dicee.TriplePredictionDataset (train_set: numpy.ndarray, num_entities: int,
                                     num_relations: int, neg_sample_ratio: int = 1, label_smoothing_rate: float = 0.0)
```

Bases: torch.utils.data.Dataset

Triple Dataset

D:= {(x)_i}_i ^N, where

. x:(h,r, t) in KG is a unique h in E and a relation r in R and . collect_fn => Generates negative triples

collect_fn:

orall (h,r,t) in G obtain, create negative triples{(h,r,x),(r,t),(h,m,t)}

y:labels are represented in torch.float16

train_set_idx

Indexed triples for the training.

entity_idx

mapping.

relation_idx

mapping.

form

?

store

?

label_smoothing_rate

collate_fn: batch:List[torch.IntTensor] Returns —— torch.utils.data.Dataset

```
__len__()
```

```
__getitem__(idx)
```

```
collate_fn (batch: List[torch.Tensor])
```

```
class dicee.CVDataModule(train_set_idx: numpy.ndarray, num_entities, num_relations,  
                        neg_sample_ratio, batch_size, num_workers)
```

Bases: `pytorch_lightning.LightningDataModule`

Create a Dataset for cross validation

Parameters

train_set_idx

Indexed triples for the training.

num_entities

entity to index mapping.

num_relations

relation to index mapping.

batch_size

int

form

?

num_workers

int for <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

Returns

?

train_dataloader () → `torch.utils.data.DataLoader`

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:param-ref:~pytorch_lightning.trainer.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in `prepare_data`

- `fit()`
- `prepare_data()`
- `setup()`

Note:

Lightning tries to add the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

setup (*args, **kwargs)

Called at the beginning of fit (train + validate), validate, test, or predict. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

Args:

stage: either 'fit', 'validate', 'test', or 'predict'

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

        # don't do this
        self.something = else

    def setup(self, stage):
        data = load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

transfer_batch_to_device (*args, **kwargs)

Override this hook if your DataLoader returns tensors wrapped in a custom data structure.

The data types listed below (and any arbitrary nesting of them) are supported out of the box:

- torch.Tensor or anything that implements .to(...)
- list
- dict
- tuple

For anything else, you need to define how the data is moved to the target device (CPU, GPU, TPU, ...).

Note:

This hook should only transfer the data and not modify it, nor should it move the data to any other device than the one passed in as argument (unless you know what you are doing). To check the current state of execution of this hook you can use `self.trainer.training/testing/validating/predicting` so that you can add different logic as per your requirement.

Args:

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:
```

(continues on next page)

(continued from previous page)

```
        # 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
```

Raises:

MisconfigurationException:

If using IPUs, `Trainer(accelerator='ipu')`.

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):
```

(continues on next page)

```

super().__init__()
self.prepare_data_per_node = False

```

This is called before requesting the dataloaders:

```

model.prepare_data()
initialize_distributed()
model.setup(stage)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
model.predict_dataloader()

```

```

class dicee.QueryGenerator (train_path: str, val_path: str, test_path: str, ent2id: Dict = None,
                             rel2id: Dict = None, seed: int = 1, gen_valid: bool = False, gen_test: bool = True)

```

```
list2tuple (list_data)
```

```
tuple2list (x: List | Tuple) → List | Tuple
```

Convert a nested tuple to a nested list.

```
set_global_seed (seed: int)
```

Set seed

```
construct_graph (paths: List[str]) → Tuple[Dict, Dict]
```

Construct graph from triples Returns dicts with incoming and outgoing edges

```
fill_query (query_structure: List[str | List], ent_in: Dict, ent_out: Dict, answer: int) → bool
```

Private method for fill_query logic.

```
achieve_answer (query: List[str | List], ent_in: Dict, ent_out: Dict) → set
```

Private method for achieve_answer logic. @TODO: Document the code

```
write_links (ent_out, small_ent_out)
```

```
ground_queries (query_structure: List[str | List], ent_in: Dict, ent_out: Dict, small_ent_in: Dict,
                 small_ent_out: Dict, gen_num: int, query_name: str)
```

Generating queries and achieving answers

```
unmap (query_type, queries, tp_answers, fp_answers, fn_answers)
```

```
unmap_query (query_structure, query, id2ent, id2rel)
```

```
generate_queries (query_struct: List, gen_num: int, query_type: str)
```

Passing incoming and outgoing edges to ground queries depending on mode [train valid or text] and getting queries and answers in return @ TODO: create a class for each single query struct

```
save_queries (query_type: str, gen_num: int, save_path: str)
```

```
abstract load_queries (path)
```

```
get_queries (query_type: str, gen_num: int)
```

```
static save_queries_and_answers (path: str,
                                 data: List[Tuple[str, Tuple[collections.defaultdict]]]) → None
```

Save Queries into Disk

```
static load_queries_and_answers (path: str)  
    → List[Tuple[str, Tuple[collections.defaultdict]]]  
    Load Queries from Disk to Memory  
  
