Texar Documentation

Release v0.1.0

Texar

Dec 08, 2018

Contents

Texar is a modularized, versatile, and extensible toolkit for text generation tasks and beyond.

CHAPTER 1

Overview

Texar is an open-source toolkit based on Tensorflow, aiming to support a broad set of machine learning especially **text generation tasks**, such as machine translation, dialog, summarization, content manipulation, language modeling, and so on. Texar is designed for both researchers and practitioners for fast prototyping and experimentation.

With the design goals of **modularity, versatility, and extensibility** in mind, Texar extracts the common patterns underlying the diverse tasks and methodologies, creates a library of highly reusable modules and functionalities, and facilitates **arbitrary model architectures and algorithmic paradigms**, e.g.,

- encoder(s) to decoder(s), sequential- and self-attentions, memory, hierarchical models, classifiers...
- maximum likelihood learning, reinforcement learning, adversarial learning, probabilistic modeling, ...

With Texar, cutting-edge complex models can be easily constructed, freely enriched with best modeling/training practices, readily fitted into standard training/evaluation pipelines, and fastly experimented and evolved by, e.g., plugging-in and swapping-out different modules.

1.1 Key Features

- **Versatility**. Texar contains a wide range of modules and functionalities for composing arbitrary model architectures and implementing various learning algorithms, as well as for data processing, evaluation, prediction, etc.
- **Modularity**. Texar decomposes diverse complex machine learning models/algorithms into a set of highly-reusable modules. In particular, model **architecture, losses, and learning processes** are fully decomposed.Users can construct their own models at a high conceptual level just like assembling building blocks. It is convenient to plug in or swap out modules, and configure rich options of each module. For example, switching between maximum likelihood learning and reinforcement learning involves only changing several lines of code.
- **Extensibility**. It is straightforward to integrate any user-customized, external modules. Also, Texar is fully compatible with the native Tensorflow interfaces and can take advantage of the rich Tensorflow features, and resources from the vibrant open-source community.

- Interfaces with different functionality levels. Users can customize a model through 1) simple **Python/YAML configuration files** of provided model templates/examples; 2) programming with **Python Library APIs** for maximal customizability.
- Easy-to-use APIs: 1) Convenient automatic variable re-use—no worry about the complicated TF variable scopes; 2) Pytorch-like callable modules; 3) Rich configuration options for each module, all with default values; ...
- Well-structured high-quality code of uniform design patterns and consistent styles.
- Clean, detailed documentation and rich examples.

1.2 Library API Example

Builds a (self-)attentional sequence encoder-decoder model, with different learning algorithms:

```
import texar as tx

# Data
data = tx.data.PairedTextData(hparams=hparams_data) # Hyperparameter configs in␣
↪`hparams`
iterator = tx.data.DataIterator(data)
batch = iterator.get_next() # A data mini-batch

# Model architecture
embedder = tx.modules.WordEmbedder(data.target_vocab.size, hparams=hparams_emb)
encoder = tx.modules.TransformerEncoder(hparams=hparams_encoder)
outputs_enc = encoder(inputs=embedder(batch['source_text_ids']),
                      sequence_length=batch['source_length'])

decoder = tx.modules.AttentionRNNDecoder(memory=output_enc,
                                         memory_sequence_length=batch['source_length
↪'],
                                         hparams=hparams_decoder)
outputs, _, _ = decoder(inputs=embedder(batch['target_text_ids']),
                        sequence_length=batch['target_length']-1)

# Loss for maximum likelihood learning
loss = tx.losses.sequence_sparse_softmax_cross_entropy(
    labels=batch['target_text_ids'][:, 1:],
    logits=outputs.logits,
    sequence_length=batch['target_length']-1) # Automatic masks

# Beam search decoding
outputs_bs, _, _ = tx.modules.beam_search_decode(
    decoder,
    embedding=embedder,
    start_tokens=[data.target_vocab.bos_token_id]*num_samples,
    end_token=data.target_vocab.eos_token_id)
```

```
# Policy gradient agent for RL learning
agent = tx.agents.SeqPGAgent(samples=outputs.sample_id,
                             logits=outputs.logits,
                             sequence_length=batch['target_length']-1,
                             hparams=config_model.agent)
```