dicee.__version__ = '0.1.4'
```

1.2 Indices and tables

- `genindex`
- `modindex`
- `search`

Python Module Index

d

- `dicee`, 1
- `dicee.abstracts`, 92
- `dicee.analyse_experiments`, 99
- `dicee.callbacks`, 100
- `dicee.config`, 107
- `dicee.dataset_classes`, 109
- `dicee.eval_static_funcs`, 118
- `dicee.evaluator`, 119
- `dicee.executer`, 121
- `dicee.knowledge_graph`, 123
- `dicee.knowledge_graph_embeddings`, 123
- `dicee.models`, 1
 - `dicee.models.base_model`, 1
 - `dicee.models.clifford`, 10
 - `dicee.models.complex`, 17
 - `dicee.models.function_space`, 19
 - `dicee.models.octonion`, 23
 - `dicee.models.pykeen_models`, 25
 - `dicee.models.quaternion`, 26
 - `dicee.models.real`, 29
 - `dicee.models.static_funcs`, 31
 - `dicee.models.transformers`, 31
- `dicee.query_generator`, 128
- `dicee.read_preprocess_save_load_kg`, 78
- `dicee.read_preprocess_save_load_kg.preprocess`, 78
- `dicee.read_preprocess_save_load_kg.read_from_disk`, 80
- `dicee.read_preprocess_save_load_kg.save_load_disk`, 80
- `dicee.read_preprocess_save_load_kg.util`, 81
- `dicee.sanity_checkers`, 129
- `dicee.scripts`, 85
 - `dicee.scripts.index`, 85
 - `dicee.scripts.run`, 85
 - `dicee.scripts.serve`, 85
- `dicee.static_funcs`, 129
- `dicee.static_funcs_training`, 132
- `dicee.static_preprocess_funcs`, 133
- `dicee.trainer`, 86
 - `dicee.trainer.dice_trainer`, 86
 - `dicee.trainer.torch_trainer`, 88
 - `dicee.trainer.torch_trainer_ddp`, 89

Index

Non-alphabetical

`__getitem__()` (*dicee.AllvsAll method*), 169
`__getitem__()` (*dicee.BPE_NegativeSamplingDataset method*), 166
`__getitem__()` (*dicee.dataset_classes.AllvsAll method*), 113
`__getitem__()` (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 110
`__getitem__()` (*dicee.dataset_classes.KvsAll method*), 112
`__getitem__()` (*dicee.dataset_classes.KvsSampleDataset method*), 114
`__getitem__()` (*dicee.dataset_classes.MultiClassClassificationDataset method*), 111
`__getitem__()` (*dicee.dataset_classes.MultiLabelDataset method*), 111
`__getitem__()` (*dicee.dataset_classes.NegSampleDataset method*), 114
`__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*), 167
`__getitem__()` (*dicee.NegSampleDataset method*), 170
`__getitem__()` (*dicee.OnevsAllDataset method*), 168
`__getitem__()` (*dicee.TriplePredictionDataset method*), 170
`__iter__()` (*dicee.config.Namespace method*), 109
`__len__()` (*dicee.AllvsAll method*), 169
`__len__()` (*dicee.BPE_NegativeSamplingDataset method*), 166
`__len__()` (*dicee.dataset_classes.AllvsAll method*), 113
`__len__()` (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 110
`__len__()` (*dicee.dataset_classes.KvsAll method*), 112
`__len__()` (*dicee.dataset_classes.KvsSampleDataset method*), 114
`__len__()` (*dicee.dataset_classes.MultiClassClassificationDataset method*), 111
`__len__()` (*dicee.dataset_classes.MultiLabelDataset method*), 111
`__len__()` (*dicee.dataset_classes.NegSampleDataset method*), 114
`__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*), 170
`__len__()` (*dicee.OnevsAllDataset method*), 168
`__len__()` (*dicee.TriplePredictionDataset method*), 170
`__str__()` (*dicee.KGE method*), 160
`__str__()` (*dicee.knowledge_graph_embeddings.KGE method*), 124
`__version__` (*in module dicee*), 175

A

`AbstractCallback` (*class in dicee.abstracts*), 96
`AbstractPPECallback` (*class in dicee.abstracts*), 98
`AbstractTrainer` (*class in dicee.abstracts*), 92
`AccumulateEpochLossCallback` (*class in dicee.callbacks*), 100
`achieve_answer()` (*dicee.query_generator.QueryGenerator method*), 128
`achieve_answer()` (*dicee.QueryGenerator method*), 174
`AConEx` (*class in dicee*), 145
`AConEx` (*class in dicee.models*), 51
`AConEx` (*class in dicee.models.complex*), 18
`AConvO` (*class in dicee*), 146
`AConvO` (*class in dicee.models*), 63
`AConvO` (*class in dicee.models.octonion*), 25
`AConvQ` (*class in dicee*), 146
`AConvQ` (*class in dicee.models*), 58
`AConvQ` (*class in dicee.models.quaternion*), 28
`adaptive_swa` (*dicee.config.Namespace attribute*), 109
`add_new_entity_embeddings()` (*dicee.abstracts.BaseInteractiveKGE method*), 95
`add_noise_rate` (*dicee.config.Namespace attribute*), 108
`add_noisy_triples()` (*in module dicee*), 158
`add_noisy_triples()` (*in module dicee.static_funcs*), 131
`add_noisy_triples_into_training()` (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method*), 80

add_noisy_triples_into_training() (*dicee.read_preprocess_save_load_kg.ReadFromDisk method*), 84
 AllvsAll (*class in dicee*), 168
 AllvsAll (*class in dicee.dataset_classes*), 112
 analyse() (*in module dicee.analyse_experiments*), 99
 answer_multi_hop_query() (*dicee.KGE method*), 163
 answer_multi_hop_query() (*dicee.knowledge_graph_embeddings.KGE method*), 126
 app (*in module dicee.scripts.serve*), 86
 apply_coefficients() (*dicee.DeCaL method*), 143
 apply_coefficients() (*dicee.Keci method*), 139
 apply_coefficients() (*dicee.models.clifford.DeCaL method*), 16
 apply_coefficients() (*dicee.models.clifford.Keci method*), 12
 apply_coefficients() (*dicee.models.DeCaL method*), 69
 apply_coefficients() (*dicee.models.Keci method*), 65
 apply_reciprical_or_noise() (*in module dicee.read_preprocess_save_load_kg.util*), 81
 ASWA (*class in dicee.callbacks*), 103

B

backend (*dicee.config.Namespace attribute*), 108
 BaseInteractiveKGE (*class in dicee.abstracts*), 94
 BaseKGE (*class in dicee*), 155
 BaseKGE (*class in dicee.models*), 43, 46, 49, 53, 58, 70, 73
 BaseKGE (*class in dicee.models.base_model*), 7
 BaseKGELightning (*class in dicee.models*), 38
 BaseKGELightning (*class in dicee.models.base_model*), 2
 batch_kronecker_product() (*dicee.callbacks.KronE static method*), 106
 batch_size (*dicee.config.Namespace attribute*), 107
 bias (*dicee.models.transformers.GPTConfig attribute*), 35
 Block (*class in dicee.models.transformers*), 35
 block_size (*dicee.config.Namespace attribute*), 109
 block_size (*dicee.models.transformers.GPTConfig attribute*), 35
 BPE_NegativeSamplingDataset (*class in dicee*), 166
 BPE_NegativeSamplingDataset (*class in dicee.dataset_classes*), 110
 build_chain_funcs() (*dicee.models.FMult2 method*), 76
 build_chain_funcs() (*dicee.models.function_space.FMult2 method*), 21
 build_func() (*dicee.models.FMult2 method*), 76
 build_func() (*dicee.models.function_space.FMult2 method*), 21
 Byte (*class in dicee*), 153
 Byte (*class in dicee.models.transformers*), 31
 byte_pair_encoding (*dicee.config.Namespace attribute*), 109

C

callbacks (*dicee.config.Namespace attribute*), 108
 CausalSelfAttention (*class in dicee.models.transformers*), 33
 chain_func() (*dicee.models.FMult method*), 75
 chain_func() (*dicee.models.function_space.FMult method*), 20
 chain_func() (*dicee.models.function_space.GFMult method*), 20
 chain_func() (*dicee.models.GFMult method*), 76
 cl_pqr() (*dicee.DeCaL method*), 142
 cl_pqr() (*dicee.models.clifford.DeCaL method*), 15
 cl_pqr() (*dicee.models.DeCaL method*), 68
 clifford_mul() (*dicee.CMult method*), 136
 clifford_mul() (*dicee.models.clifford.CMult method*), 10
 clifford_mul() (*dicee.models.CMult method*), 67
 clifford_multiplication() (*dicee.Keci method*), 139
 clifford_multiplication() (*dicee.models.clifford.Keci method*), 12
 clifford_multiplication() (*dicee.models.Keci method*), 65
 CMult (*class in dicee*), 136
 CMult (*class in dicee.models*), 66
 CMult (*class in dicee.models.clifford*), 10
 collate_fn() (*dicee.BPE_NegativeSamplingDataset method*), 166
 collate_fn() (*dicee.dataset_classes.BPE_NegativeSamplingDataset method*), 110
 collate_fn() (*dicee.dataset_classes.TriplePredictionDataset method*), 115
 collate_fn() (*dicee.TriplePredictionDataset method*), 170
 comp_func() (*dicee.LFMult method*), 152
 comp_func() (*dicee.models.function_space.LFMult method*), 22
 comp_func() (*dicee.models.LFMult method*), 78
 Complex (*class in dicee*), 144

Complex (class in *dicee.models*), 52
 Complex (class in *dicee.models.complex*), 18
 compute_convergence() (in module *dicee.callbacks*), 103
 compute_func() (*dicee.models.FMult* method), 75
 compute_func() (*dicee.models.FMult2* method), 76
 compute_func() (*dicee.models.function_space.FMult* method), 20
 compute_func() (*dicee.models.function_space.FMult2* method), 21
 compute_func() (*dicee.models.function_space.GFMult* method), 20