Many more examples are available here

1.3 Installtion

```
git clone https://github.com/asyml/texar.git
cd texar
pip install -e .
```

1.4 Getting Started

- Examples
- Documentations
- GitHub

1.5 Reference

If you use Texar, please cite the report with the following BibTex entry:

```
Texar: A Modularized, Versatile, and Extensible Toolkit for Text Generation
Zhiting Hu, Haoran Shi, Zichao Yang, Bowen Tan, Tiancheng Zhao, Junxian He, Wentao␣
↪Wang, Xingjiang Yu, Lianhui Qin, Di Wang, Xuezhe Ma, Hector Liu, Xiaodan Liang,␣
↪Wanrong Zhu, Devendra Singh Sachan, Eric P. Xing
2018

@article{hu2018texar,
  title={Texar: A Modularized, Versatile, and Extensible Toolkit for Text Generation},
  author={Hu, Zhiting and Shi, Haoran and Yang, Zichao and Tan, Bowen and Zhao,␣
↪Tiancheng and He, Junxian and Wang, Wentao and Yu, Xingjiang and Qin, Lianhui and␣
↪Wang, Di and Ma, Xuezhe and Liu, Hector and Liang, Xiaodan and Zhu, Wanrong and␣
↪Sachan, Devendra Singh and Xing, Eric},
  year={2018}
}
```

1.6 License

Apache License 2.0

CHAPTER 2

Examples

Rich examples are included to demonstrate the use of Texar. The implementations of cutting-edge models/algorithms also provide references for reproducibility and comparisons.

More examples are continuously added...

2.1 Examples by Models/Algorithms

2.1.1 RNN / Seq2seq

- language_model_ptb: Basic RNN language model
- seq2seq_attn: Attentional seq2seq
- seq2seq_configs: Seq2seq implemented with Texar model template.
- seq2seq_rl: Attentional seq2seq trained with policy gradient.
- hierarchical_dialog: Hierarchical recurrent encoder-decoder model for conversation response generation.
- torchtext: Use of torchtext data loader

2.1.2 Transformer (Self-attention)

- transformer: Transformer for machine translation
- vae_text: VAE with a transformer decoder for improved language modeling

2.1.3 Variational Autoencoder (VAE)

- vae_text: VAE language model

2.1.4 GANs / Discriminiator-supervision

- seqGAN: GANs for text generation
- text_style_transfer: Discriminator supervision for controlled text generation

2.1.5 Reinforcement Learning

- seq2seq_rl: Attentional seq2seq trained with policy gradient.
- seqGAN: Policy gradient for sequence generation
- rl_gym: Various RL algoritms for games on OpenAI Gym

2.1.6 Memory Network

- memory_network_lm: End-to-end memory network for language modeling

2.1.7 Classifier / Predictions

- sentence_classifier: Basic CNN-based sentence classifier
- sequence_tagging: BiLSTM-CNN model for Named Entity Recognition (NER)

2.2 Examples by Tasks

2.2.1 Language Modeling

- language_model_ptb: Basic RNN language model
- vae_text: VAE language model
- seqGAN: GAN + policy gradient
- memory_network_lm: End-to-end memory network for language modeling

2.2.2 Machine Translation

- seq2seq_attn: Attentional seq2seq
- seq2seq_configs: Seq2seq implemented with Texar model template.
- seq2seq_rl: Attentional seq2seq trained with policy gradient.
- transformer: Transformer for machine translation

2.2.3 Dialog

- hierarchical_dialog: Hierarchical recurrent encoder-decoder model for conversation response generation.

2.2.4 Text Style Transfer

- text_style_transfer: Discriminator supervision for controlled text generation

2.2.5 Classification

- sentence_classifier: Basic CNN-based sentence classifier

2.2.6 Sequence Tagging

- sequence_tagging: BiLSTM-CNN model for Named Entity Recognition (NER)

2.2.7 Games

- rl_gym: Various RL algoritms for games on OpenAI Gym

CHAPTER 3

API

3.1 HParams

class texar.**HParams**(*hparams*, *default_hparams*, *allow_new_hparam=False*)

A class that maintains hyperparameters for configing Texar modules. The class has several useful features:

- **Auto-completion of missing values.** Users can specify only a subset of hyperparameters they care about. Other hyperparameters will automatically take the default values. The auto-completion performs **recursively** so that hyperparameters taking *dict* values will also be auto-completed **All Texar modules** provide a `default_hparams()` containing allowed hyperparameters and their default values. For example

```
## Recursive auto-completion
default_hparams = {"a": 1, "b": {"c": 2, "d": 3}}
hparams = {"b": {"c": 22}}
hparams_ = HParams(hparams, default_hparams)
hparams_.todict() == {"a": 1, "b": {"c": 22, "d": 3}}
    # "a" and "d" are auto-completed

## All Texar modules have built-in `default_hparams`
hparams = {"dropout_rate": 0.1}
emb = tx.modules.WordEmbedder(hparams=hparams, ...)
emb.hparams.todict() == {
    "dropout_rate": 0.1,  # provided value
    "dim": 100            # default value
    ...
}
```

- **Automatic typecheck.** For most hyperparameters, provided value must have the same or compatible dtype with the default value. HParams does necessary typecheck, and raises Error if improper dtype is provided. Also, hyperparameters not listed in *default_hparams* are not allowed, except for "kwargs" as detailed below.
- **Flexible dtype for specified hyperparameters.** Some hyperparameters may allow different dtypes of values.