 compute_func() (*dicee.models.GFMult* method), 76
 compute_mrr() (*dicee.callbacks.ASWA* static method), 104
 compute_sigma_pp() (*dicee.DeCaL* method), 143
 compute_sigma_pp() (*dicee.Keci* method), 139
 compute_sigma_pp() (*dicee.models.clifford.DeCaL* method), 16
 compute_sigma_pp() (*dicee.models.clifford.Keci* method), 11
 compute_sigma_pp() (*dicee.models.DeCaL* method), 69
 compute_sigma_pp() (*dicee.models.Keci* method), 64
 compute_sigma_pq() (*dicee.DeCaL* method), 144
 compute_sigma_pq() (*dicee.Keci* method), 139
 compute_sigma_pq() (*dicee.models.clifford.DeCaL* method), 17
 compute_sigma_pq() (*dicee.models.clifford.Keci* method), 12
 compute_sigma_pq() (*dicee.models.DeCaL* method), 70
 compute_sigma_pq() (*dicee.models.Keci* method), 64
 compute_sigma_pr() (*dicee.DeCaL* method), 144
 compute_sigma_pr() (*dicee.models.clifford.DeCaL* method), 17
 compute_sigma_pr() (*dicee.models.DeCaL* method), 70
 compute_sigma_qq() (*dicee.DeCaL* method), 143
 compute_sigma_qq() (*dicee.Keci* method), 139
 compute_sigma_qq() (*dicee.models.clifford.DeCaL* method), 16
 compute_sigma_qq() (*dicee.models.clifford.Keci* method), 12
 compute_sigma_qq() (*dicee.models.DeCaL* method), 70
 compute_sigma_qq() (*dicee.models.Keci* method), 64
 compute_sigma_qr() (*dicee.DeCaL* method), 144
 compute_sigma_qr() (*dicee.models.clifford.DeCaL* method), 17
 compute_sigma_qr() (*dicee.models.DeCaL* method), 70
 compute_sigma_rr() (*dicee.DeCaL* method), 144
 compute_sigma_rr() (*dicee.models.clifford.DeCaL* method), 17
 compute_sigma_rr() (*dicee.models.DeCaL* method), 70
 compute_sigmas_multivect() (*dicee.DeCaL* method), 142
 compute_sigmas_multivect() (*dicee.models.clifford.DeCaL* method), 15
 compute_sigmas_multivect() (*dicee.models.DeCaL* method), 68
 compute_sigmas_single() (*dicee.DeCaL* method), 142
 compute_sigmas_single() (*dicee.models.clifford.DeCaL* method), 15
 compute_sigmas_single() (*dicee.models.DeCaL* method), 68
 ConEx (class in *dicee*), 148
 ConEx (class in *dicee.models*), 51
 ConEx (class in *dicee.models.complex*), 17
 configure_optimizers() (*dicee.models.base_model.BaseKGELightning* method), 6
 configure_optimizers() (*dicee.models.BaseKGELightning* method), 42
 configure_optimizers() (*dicee.models.transformers.GPT* method), 36
 construct_cl_multivector() (*dicee.DeCaL* method), 143
 construct_cl_multivector() (*dicee.Keci* method), 140
 construct_cl_multivector() (*dicee.models.clifford.DeCaL* method), 16
 construct_cl_multivector() (*dicee.models.clifford.Keci* method), 13
 construct_cl_multivector() (*dicee.models.DeCaL* method), 69
 construct_cl_multivector() (*dicee.models.Keci* method), 65
 construct_dataset() (in module *dicee*), 166
 construct_dataset() (in module *dicee.dataset_classes*), 110
 construct_graph() (*dicee.query_generator.QueryGenerator* method), 128
 construct_graph() (*dicee.QueryGenerator* method), 174
 construct_input_and_output() (*dicee.abstracts.BaseInteractiveKGE* method), 96
 construct_multi_coeff() (*dicee.LFMult* method), 152
 construct_multi_coeff() (*dicee.models.function_space.LFMult* method), 22
 construct_multi_coeff() (*dicee.models.LFMult* method), 77
 continual_start() (*dicee.DICE_Trainer* method), 159
 continual_start() (*dicee.executor.ContinuousExecute* method), 122
 continual_start() (*dicee.trainer.DICE_Trainer* method), 91
 continual_start() (*dicee.trainer.dice_trainer.DICE_Trainer* method), 87
 continual_training_setup_executor() (in module *dicee*), 158

continual_training_setup_executor() (in module *dicee.static_funcs*), 132
 ContinuousExecute (class in *dicee.executer*), 122
 ConvO (class in *dicee*), 147
 ConvO (class in *dicee.models*), 62
 ConvO (class in *dicee.models.octonion*), 24
 ConvQ (class in *dicee*), 147
 ConvQ (class in *dicee.models*), 57
 ConvQ (class in *dicee.models.quaternion*), 28
 create_constraints() (in module *dicee.read_preprocess_save_load_kg.util*), 82
 create_constraints() (in module *dicee.static_preprocess_funcs*), 133
 create_experiment_folder() (in module *dicee*), 158
 create_experiment_folder() (in module *dicee.static_funcs*), 132
 create_random_data() (*dicee.callbacks.PseudoLabellingCallback* method), 103
 create_recipriocal_triples() (in module *dicee*), 157
 create_recipriocal_triples() (in module *dicee.read_preprocess_save_load_kg.util*), 82
 create_recipriocal_triples() (in module *dicee.static_funcs*), 130
 create_vector_database() (*dicee.KGE* method), 160
 create_vector_database() (*dicee.knowledge_graph_embeddings.KGE* method), 123
 crop_block_size() (*dicee.models.transformers.GPT* method), 36
 CVDDataModule (class in *dicee*), 170
 CVDDataModule (class in *dicee.dataset_classes*), 115

D

dataset_dir (*dicee.config.Namespace* attribute), 107
 dataset_sanity_checking() (in module *dicee.read_preprocess_save_load_kg.util*), 82
 DDPTTrainer (class in *dicee.trainer.torch_trainer_ddp*), 90
 DeCaL (class in *dicee*), 141
 DeCaL (class in *dicee.models*), 67
 DeCaL (class in *dicee.models.clifford*), 14
 decide() (*dicee.callbacks.ASWA* method), 104
 deploy() (*dicee.KGE* method), 164
 deploy() (*dicee.knowledge_graph_embeddings.KGE* method), 127
 deploy_head_entity_prediction() (in module *dicee*), 158
 deploy_head_entity_prediction() (in module *dicee.static_funcs*), 131
 deploy_relation_prediction() (in module *dicee*), 158
 deploy_relation_prediction() (in module *dicee.static_funcs*), 131
 deploy_tail_entity_prediction() (in module *dicee*), 158
 deploy_tail_entity_prediction() (in module *dicee.static_funcs*), 131
 deploy_triple_prediction() (in module *dicee*), 158
 deploy_triple_prediction() (in module *dicee.static_funcs*), 131
 DICE_Trainer (class in *dicee*), 159
 DICE_Trainer (class in *dicee.trainer*), 90
 DICE_Trainer (class in *dicee.trainer.dice_trainer*), 87
 dicee
 module, 1
 dicee.abstracts
 module, 92
 dicee.analyse_experiments
 module, 99
 dicee.callbacks
 module, 100
 dicee.config
 module, 107
 dicee.dataset_classes
 module, 109
 dicee.eval_static_funcs
 module, 118
 dicee.evaluator
 module, 119
 dicee.executer
 module, 121
 dicee.knowledge_graph
 module, 123
 dicee.knowledge_graph_embeddings
 module, 123
 dicee.models
 module, 1

- dicee.models.base_model
 - module, 1
- dicee.models.clifford
 - module, 10
- dicee.models.complex
 - module, 17
- dicee.models.function_space
 - module, 19
- dicee.models.octonion
 - module, 23
- dicee.models.pykeen_models
 - module, 25
- dicee.models.quaternion
 - module, 26
- dicee.models.real
 - module, 29
- dicee.models.static_funcs
 - module, 31
- dicee.models.transformers
 - module, 31
- dicee.query_generator
 - module, 128
- dicee.read_preprocess_save_load_kg
 - module, 78
- dicee.read_preprocess_save_load_kg.preprocess
 - module, 78
- dicee.read_preprocess_save_load_kg.read_from_disk
 - module, 80
- dicee.read_preprocess_save_load_kg.save_load_disk
 - module, 80
- dicee.read_preprocess_save_load_kg.util
 - module, 81
- dicee.sanity_checkers
 - module, 129
- dicee.scripts
 - module, 85
- dicee.scripts.index
 - module, 85
- dicee.scripts.run
 - module, 85
- dicee.scripts.serve
 - module, 85
- dicee.static_funcs
 - module, 129
- dicee.static_funcs_training
 - module, 132
- dicee.static_preprocess_funcs
 - module, 133
- dicee.trainer
 - module, 86
- dicee.trainer.dice_trainer
 - module, 86
- dicee.trainer.torch_trainer
 - module, 88
- dicee.trainer.torch_trainer_ddp
 - module, 89
- DistMult (*class in dicee*), 138
- DistMult (*class in dicee.models*), 48
- DistMult (*class in dicee.models.real*), 29
- download_file() (*in module dicee*), 159
- download_file() (*in module dicee.static_funcs*), 132
- download_files_from_url() (*in module dicee*), 159
- 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*), 132
- dropout (*dicee.models.transformers.GPTConfig attribute*), 35
- dummy_eval() (*dicee.evaluator.Evaluator method*), 120

E

`efficient_zero_grad()` (in module `dicee.static_funcs_training`), 132
`embedding_dim` (`dicee.config.Namespace` attribute), 107
`enable_log` (in module `dicee.static_preprocess_funcs`), 133
`end()` (`dicee.Execute` method), 165
`end()` (`dicee.executor.Execute` method), 122
`entities_str` (`dicee.knowledge_graph.KG` property), 123
`estimate_mfu()` (`dicee.models.transformers.GPT` method), 36
`estimate_q()` (in module `dicee.callbacks`), 103
`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), 108
`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), 120
`eval_with_vs_all()` (`dicee.evaluator.Evaluator` method), 120
`evaluate()` (in module `dicee`), 159
`evaluate()` (in module `dicee.static_funcs`), 132
`evaluate_bpe_lp()` (in module `dicee.static_funcs_training`), 132
`evaluate_link_prediction_performance()` (in module `dicee.eval_static_funcs`), 118
`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), 120
`evaluate_lp()` (in module `dicee.static_funcs_training`), 132
`evaluate_lp_bpe_k_vs_all()` (`dicee.evaluator.Evaluator` method), 120
`evaluate_lp_bpe_k_vs_all()` (in module `dicee.eval_static_funcs`), 119
`evaluate_lp_k_vs_all()` (`dicee.evaluator.Evaluator` method), 120
`evaluate_lp_with_byte()` (`dicee.evaluator.Evaluator` method), 120
`Evaluator` (class in `dicee.evaluator`), 119
`Execute` (class in `dicee`), 164
`Execute` (class in `dicee.executor`), 121
`Experiment` (class in `dicee.analyse_experiments`), 99
`exponential_function()` (in module `dicee`), 158
`exponential_function()` (in module `dicee.static_funcs`), 132
`extract_input_outputs()` (`dicee.trainer.torch_trainer_ddp.DDPTrainer` method), 90
`extract_input_outputs()` (`dicee.trainer.torch_trainer_ddp.NodeTrainer` method), 90
`extract_input_outputs_set_device()` (`dicee.trainer.torch_trainer.TorchTrainer` method), 89

F

`feature_map_dropout_rate` (`dicee.config.Namespace` attribute), 109
`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), 90
`fit()` (`dicee.trainer.torch_trainer.TorchTrainer` method), 88
`FMult` (class in `dicee.models`), 75
`FMult` (class in `dicee.models.function_space`), 20
`FMult2` (class in `dicee.models`), 76
`FMult2` (class in `dicee.models.function_space`), 21
`forward()` (`dicee.BaseKGE` method), 156
`forward()` (`dicee.BytE` method), 154
`forward()` (`dicee.models.base_model.BaseKGE` method), 8
`forward()` (`dicee.models.base_model.IdentityClass` static method), 9
`forward()` (`dicee.models.BaseKGE` method), 44, 47, 50, 54, 59, 71, 74
`forward()` (`dicee.models.IdentityClass` static method), 46, 56, 61
`forward()` (`dicee.models.transformers.Block` method), 35
`forward()` (`dicee.models.transformers.BytE` method), 32
`forward()` (`dicee.models.transformers.CausalSelfAttention` method), 34
`forward()` (`dicee.models.transformers.GPT` method), 36
`forward()` (`dicee.models.transformers.LayerNorm` method), 33
`forward()` (`dicee.models.transformers.MLP` method), 35

`forward_backward_update()` (*dicee.trainer.torch_trainer.TorchTrainer method*), 89
`forward_byte_pair_encoded_k_vs_all()` (*dicee.BaseKGE method*), 156
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.base_model.BaseKGE method*), 7
`forward_byte_pair_encoded_k_vs_all()` (*dicee.models.BaseKGE method*), 44, 46, 50, 53, 59, 71, 74
`forward_byte_pair_encoded_triple()` (*dicee.BaseKGE method*), 156
`forward_byte_pair_encoded_triple()` (*dicee.models.base_model.BaseKGE method*), 8
`forward_byte_pair_encoded_triple()` (*dicee.models.BaseKGE method*), 44, 46, 50, 54, 59, 71, 74
`forward_k_vs_all()` (*dicee.AConEx method*), 145
`forward_k_vs_all()` (*dicee.AConvO method*), 146
`forward_k_vs_all()` (*dicee.AConvQ method*), 147
`forward_k_vs_all()` (*dicee.BaseKGE method*), 157
`forward_k_vs_all()` (*dicee.CMult method*), 137
`forward_k_vs_all()` (*dicee.ComplEx method*), 145
`forward_k_vs_all()` (*dicee.ConEx method*), 148
`forward_k_vs_all()` (*dicee.ConvO method*), 148
`forward_k_vs_all()` (*dicee.ConvQ method*), 147
`forward_k_vs_all()` (*dicee.DeCaL method*), 143
`forward_k_vs_all()` (*dicee.DistMult method*), 138
`forward_k_vs_all()` (*dicee.Keci method*), 140
`forward_k_vs_all()` (*dicee.models.AConEx method*), 52
`forward_k_vs_all()` (*dicee.models.AConvO method*), 63
`forward_k_vs_all()` (*dicee.models.AConvQ method*), 58
`forward_k_vs_all()` (*dicee.models.base_model.BaseKGE method*), 8
`forward_k_vs_all()` (*dicee.models.BaseKGE method*), 44, 47, 50, 54, 60, 72, 75
`forward_k_vs_all()` (*dicee.models.clifford.CMult method*), 11
`forward_k_vs_all()` (*dicee.models.clifford.DeCaL method*), 15
`forward_k_vs_all()` (*dicee.models.clifford.Keci method*), 13
`forward_k_vs_all()` (*dicee.models.CMult method*), 67
`forward_k_vs_all()` (*dicee.models.ComplEx method*), 53
`forward_k_vs_all()` (*dicee.models.complex.AConEx method*), 18
`forward_k_vs_all()` (*dicee.models.complex.ComplEx method*), 19
`forward_k_vs_all()` (*dicee.models.complex.ConEx method*), 18
`forward_k_vs_all()` (*dicee.models.ConEx method*), 51
`forward_k_vs_all()` (*dicee.models.ConvO method*), 63
`forward_k_vs_all()` (*dicee.models.ConvQ method*), 58
`forward_k_vs_all()` (*dicee.models.DeCaL method*), 69
`forward_k_vs_all()` (*dicee.models.DistMult method*), 48
`forward_k_vs_all()` (*dicee.models.Keci method*), 65
`forward_k_vs_all()` (*dicee.models.octonion.AConvO method*), 25
`forward_k_vs_all()` (*dicee.models.octonion.ConvO method*), 25
`forward_k_vs_all()` (*dicee.models.octonion.OMult method*), 24
`forward_k_vs_all()` (*dicee.models.OMult method*), 62
`forward_k_vs_all()` (*dicee.models.pykeen_models.PykeenKGE method*), 25
`forward_k_vs_all()` (*dicee.models.PykeenKGE method*), 73
`forward_k_vs_all()` (*dicee.models.QMult method*), 57
`forward_k_vs_all()` (*dicee.models.quaternion.AConvQ method*), 29
`forward_k_vs_all()` (*dicee.models.quaternion.ConvQ method*), 28
`forward_k_vs_all()` (*dicee.models.quaternion.QMult method*), 28
`forward_k_vs_all()` (*dicee.models.real.DistMult method*), 30
`forward_k_vs_all()` (*dicee.models.real.Shallom method*), 30
`forward_k_vs_all()` (*dicee.models.real.TransE method*), 30
`forward_k_vs_all()` (*dicee.models.Shallom method*), 49
`forward_k_vs_all()` (*dicee.models.TransE method*), 48
`forward_k_vs_all()` (*dicee.OMult method*), 151
`forward_k_vs_all()` (*dicee.PykeenKGE method*), 153
`forward_k_vs_all()` (*dicee.QMult method*), 150
`forward_k_vs_all()` (*dicee.Shallom method*), 151
`forward_k_vs_all()` (*dicee.TransE method*), 141
`forward_k_vs_sample()` (*dicee.AConEx method*), 146
`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*), 140
`forward_k_vs_sample()` (*dicee.models.AConEx method*), 52
`forward_k_vs_sample()` (*dicee.models.base_model.BaseKGE method*), 8
`forward_k_vs_sample()` (*dicee.models.BaseKGE method*), 44, 47, 50, 54, 60, 72, 75
`forward_k_vs_sample()` (*dicee.models.clifford.Keci method*), 13
`forward_k_vs_sample()` (*dicee.models.complex.AConEx method*), 18

`forward_k_vs_sample()` (*dicee.models.complex.ConEx method*), 18
`forward_k_vs_sample()` (*dicee.models.ConEx method*), 51
`forward_k_vs_sample()` (*dicee.models.DistMult method*), 48
`forward_k_vs_sample()` (*dicee.models.Keci method*), 65
`forward_k_vs_sample()` (*dicee.models.pykeen_models.PykeenKGE method*), 26
`forward_k_vs_sample()` (*dicee.models.PykeenKGE method*), 73
`forward_k_vs_sample()` (*dicee.models.QMult method*), 57
`forward_k_vs_sample()` (*dicee.models.quaternion.QMult method*), 28
`forward_k_vs_sample()` (*dicee.models.real.DistMult method*), 30
`forward_k_vs_sample()` (*dicee.PykeenKGE method*), 153
`forward_k_vs_sample()` (*dicee.QMult method*), 150
`forward_k_vs_with_explicit()` (*dicee.Keci method*), 140
`forward_k_vs_with_explicit()` (*dicee.models.clifford.Keci method*), 13
`forward_k_vs_with_explicit()` (*dicee.models.Keci method*), 65
`forward_triples()` (*dicee.AConEx method*), 145
`forward_triples()` (*dicee.AConvO method*), 146
`forward_triples()` (*dicee.AConvQ method*), 146
`forward_triples()` (*dicee.BaseKGE method*), 156
`forward_triples()` (*dicee.CMult method*), 137
`forward_triples()` (*dicee.ConEx method*), 148
`forward_triples()` (*dicee.ConvO method*), 148
`forward_triples()` (*dicee.ConvQ method*), 147
`forward_triples()` (*dicee.DeCaL method*), 142
`forward_triples()` (*dicee.Keci method*), 141
`forward_triples()` (*dicee.LFMMult method*), 151
`forward_triples()` (*dicee.models.AConEx method*), 52
`forward_triples()` (*dicee.models.AConvO method*), 63
`forward_triples()` (*dicee.models.AConvQ method*), 58
`forward_triples()` (*dicee.models.base_model.BaseKGE method*), 8
`forward_triples()` (*dicee.models.BaseKGE method*), 44, 47, 50, 54, 59, 72, 74
`forward_triples()` (*dicee.models.clifford.CMult method*), 10
`forward_triples()` (*dicee.models.clifford.DeCaL method*), 15