  - Hyperparameters named "type" are not typechecked. For example, in *get_rnn_cell()*, hyperparameter *"type"* can take value of an RNNCell class, its string name of module path, or an RNNCell class instance. (String name or module path is allowd so that users can specify the value in YAML config files.)
  - For other hyperparameters, list them in the "@no_typecheck" field in *default_hparams* to skip typecheck. For example, in *get_rnn_cell()*, hyperparameter "*_keep_prob" can be set to either a *float* or a *tf.placeholder*.

- **Special flexibility of keyword argument hyparameters.** Hyperparameters named "kwargs" are used as keyword arguments for a class constructor or a function call. Such hyperparameters take a *dict*, and users can add arbitrary valid keyword arguments to the dict. For example:

```
default_rnn_cell_hparams = {
    "type": "LSTMCell",
    "kwargs": {"num_units": 256}
    # Other hyperparameters
    ...
}
my_hparams = {
    "kwargs" {
        "num_units": 123,
        "forget_bias": 0.0          # Other valid keyword arguments
        "activation": "tf.nn.relu" # for LSTMCell constructor
    }
}
_ = HParams(my_hparams, default_rnn_cell_hparams)
```

- **Rich interfaces.** An HParams instance provides rich interfaces for accessing, updating, or adding hyperparameters.

```
hparams = HParams(my_hparams, default_hparams)
# Access
hparams.type == hparams["type"]
# Update
hparams.type = "GRUCell"
hparams.kwargs = { "num_units": 100 }
hparams.kwargs.num_units == 100
# Add new
hparams.add_hparam("index", 1)
hparams.index == 1

# Convert to `dict` (recursively)
type(hparams.todic()) == dict

# I/O
pickle.dump(hparams, "hparams.dump")
with open("hparams.dump", 'rb') as f:
    hparams_loaded = pickle.load(f)
```

Parameters

- **hparams** – A *dict* or an *HParams* instance containing hyperparameters. If *None*, all hyperparameters are set to default values.
- **default_hparams** (*dict*) – Hyperparameters with default values. If *None*, Hyperparameters are fully defined by `hparams`.

- **allow_new_hparam** (*bool*) – If *False* (default), `hparams` cannot contain hyperparameters that are not included in `default_hparams`, except for the case of `"kwargs"` as above.

items()
Returns the list of hyperparam *(name, value)* pairs

keys()
Returns the list of hyperparam names

get(*name*, *default=None*)
Returns the hyperparameter value for the given name. If name is not available then returns `default`.

Parameters

- **name** (*str*) – the name of hyperparameter.
- **default** – the value to be returned in case name does not exist.

add_hparam(*name*, *value*)
Adds a new hyperparameter.

todict()
Returns a copy of hyperparameters as a dictionary.

3.2 Data

3.2.1 Vocabulary

SpecialTokens

class texar.data.**SpecialTokens**
Special tokens, including `PAD`, `BOS`, `EOS`, `UNK`. These tokens will by default have token ids 0, 1, 2, 3, respectively.

Vocab

class texar.data.**Vocab**(*filename*, *pad_token='<PAD>'*, *bos_token='<BOS>'*, *eos_token='<EOS>'*, *unk_token='<UNK>'*)
Vocabulary class that loads vocabulary from file, and maintains mapping tables between token strings and indexes.

Each line of the vocab file should contains one vocabulary token, e.g.,:

```
vocab_token_1
vocab token 2
vocab       token | 3 .
...
```

Parameters

- **filename** (*str*) – Path to the vocabulary file where each line contains one token.
- **bos_token** (*str*) – A special token that will be added to the beginning of sequences.
- **eos_token** (*str*) – A special token that will be added to the end of sequences.

- **unk_token** (*str*) – A special token that will replace all unknown tokens (tokens not included in the vocabulary).
- **pad_token** (*str*) – A special token that is used to do padding.

load(*filename*)
Loads the vocabulary from the file.