`forward_triples()` (*dicee.models.clifford.Keci method*), 14
`forward_triples()` (*dicee.models.CMult method*), 67
`forward_triples()` (*dicee.models.complex.AConEx method*), 18
`forward_triples()` (*dicee.models.complex.ConEx method*), 18
`forward_triples()` (*dicee.models.ConEx method*), 51
`forward_triples()` (*dicee.models.ConvO method*), 62
`forward_triples()` (*dicee.models.ConvQ method*), 57
`forward_triples()` (*dicee.models.DeCaL method*), 68
`forward_triples()` (*dicee.models.FMMult method*), 75
`forward_triples()` (*dicee.models.FMMult2 method*), 76
`forward_triples()` (*dicee.models.function_space.FMMult method*), 20
`forward_triples()` (*dicee.models.function_space.FMMult2 method*), 21
`forward_triples()` (*dicee.models.function_space.GFMMult method*), 20
`forward_triples()` (*dicee.models.function_space.LFMMult method*), 21
`forward_triples()` (*dicee.models.function_space.LFMMult1 method*), 21
`forward_triples()` (*dicee.models.GFMMult method*), 76
`forward_triples()` (*dicee.models.Keci method*), 66
`forward_triples()` (*dicee.models.LFMMult method*), 77
`forward_triples()` (*dicee.models.LFMMult1 method*), 77
`forward_triples()` (*dicee.models.octonion.AConvO method*), 25
`forward_triples()` (*dicee.models.octonion.ConvO method*), 24
`forward_triples()` (*dicee.models.Pyke method*), 49
`forward_triples()` (*dicee.models.pykeen_models.PykeenKGE method*), 26
`forward_triples()` (*dicee.models.PykeenKGE method*), 73
`forward_triples()` (*dicee.models.quaternion.AConvQ method*), 29
`forward_triples()` (*dicee.models.quaternion.ConvQ method*), 28
`forward_triples()` (*dicee.models.real.Pyke method*), 30
`forward_triples()` (*dicee.models.real.Shallom method*), 30
`forward_triples()` (*dicee.models.Shallom method*), 49
`forward_triples()` (*dicee.Pyke method*), 137
`forward_triples()` (*dicee.PykeenKGE method*), 153
`forward_triples()` (*dicee.Shallom method*), 151
`from_pretrained()` (*dicee.models.transformers.GPT class method*), 36
`func_triple_to_bpe_representation()` (*dicee.knowledge_graph.KG method*), 123
`function()` (*dicee.models.FMMult2 method*), 76
`function()` (*dicee.models.function_space.FMMult2 method*), 21

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*), 32
`generate_queries()` (*dicee.query_generator.QueryGenerator method*), 128
`generate_queries()` (*dicee.QueryGenerator method*), 174
`get()` (*dicee.scripts.serve.NeuralSearcher method*), 86
`get_aswa_state_dict()` (*dicee.callbacks.ASWA method*), 104
`get_bpe_head_and_relation_representation()` (*dicee.BaseKGE method*), 157
`get_bpe_head_and_relation_representation()` (*dicee.models.base_model.BaseKGE method*), 9
`get_bpe_head_and_relation_representation()` (*dicee.models.BaseKGE method*), 45, 47, 51, 55, 60, 72, 75
`get_bpe_token_representation()` (*dicee.abstracts.BaseInteractiveKGE method*), 94
`get_callbacks()` (*in module dicee.trainer.dice_trainer*), 87
`get_default_arguments()` (*in module dicee.analyse_experiments*), 99
`get_default_arguments()` (*in module dicee.scripts.index*), 85
`get_default_arguments()` (*in module dicee.scripts.run*), 85
`get_default_arguments()` (*in module dicee.scripts.serve*), 86
`get_domain_of_relation()` (*dicee.abstracts.BaseInteractiveKGE method*), 94
`get_ee_vocab()` (*in module dicee*), 157
`get_ee_vocab()` (*in module dicee.read_preprocess_save_load_kg.util*), 82
`get_ee_vocab()` (*in module dicee.static_funcs*), 130
`get_ee_vocab()` (*in module dicee.static_preprocess_funcs*), 133
`get_embeddings()` (*dicee.BaseKGE method*), 157
`get_embeddings()` (*dicee.models.base_model.BaseKGE method*), 9
`get_embeddings()` (*dicee.models.BaseKGE method*), 45, 48, 51, 55, 60, 72, 75
`get_embeddings()` (*dicee.models.real.Shallom method*), 30
`get_embeddings()` (*dicee.models.Shallom method*), 48
`get_embeddings()` (*dicee.Shallom method*), 151
`get_entity_embeddings()` (*dicee.abstracts.BaseInteractiveKGE method*), 95
`get_entity_index()` (*dicee.abstracts.BaseInteractiveKGE method*), 95
`get_er_vocab()` (*in module dicee*), 157
`get_er_vocab()` (*in module dicee.read_preprocess_save_load_kg.util*), 82
`get_er_vocab()` (*in module dicee.static_funcs*), 130
`get_er_vocab()` (*in module dicee.static_preprocess_funcs*), 133
`get_eval_report()` (*dicee.abstracts.BaseInteractiveKGE method*), 94
`get_head_relation_representation()` (*dicee.BaseKGE method*), 157
`get_head_relation_representation()` (*dicee.models.base_model.BaseKGE method*), 8
`get_head_relation_representation()` (*dicee.models.BaseKGE method*), 44, 47, 50, 54, 60, 72, 75
`get_kronecker_triple_representation()` (*dicee.callbacks.KronE method*), 106
`get_num_params()` (*dicee.models.transformers.GPT method*), 36
`get_padded_bpe_triple_representation()` (*dicee.abstracts.BaseInteractiveKGE method*), 94
`get_queries()` (*dicee.query_generator.QueryGenerator method*), 128
`get_queries()` (*dicee.QueryGenerator method*), 174
`get_range_of_relation()` (*dicee.abstracts.BaseInteractiveKGE method*), 94
`get_re_vocab()` (*in module dicee*), 157
`get_re_vocab()` (*in module dicee.read_preprocess_save_load_kg.util*), 82
`get_re_vocab()` (*in module dicee.static_funcs*), 130
`get_re_vocab()` (*in module dicee.static_preprocess_funcs*), 133
`get_relation_embeddings()` (*dicee.abstracts.BaseInteractiveKGE method*), 96
`get_relation_index()` (*dicee.abstracts.BaseInteractiveKGE method*), 95
`get_sentence_representation()` (*dicee.BaseKGE method*), 157
`get_sentence_representation()` (*dicee.models.base_model.BaseKGE method*), 8
`get_sentence_representation()` (*dicee.models.BaseKGE method*), 44, 47, 50, 54, 60, 72, 75
`get_transductive_entity_embeddings()` (*dicee.KGE method*), 160
`get_transductive_entity_embeddings()` (*dicee.knowledge_graph_embeddings.KGE method*), 123
`get_triple_representation()` (*dicee.BaseKGE method*), 157
`get_triple_representation()` (*dicee.models.base_model.BaseKGE method*), 8
`get_triple_representation()` (*dicee.models.BaseKGE method*), 44, 47, 50, 54, 60, 72, 75
`GFMult` (*class in dicee.models*), 76
`GFMult` (*class in dicee.models.function_space*), 20
`GPT` (*class in dicee.models.transformers*), 35
`GPTConfig` (*class in dicee.models.transformers*), 35
`gpus` (*dicee.config.Namespace attribute*), 108
`gradient_accumulation_steps` (*dicee.config.Namespace attribute*), 108
`ground_queries()` (*dicee.query_generator.QueryGenerator method*), 128
`ground_queries()` (*dicee.QueryGenerator method*), 174

H

`hidden_dropout_rate` (*dicee.config.Namespace* attribute), 109

I

`IdentityClass` (class in *dicee.models*), 45, 55, 60
`IdentityClass` (class in *dicee.models.base_model*), 9
`index_triple`() (*dicee.abstracts.BaseInteractiveKGE* method), 95
`index_triples_with_pandas`() (in module *dicee.read_preprocess_save_load_kg.util*), 82
`init_param` (*dicee.config.Namespace* attribute), 108
`init_params_with_sanity_checking`() (*dicee.BaseKGE* method), 156
`init_params_with_sanity_checking`() (*dicee.models.base_model.BaseKGE* method), 8
`init_params_with_sanity_checking`() (*dicee.models.BaseKGE* method), 44, 47, 50, 54, 59, 71, 74
`initialize_dataloader`() (*dicee.DICE_Trainer* method), 159
`initialize_dataloader`() (*dicee.trainer.DICE_Trainer* method), 91
`initialize_dataloader`() (*dicee.trainer.dice_trainer.DICE_Trainer* method), 87
`initialize_dataset`() (*dicee.DICE_Trainer* method), 159
`initialize_dataset`() (*dicee.trainer.DICE_Trainer* method), 91
`initialize_dataset`() (*dicee.trainer.dice_trainer.DICE_Trainer* method), 87
`initialize_or_load_model`() (*dicee.DICE_Trainer* method), 159
`initialize_or_load_model`() (*dicee.trainer.DICE_Trainer* method), 91
`initialize_or_load_model`() (*dicee.trainer.dice_trainer.DICE_Trainer* method), 87
`initialize_trainer`() (*dicee.DICE_Trainer* method), 159
`initialize_trainer`() (*dicee.trainer.DICE_Trainer* method), 91
`initialize_trainer`() (*dicee.trainer.dice_trainer.DICE_Trainer* method), 87
`initialize_trainer`() (in module *dicee.trainer.dice_trainer*), 87
`input_dropout_rate` (*dicee.config.Namespace* attribute), 109
`intialize_model`() (in module *dicee*), 158
`intialize_model`() (in module *dicee.static_funcs*), 131
`is_seen`() (*dicee.abstracts.BaseInteractiveKGE* method), 95
`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), 91
`k_fold_cross_validation`() (*dicee.trainer.dice_trainer.DICE_Trainer* method), 88
`k_vs_all_score`() (*dicee.ComplEx* static method), 145
`k_vs_all_score`() (*dicee.DistMult* method), 138
`k_vs_all_score`() (*dicee.Keci* method), 140
`k_vs_all_score`() (*dicee.models.clifford.Keci* method), 13
`k_vs_all_score`() (*dicee.models.ComplEx* static method), 52
`k_vs_all_score`() (*dicee.models.complex.ComplEx* static method), 19
`k_vs_all_score`() (*dicee.models.DistMult* method), 48
`k_vs_all_score`() (*dicee.models.Keci* method), 65
`k_vs_all_score`() (*dicee.models.octonion.OMult* method), 24
`k_vs_all_score`() (*dicee.models.OMult* method), 62
`k_vs_all_score`() (*dicee.models.QMult* method), 57
`k_vs_all_score`() (*dicee.models.quaternion.QMult* method), 28
`k_vs_all_score`() (*dicee.models.real.DistMult* method), 29
`k_vs_all_score`() (*dicee.OMult* method), 151
`k_vs_all_score`() (*dicee.QMult* method), 150
`Keci` (class in *dicee*), 138
`Keci` (class in *dicee.models*), 63
`Keci` (class in *dicee.models.clifford*), 11
`KeciBase` (class in *dicee*), 138
`KeciBase` (class in *dicee.models*), 66
`KeciBase` (class in *dicee.models.clifford*), 14
`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*), 123
`KGESaveCallback` (class in *dicee.callbacks*), 102
`KronE` (class in *dicee.callbacks*), 106
`KvsAll` (class in *dicee*), 168
`KvsAll` (class in *dicee.dataset_classes*), 112
`KvsSampleDataset` (class in *dicee*), 169
`KvsSampleDataset` (class in *dicee.dataset_classes*), 113

L

LayerNorm (class in *dicee.models.transformers*), 33
LFMult (class in *dicee*), 151
LFMult (class in *dicee.models*), 77
LFMult (class in *dicee.models.function_space*), 21
LFMult1 (class in *dicee.models*), 77
LFMult1 (class in *dicee.models.function_space*), 21
linear () (*dicee.LFMult* method), 152
linear () (*dicee.models.function_space.LFMult* method), 22
linear () (*dicee.models.LFMult* method), 77
list2tuple () (*dicee.query_generator.QueryGenerator* method), 128
list2tuple () (*dicee.QueryGenerator* method), 174
load () (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk* method), 84
load () (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk* method), 80
load_indexed_data () (*dicee.Execute* method), 164
load_indexed_data () (*dicee.executer.Execute* method), 121
load_json () (in module *dicee*), 158
load_json () (in module *dicee.static_funcs*), 131
load_model () (in module *dicee*), 157
load_model () (in module *dicee.static_funcs*), 131
load_model_ensemble () (in module *dicee*), 158
load_model_ensemble () (in module *dicee.static_funcs*), 131
load_numpy () (in module *dicee*), 159
load_numpy () (in module *dicee.static_funcs*), 132
load_numpy_ndarray () (in module *dicee.read_preprocess_save_load_kg.util*), 82
load_pickle () (in module *dicee*), 157, 166
load_pickle () (in module *dicee.read_preprocess_save_load_kg.util*), 82
load_pickle () (in module *dicee.static_funcs*), 130
load_queries () (*dicee.query_generator.QueryGenerator* method), 128
load_queries () (*dicee.QueryGenerator* method), 174
load_queries_and_answers () (*dicee.query_generator.QueryGenerator* static method), 128
load_queries_and_answers () (*dicee.QueryGenerator* static method), 174
load_with_pandas () (in module *dicee.read_preprocess_save_load_kg.util*), 82
LoadSaveToDisk (class in *dicee.read_preprocess_save_load_kg*), 84
LoadSaveToDisk (class in *dicee.read_preprocess_save_load_kg.save_load_disk*), 80
loss_function () (*dicee.BytE* method), 154
loss_function () (*dicee.models.base_model.BaseKGELightning* method), 3
loss_function () (*dicee.models.BaseKGELightning* method), 39
loss_function () (*dicee.models.transformers.BytE* method), 32
lr (*dicee.config.Namespace* attribute), 108

M

main () (in module *dicee.scripts.index*), 85
main () (in module *dicee.scripts.run*), 85
main () (in module *dicee.scripts.serve*), 86
mapping_from_first_two_cols_to_third () (in module *dicee*), 166
mapping_from_first_two_cols_to_third () (in module *dicee.static_preprocess_funcs*), 133
mem_of_model () (*dicee.models.base_model.BaseKGELightning* method), 2
mem_of_model () (*dicee.models.BaseKGELightning* method), 38
MLP (class in *dicee.models.transformers*), 34
model (*dicee.config.Namespace* attribute), 107
module
 dicee, 1
 dicee.abstracts, 92
 dicee.analyse_experiments, 99
 dicee.callbacks, 100
 dicee.config, 107
 dicee.dataset_classes, 109
 dicee.eval_static_funcs, 118
 dicee.evaluator, 119
 dicee.executer, 121
 dicee.knowledge_graph, 123
 dicee.knowledge_graph_embeddings, 123
 dicee.models, 1
 dicee.models.base_model, 1
 dicee.models.clifford, 10
 dicee.models.complex, 17

- `dicee.models.function_space`, 19
- `dicee.models.octonion`, 23
- `dicee.models.pykeen_models`, 25
- `dicee.models.quaternion`, 26
- `dicee.models.real`, 29
- `dicee.models.static_funcs`, 31
- `dicee.models.transformers`, 31
- `dicee.query_generator`, 128
- `dicee.read_preprocess_save_load_kg`, 78
- `dicee.read_preprocess_save_load_kg.preprocess`, 78
- `dicee.read_preprocess_save_load_kg.read_from_disk`, 80
- `dicee.read_preprocess_save_load_kg.save_load_disk`, 80
- `dicee.read_preprocess_save_load_kg.util`, 81
- `dicee.sanity_checkers`, 129
- `dicee.scripts`, 85
- `dicee.scripts.index`, 85
- `dicee.scripts.run`, 85
- `dicee.scripts.serve`, 85
- `dicee.static_funcs`, 129
- `dicee.static_funcs_training`, 132
- `dicee.static_preprocess_funcs`, 133
- `dicee.trainer`, 86
- `dicee.trainer.dice_trainer`, 86
- `dicee.trainer.torch_trainer`, 88
- `dicee.trainer.torch_trainer_ddp`, 89
- `MultiClassClassificationDataset` (class in *dicee*), 167
- `MultiClassClassificationDataset` (class in *dicee.dataset_classes*), 111
- `MultiLabelDataset` (class in *dicee*), 166
- `MultiLabelDataset` (class in *dicee.dataset_classes*), 111

N

- `n_embd` (*dicee.models.transformers.GPTConfig* attribute), 35
- `n_head` (*dicee.models.transformers.GPTConfig* attribute), 35
- `n_layer` (*dicee.models.transformers.GPTConfig* attribute), 35
- `name` (*dicee.abstracts.BaseInteractiveKGE* property), 94
- `Namespace` (class in *dicee.config*), 107
- `neg_ratio` (*dicee.config.Namespace* attribute), 108
- `negnorm()` (*dicee.KGE* method), 163
- `negnorm()` (*dicee.knowledge_graph_embeddings.KGE* method), 126
- `NegSampleDataset` (class in *dicee*), 169
- `NegSampleDataset` (class in *dicee.dataset_classes*), 114
- `neural_searcher` (in module *dicee.scripts.serve*), 86
- `NeuralSearcher` (class in *dicee.scripts.serve*), 86
- `NodeTrainer` (class in *dicee.trainer.torch_trainer_ddp*), 90
- `normalization` (*dicee.config.Namespace* attribute), 108
- `num_core` (*dicee.config.Namespace* attribute), 108
- `num_epochs` (*dicee.config.Namespace* attribute), 107
- `num_folds_for_cv` (*dicee.config.Namespace* attribute), 108
- `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*), 61
- `octonion_mul()` (in module *dicee.models.octonion*), 23
- `octonion_mul_norm()` (in module *dicee.models*), 61
- `octonion_mul_norm()` (in module *dicee.models.octonion*), 23
- `octonion_normalizer()` (*dicee.AConvO* static method), 146
- `octonion_normalizer()` (*dicee.ConvO* static method), 148
- `octonion_normalizer()` (*dicee.models.AConvO* static method), 63
- `octonion_normalizer()` (*dicee.models.ConvO* static method), 62
- `octonion_normalizer()` (*dicee.models.octonion.AConvO* static method), 25
- `octonion_normalizer()` (*dicee.models.octonion.ConvO* static method), 24
- `octonion_normalizer()` (*dicee.models.octonion.OMult* static method), 24
- `octonion_normalizer()` (*dicee.models.OMult* static method), 61
- `octonion_normalizer()` (*dicee.OMult* static method), 151
- `OMult` (class in *dicee*), 150

OMult (class in *dicее.models*), 61
 OMult (class in *dicее.models.octonion*), 23
 on_epoch_end() (*dicее.callbacks.KGESaveCallback* method), 103
 on_epoch_end() (*dicее.callbacks.PseudoLabellingCallback* method), 103
 on_fit_end() (*dicее.abstracts.AbstractCallback* method), 98
 on_fit_end() (*dicее.abstracts.AbstractPPECallback* method), 98
 on_fit_end() (*dicее.abstracts.AbstractTrainer* method), 92
 on_fit_end() (*dicее.callbacks.AccumulateEpochLossCallback* method), 100
 on_fit_end() (*dicее.callbacks.ASWA* method), 104
 on_fit_end() (*dicее.callbacks.Eval* method), 105
 on_fit_end() (*dicее.callbacks.KGESaveCallback* method), 103
 on_fit_end() (*dicее.callbacks.PrintCallback* method), 101
 on_fit_start() (*dicее.abstracts.AbstractCallback* method), 97