Parameters **filename** (*str*) – Path to the vocabulary file.

Returns A tuple of TF and python mapping tables between word string and index, (*id_to_token_map*, *token_to_id_map*, *id_to_token_map_py*, *token_to_id_map_py*), where *id_to_token_map* and *token_to_id_map* are TF HashTable instances, and *id_to_token_map_py* and *token_to_id_map_py* are python *defaultdict* instances.

map_ids_to_tokens(*ids*)
Maps ids into text tokens.

The returned tokens are a Tensor.

Parameters **ids** – An *int* tensor of token ids.

Returns A tensor of text tokens of the same shape.

map_tokens_to_ids(*tokens*)
Maps text tokens into ids.

The returned ids are a Tensor.

Parameters **tokens** – An tensor of text tokens.

Returns A tensor of token ids of the same shape.

map_ids_to_tokens_py(*ids*)
Maps ids into text tokens.

The input `ids` and returned tokens are both python arrays or list.

Parameters **ids** – An *int* numpy arry or (possibly nested) list of token ids.

Returns A numpy array of text tokens of the same shape as `ids`.

map_tokens_to_ids_py(*tokens*)
Maps text tokens into ids.

The input `tokens` and returned ids are both python arrays or list.

Parameters **tokens** – A numpy array or (possibly nested) list of text tokens.

Returns A numpy array of token ids of the same shape as `tokens`.

id_to_token_map
The :tf_main:‘HashTable <contrib/lookup/HashTable>‘instance that maps from token index to the string form.

token_to_id_map
The HashTable instance that maps from token string to the index.

id_to_token_map_py
The python *defaultdict* instance that maps from token index to the string form.

token_to_id_map_py
The python *defaultdict* instance that maps from token string to the index.

size
The vocabulary size.

bos_token
A string of the special token indicating the beginning of sequence.

bos_token_id
The *int* index of the special token indicating the beginning of sequence.

eos_token
A string of the special token indicating the end of sequence.

eos_token_id
The *int* index of the special token indicating the end of sequence.

unk_token
A string of the special token indicating unknown token.

unk_token_id
The *int* index of the special token indicating unknown token.

pad_token
A string of the special token indicating padding token. The default padding token is an empty string.

pad_token_id
The *int* index of the special token indicating padding token.

special_tokens
The list of special tokens [*pad_token*, *bos_token*, *eos_token*, *unk_token*].

3.2.2 Embedding

Embedding

class texar.data.**Embedding**(*vocab*, *hparams=None*)
Embedding class that loads token embedding vectors from file. Token embeddings not in the embedding file are initialized as specified in `hparams`.

Parameters

- **vocab** (*dict*) – A dictionary that maps token strings to integer index.
- **read_fn** – Callable that takes *(filename, vocab, word_vecs)* and returns the updated *word_vecs*. E.g., `load_word2vec()` and `load_glove()`.

static default_hparams()
Returns a dictionary of hyperparameters with default values:

```
{
    "file": "",
    "dim": 50,
    "read_fn": "load_word2vec",
    "init_fn": {
        "type": "numpy.random.uniform",
        "kwargs": {
            "low": -0.1,
            "high": 0.1,
        }
    },
}
```

Here:

"file" [str] Path to the embedding file. If not provided, all embeddings are initialized with the initialization function.

"dim": int Dimension size of each embedding vector

"read_fn" [str or callable] Function to read the embedding file. This can be the function, or its string name or full module path. E.g.,

```
"read_fn": texar.data.load_word2vec
"read_fn": "load_word2vec"
"read_fn": "texar.data.load_word2vec"
"read_fn": "my_module.my_read_fn"
```

If function string name is used, the function must be in one of the modules: `texar.data` or `texar.custom`.

The function must have the same signature as with *load_word2vec()*.

"init_fn" [dict] Hyperparameters of the initialization function used to initialize embedding of tokens missing in the embedding file.

The function must accept argument named *size* or *shape* to specify the output shape, and return a numpy array of the shape.

The *dict* has the following fields:

"type" [str or callable] The initialization function. Can be either the function, or its string name or full module path.

"kwargs" [dict] Keyword arguments for calling the function. The function is called with `init_fn(size=[.., ..], **kwargs)`.

word_vecs
2D numpy array of shape *[vocab_size, embedding_dim]*.

vector_size
The embedding dimention size.

load_word2vec

texar.data.**load_word2vec**(*filename*, *vocab*, *word_vecs*)
Loads embeddings in the word2vec binary format which has a header line containing the number of vectors and their dimensionality (two integers), followed with number-of-vectors lines each of which is formatted as '<word-string> <embedding-vector>'.