 on_fit_start() (*dicее.abstracts.AbstractPPECallback* method), 98
 on_fit_start() (*dicее.abstracts.AbstractTrainer* method), 92
 on_fit_start() (*dicее.callbacks.Eval* method), 105
 on_fit_start() (*dicее.callbacks.KGESaveCallback* method), 102
 on_fit_start() (*dicее.callbacks.KronE* method), 106
 on_fit_start() (*dicее.callbacks.PrintCallback* method), 101
 on_init_end() (*dicее.abstracts.AbstractCallback* method), 96
 on_init_start() (*dicее.abstracts.AbstractCallback* method), 96
 on_train_batch_end() (*dicее.abstracts.AbstractCallback* method), 97
 on_train_batch_end() (*dicее.abstracts.AbstractTrainer* method), 93
 on_train_batch_end() (*dicее.callbacks.Eval* method), 106
 on_train_batch_end() (*dicее.callbacks.KGESaveCallback* method), 102
 on_train_batch_end() (*dicее.callbacks.PrintCallback* method), 101
 on_train_batch_start() (*dicее.callbacks.Perturb* method), 107
 on_train_epoch_end() (*dicее.abstracts.AbstractCallback* method), 97
 on_train_epoch_end() (*dicее.abstracts.AbstractTrainer* method), 93
 on_train_epoch_end() (*dicее.callbacks.ASWA* method), 104
 on_train_epoch_end() (*dicее.callbacks.Eval* method), 105
 on_train_epoch_end() (*dicее.callbacks.KGESaveCallback* method), 103
 on_train_epoch_end() (*dicее.callbacks.PrintCallback* method), 102
 on_train_epoch_end() (*dicее.models.base_model.BaseKGELighting* method), 3
 on_train_epoch_end() (*dicее.models.BaseKGELighting* method), 40
 OnevsAllDataset (class in *dicее*), 167
 OnevsAllDataset (class in *dicее.dataset_classes*), 111
 optim (*dicее.config.Namespace* attribute), 107

P

p (*dicее.config.Namespace* attribute), 109
 parameters() (*dicее.abstracts.BaseInteractiveKGE* method), 96
 path_single_kg (*dicее.config.Namespace* attribute), 107
 path_to_store_single_run (*dicее.config.Namespace* attribute), 107
 Perturb (class in *dicее.callbacks*), 106
 poly_NN() (*dicее.LFMult* method), 152
 poly_NN() (*dicее.models.function_space.LFMult* method), 22
 poly_NN() (*dicее.models.LFMult* method), 77
 polynomial() (*dicее.LFMult* method), 152
 polynomial() (*dicее.models.function_space.LFMult* method), 22
 polynomial() (*dicее.models.LFMult* method), 78
 pop() (*dicее.LFMult* method), 152
 pop() (*dicее.models.function_space.LFMult* method), 22
 pop() (*dicее.models.LFMult* method), 78
 predict() (*dicее.KGE* method), 161
 predict() (*dicее.knowledge_graph_embeddings.KGE* method), 125
 predict_data_loader() (*dicее.models.base_model.BaseKGELighting* method), 5
 predict_data_loader() (*dicее.models.BaseKGELighting* method), 41
 predict_missing_head_entity() (*dicее.KGE* method), 160
 predict_missing_head_entity() (*dicее.knowledge_graph_embeddings.KGE* method), 124
 predict_missing_relations() (*dicее.KGE* method), 161
 predict_missing_relations() (*dicее.knowledge_graph_embeddings.KGE* method), 124
 predict_missing_tail_entity() (*dicее.KGE* method), 161
 predict_missing_tail_entity() (*dicее.knowledge_graph_embeddings.KGE* method), 125
 predict_topk() (*dicее.KGE* method), 162
 predict_topk() (*dicее.knowledge_graph_embeddings.KGE* method), 125
 prepare_data() (*dicее.CVDataModule* method), 173

prepare_data() (*dicee.dataset_classes.CVDataModule method*), 117
 preprocess_with_byte_pair_encoding() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 83
 preprocess_with_byte_pair_encoding() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 79
 preprocess_with_byte_pair_encoding_with_padding() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 83
 preprocess_with_byte_pair_encoding_with_padding() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 79
 preprocess_with_pandas() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 83
 preprocess_with_pandas() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 79
 preprocess_with_polars() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 84
 preprocess_with_polars() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 79
 preprocesses_input_args() (*in module dicee.static_preprocess_funcs*), 133
 PreprocessKG (*class in dicee.read_preprocess_save_load_kg*), 83
 PreprocessKG (*class in dicee.read_preprocess_save_load_kg.preprocess*), 78
 print_peak_memory() (*in module dicee.trainer.torch_trainer_ddp*), 89
 PrintCallback (*class in dicee.callbacks*), 101
 PseudoLabellingCallback (*class in dicee.callbacks*), 103
 Pyke (*class in dicee*), 137
 Pyke (*class in dicee.models*), 49
 Pyke (*class in dicee.models.real*), 30
 pykeen_model_kwargs (*dicee.config.Namespace attribute*), 109
 PykeenKGE (*class in dicee*), 152
 PykeenKGE (*class in dicee.models*), 73
 PykeenKGE (*class in dicee.models.pykeen_models*), 25

Q

q (*dicee.config.Namespace attribute*), 109
 QMult (*class in dicee*), 149
 QMult (*class in dicee.models*), 56
 QMult (*class in dicee.models.quaternion*), 26
 quaternion_mul() (*in module dicee.models*), 53
 quaternion_mul() (*in module dicee.models.static_funcs*), 31
 quaternion_mul_with_unit_norm() (*in module dicee.models*), 56
 quaternion_mul_with_unit_norm() (*in module dicee.models.quaternion*), 26
 quaternion_multiplication_followed_by_inner_product() (*dicee.models.QMult method*), 56
 quaternion_multiplication_followed_by_inner_product() (*dicee.models.quaternion.QMult method*), 27
 quaternion_multiplication_followed_by_inner_product() (*dicee.QMult method*), 149
 quaternion_normalizer() (*dicee.models.QMult static method*), 56
 quaternion_normalizer() (*dicee.models.quaternion.QMult static method*), 27
 quaternion_normalizer() (*dicee.QMult static method*), 149
 QueryGenerator (*class in dicee*), 174
 QueryGenerator (*class in dicee.query_generator*), 128

R

random_prediction() (*in module dicee*), 158
 random_prediction() (*in module dicee.static_funcs*), 131
 random_seed (*dicee.config.Namespace attribute*), 108
 read_from_disk() (*in module dicee.read_preprocess_save_load_kg.util*), 82
 read_from_triple_store() (*in module dicee.read_preprocess_save_load_kg.util*), 82
 read_only_few (*dicee.config.Namespace attribute*), 108
 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*), 131
 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*), 82
 read_with_polars() (*in module dicee.read_preprocess_save_load_kg.util*), 81
 ReadFromDisk (*class in dicee.read_preprocess_save_load_kg*), 84
 ReadFromDisk (*class in dicee.read_preprocess_save_load_kg.read_from_disk*), 80
 relations_str (*dicee.knowledge_graph.KG property*), 123
 reload_dataset() (*in module dicee*), 166
 reload_dataset() (*in module dicee.dataset_classes*), 110
 remove_triples_from_train_with_condition() (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 84
 remove_triples_from_train_with_condition() (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 79
 residual_convolution() (*dicee.AConEx method*), 145
 residual_convolution() (*dicee.AConvO method*), 146
 residual_convolution() (*dicee.AConvQ method*), 146

`residual_convolution()` (*dicee.ConvEx method*), 148
`residual_convolution()` (*dicee.ConvO method*), 148
`residual_convolution()` (*dicee.ConvQ method*), 147
`residual_convolution()` (*dicee.models.AConvEx method*), 51
`residual_convolution()` (*dicee.models.AConvO method*), 63
`residual_convolution()` (*dicee.models.AConvQ method*), 58
`residual_convolution()` (*dicee.models.complex.AConvEx method*), 18
`residual_convolution()` (*dicee.models.complex.ConvEx method*), 17
`residual_convolution()` (*dicee.models.ConvEx method*), 51
`residual_convolution()` (*dicee.models.ConvO method*), 62
`residual_convolution()` (*dicee.models.ConvQ method*), 57
`residual_convolution()` (*dicee.models.octonion.AConvO method*), 25
`residual_convolution()` (*dicee.models.octonion.ConvO method*), 24
`residual_convolution()` (*dicee.models.quaternion.AConvQ method*), 29
`residual_convolution()` (*dicee.models.quaternion.ConvQ method*), 28
`retrieve_embeddings()` (*in module dicee.scripts.serve*), 86
`return_multi_hop_query_results()` (*dicee.KGE method*), 163
`return_multi_hop_query_results()` (*dicee.knowledge_graph_embeddings.KGE method*), 126
`root()` (*in module dicee.scripts.serve*), 86

S

`sample_entity()` (*dicee.abstracts.BaseInteractiveKGE method*), 95
`sample_relation()` (*dicee.abstracts.BaseInteractiveKGE method*), 95
`sample_triples_ratio` (*dicee.config.Namespace attribute*), 108
`sanity_checking_with_arguments()` (*in module dicee.sanity_checkers*), 129
`save()` (*dicee.abstracts.BaseInteractiveKGE method*), 95
`save()` (*dicee.read_preprocess_save_load_kg.LoadSaveToDisk method*), 84
`save()` (*dicee.read_preprocess_save_load_kg.save_load_disk.LoadSaveToDisk method*), 80
`save_checkpoint()` (*dicee.abstracts.AbstractTrainer static method*), 93
`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*), 131
`save_embeddings_as_csv` (*dicee.config.Namespace attribute*), 107
`save_experiment()` (*dicee.analyse_experiments.Experiment method*), 99
`save_model_at_every_epoch` (*dicee.config.Namespace attribute*), 108
`save_numpy_ndarray()` (*in module dicee*), 158
`save_numpy_ndarray()` (*in module dicee.read_preprocess_save_load_kg.util*), 82
`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*), 82
`save_pickle()` (*in module dicee.static_funcs*), 130
`save_queries()` (*dicee.query_generator.QueryGenerator method*), 128
`save_queries()` (*dicee.QueryGenerator method*), 174
`save_queries_and_answers()` (*dicee.query_generator.QueryGenerator static method*), 128
`save_queries_and_answers()` (*dicee.QueryGenerator static method*), 174
`save_trained_model()` (*dicee.Execute method*), 165
`save_trained_model()` (*dicee.executer.Execute method*), 121
`scalar_batch_NN()` (*dicee.LFMMult method*), 152
`scalar_batch_NN()` (*dicee.models.function_space.LFMMult method*), 22
`scalar_batch_NN()` (*dicee.models.LFMMult method*), 77
`score()` (*dicee.CMult method*), 137
`score()` (*dicee.ComplEx static method*), 145
`score()` (*dicee.DistMult method*), 138
`score()` (*dicee.Keci method*), 141
`score()` (*dicee.models.clifford.CMult method*), 10
`score()` (*dicee.models.clifford.Keci method*), 14
`score()` (*dicee.models.CMult method*), 67
`score()` (*dicee.models.ComplEx static method*), 52
`score()` (*dicee.models.complex.ComplEx static method*), 19
`score()` (*dicee.models.DistMult method*), 48
`score()` (*dicee.models.Keci method*), 66
`score()` (*dicee.models.octonion.OMult method*), 24
`score()` (*dicee.models.OMult method*), 62
`score()` (*dicee.models.QMult method*), 57
`score()` (*dicee.models.quaternion.QMult method*), 28
`score()` (*dicee.models.real.DistMult method*), 30

`score()` (*dicee.models.real.TransE method*), 30
`score()` (*dicee.models.TransE method*), 48
`score()` (*dicee.OMult method*), 151
`score()` (*dicee.QMult method*), 150
`score()` (*dicee.TransE method*), 141
`scoring_technique` (*dicee.config.Namespace attribute*), 108
`search()` (*dicee.scripts.serve.NeuralSearcher method*), 86
`search_embeddings()` (*in module dicee.scripts.serve*), 86
`select_model()` (*in module dicee*), 157
`select_model()` (*in module dicee.static_funcs*), 130
`sequential_vocabulary_construction()` (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 84
`sequential_vocabulary_construction()` (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 79
`set_global_seed()` (*dicee.query_generator.QueryGenerator method*), 128
`set_global_seed()` (*dicee.QueryGenerator method*), 174
`set_model_eval_mode()` (*dicee.abstracts.BaseInteractiveKGE method*), 95
`set_model_train_mode()` (*dicee.abstracts.BaseInteractiveKGE method*), 94
`setup()` (*dicee.CVDDataModule method*), 172
`setup()` (*dicee.dataset_classes.CVDDataModule method*), 116
`Shallom` (*class in dicee*), 151
`Shallom` (*class in dicee.models*), 48
`Shallom` (*class in dicee.models.real*), 30
`single_hop_query_answering()` (*dicee.KGE method*), 163
`single_hop_query_answering()` (*dicee.knowledge_graph_embeddings.KGE method*), 126
`sparql_endpoint` (*dicee.config.Namespace attribute*), 107
`start()` (*dicee.DICE_Trainer method*), 159
`start()` (*dicee.Execute method*), 165
`start()` (*dicee.executor.Execute method*), 122
`start()` (*dicee.read_preprocess_save_load_kg.PreprocessKG method*), 83
`start()` (*dicee.read_preprocess_save_load_kg.preprocess.PreprocessKG method*), 78
`start()` (*dicee.read_preprocess_save_load_kg.read_from_disk.ReadFromDisk method*), 80
`start()` (*dicee.read_preprocess_save_load_kg.ReadFromDisk method*), 84
`start()` (*dicee.trainer.DICE_Trainer method*), 91
`start()` (*dicee.trainer.dice_trainer.DICE_Trainer method*), 88
`storage_path` (*dicee.config.Namespace attribute*), 107
`store()` (*in module dicee*), 158
`store()` (*in module dicee.static_funcs*), 131
`store_ensemble()` (*dicee.abstracts.AbstractPPECallback method*), 99
`swa` (*dicee.config.Namespace attribute*), 109

T

`t_conorm()` (*dicee.KGE method*), 163
`t_conorm()` (*dicee.knowledge_graph_embeddings.KGE method*), 126
`t_norm()` (*dicee.KGE method*), 163
`t_norm()` (*dicee.knowledge_graph_embeddings.KGE method*), 126
`tensor_t_norm()` (*dicee.KGE method*), 163
`tensor_t_norm()` (*dicee.knowledge_graph_embeddings.KGE method*), 126
`test_dataloader()` (*dicee.models.base_model.BaseKGELightning method*), 4
`test_dataloader()` (*dicee.models.BaseKGELightning method*), 40
`test_epoch_end()` (*dicee.models.base_model.BaseKGELightning method*), 4
`test_epoch_end()` (*dicee.models.BaseKGELightning method*), 40
`timeit()` (*in module dicee*), 157, 166
`timeit()` (*in module dicee.read_preprocess_save_load_kg.util*), 81
`timeit()` (*in module dicee.static_funcs*), 130
`timeit()` (*in module dicee.static_preprocess_funcs*), 133
`to_df()` (*dicee.analyse_experiments.Experiment method*), 99
`TorchDDPTrainer` (*class in dicee.trainer.torch_trainer_ddp*), 89
`TorchTrainer` (*class in dicee.trainer.torch_trainer*), 88
`train()` (*dicee.KGE method*), 164
`train()` (*dicee.knowledge_graph_embeddings.KGE method*), 127
`train()` (*dicee.trainer.torch_trainer_ddp.DDPTrainer method*), 90
`train()` (*dicee.trainer.torch_trainer_ddp.NodeTrainer method*), 90
`train_dataloader()` (*dicee.CVDDataModule method*), 171
`train_dataloader()` (*dicee.dataset_classes.CVDDataModule method*), 115
`train_dataloader()` (*dicee.models.base_model.BaseKGELightning method*), 5
`train_dataloader()` (*dicee.models.BaseKGELightning method*), 41
`train_k_vs_all()` (*dicee.KGE method*), 164
`train_k_vs_all()` (*dicee.knowledge_graph_embeddings.KGE method*), 127

train_triples() (*dicee.KGE method*), 164
 train_triples() (*dicee.knowledge_graph_embeddings.KGE method*), 127
 trainer (*dicee.config.Namespace attribute*), 108
 training_step() (*dicee.BytE method*), 154
 training_step() (*dicee.models.base_model.BaseKGELightning method*), 2
 training_step() (*dicee.models.BaseKGELightning method*), 38
 training_step() (*dicee.models.transformers.BytE method*), 32
 TransE (*class in dicee*), 141
 TransE (*class in dicee.models*), 48
 TransE (*class in dicee.models.real*), 30
 transfer_batch_to_device() (*dicee.CVDataModule method*), 172
 transfer_batch_to_device() (*dicee.dataset_classes.CVDataModule method*), 116
 trapezoid() (*dicee.models.FMult2 method*), 76
 trapezoid() (*dicee.models.function_space.FMult2 method*), 21
 tri_score() (*dicee.LFMult method*), 152
 tri_score() (*dicee.models.function_space.LFMult method*), 22
 tri_score() (*dicee.models.function_space.LFMult1 method*), 21
 tri_score() (*dicee.models.LFMult method*), 77
 tri_score() (*dicee.models.LFMult1 method*), 77
 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*), 114
 tuple2list() (*dicee.query_generator.QueryGenerator method*), 128
 tuple2list() (*dicee.QueryGenerator method*), 174

U

unmap() (*dicee.query_generator.QueryGenerator method*), 128
 unmap() (*dicee.QueryGenerator method*), 174
 unmap_query() (*dicee.query_generator.QueryGenerator method*), 128
 unmap_query() (*dicee.QueryGenerator method*), 174

V

val_dataloader() (*dicee.models.base_model.BaseKGELightning method*), 4
 val_dataloader() (*dicee.models.BaseKGELightning method*), 40
 validate_knowledge_graph() (*in module dicee.sanity_checkers*), 129
 vocab_preparation() (*dicee.evaluator.Evaluator method*), 119
 vocab_size (*dicee.models.transformers.GPTConfig attribute*), 35
 vocab_to_parquet() (*in module dicee*), 158
 vocab_to_parquet() (*in module dicee.static_funcs*), 132
 vtp_score() (*dicee.LFMult method*), 152
 vtp_score() (*dicee.models.function_space.LFMult method*), 22
 vtp_score() (*dicee.models.function_space.LFMult1 method*), 21
 vtp_score() (*dicee.models.LFMult method*), 78
 vtp_score() (*dicee.models.LFMult1 method*), 77

W

weight_decay (*dicee.config.Namespace attribute*), 108
 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