Parameters

- **filename** (*str*) – Path to the embedding file.
- **vocab** (*dict*) – A dictionary that maps token strings to integer index. Tokens not in `vocab` are not read.
- **word_vecs** – A 2D numpy array of shape *[vocab_size, embed_dim]* which is updated as reading from the file.

Returns The updated `word_vecs`.

load_glove

texar.data.**load_glove**(*filename*, *vocab*, *word_vecs*)

Loads embeddings in the glove text format in which each line is '<word-string> <embedding-vector>'. Dimensions of the embedding vector are separated with whitespace characters.

Parameters

- **filename** (*str*) – Path to the embedding file.
- **vocab** (*dict*) – A dictionary that maps token strings to integer index. Tokens not in `vocab` are not read.
- **word_vecs** – A 2D numpy array of shape *[vocab_size, embed_dim]* which is updated as reading from the file.

Returns The updated `word_vecs`.

3.2.3 Data

DataBase

class texar.data.**DataBase**(*hparams*)

Base class inheritted by all data classes.

static default_hparams()

Returns a dictionary of default hyperparameters.

```
{
    "num_epochs": 1,
    "batch_size": 64,
    "allow_smaller_final_batch": True,
    "shuffle": True,
    "shuffle_buffer_size": None,
    "shard_and_shuffle": False,
    "num_parallel_calls": 1,
    "prefetch_buffer_size": 0,
    "max_dataset_size": -1,
    "seed": None,
    "name": "data",
}
```

Here:

"num_epochs" [int] Number of times the dataset should be repeated. An OutOfRangeError signal will be raised after the whole repeated dataset has been iterated through.

E.g., For training data, set it to 1 (default) so that you will get the signal after each epoch of training. Set to -1 to repeat the dataset indefinitely.

"batch_size" [int] Batch size, i.e., the number of consecutive elements of the dataset to combine in a single batch.

"allow_smaller_final_batch" [bool] Whether to allow the final batch to be smaller if there are insufficient elements left. If *False*, the final batch is discarded if it is smaller than batch size. Note that, if *True*, *output_shapes* of the resulting dataset will have a a **static** batch_size dimension equal to "batch_size".

"shuffle" [bool] Whether to randomly shuffle the elements of the dataset.

"shuffle_buffer_size" [int] The buffer size for data shuffling. The larger, the better the resulting data is mixed.

If *None* (default), buffer size is set to the size of the whole dataset (i.e., make the shuffling the maximally effective).

"shard_and_shuffle" [bool] Whether to first shard the dataset and then shuffle each block respectively. Useful when the whole data is too large to be loaded efficiently into the memory.

If *True*, `shuffle_buffer_size` must be specified to determine the size of each shard.

"num_parallel_calls" [int] Number of elements from the datasets to process in parallel.

"prefetch_buffer_size" [int] The maximum number of elements that will be buffered when prefetching.

max_dataset_size [int] Maximum number of instances to include in the dataset. If set to *-1* or greater than the size of dataset, all instances will be included. This constraint is imposed after data shuffling and filtering.

seed [int, optional] The random seed for shuffle.

Note that if a seed is set, the shuffle order will be exact the same every time when going through the (repeated) dataset.

For example, consider a dataset with elements [1, 2, 3], with "num_epochs"=*2* and some fixed seed, the resulting sequence can be: 2 1 3, 1 3 2 | 2 1 3, 1 3 2, ... That is, the orders are different **within** every *num_epochs*, but are the same **across** the *num_epochs*.

name [str] Name of the data.

num_epochs
Number of epochs.

batch_size
The batch size.

hparams
A *HParams* instance of the data hyperparameters.

name
Name of the module.

MonoTextData

class texar.data.**MonoTextData**(*hparams*)
Text data processor that reads single set of text files. This can be used for, e.g., language models, auto-encoders, etc.

Parameters **hparams** – A *dict* or instance of *HParams* containing hyperparameters. See *default_hparams()* for the defaults.

By default, the processor reads raw data files, performs tokenization, batching and other pre-processing steps, and results in a TF Dataset whose element is a python *dict* including three fields:

- **"text":** A string Tensor of shape *[batch_size, max_time]* containing the **raw** text toknes. *max_time* is the length of the longest sequence in the batch. Short sequences in the batch are padded with **empty string**. BOS and EOS tokens are added as per *hparams*. Out-of-vocabulary tokens are **NOT** replaced with UNK.
- **"text_ids":** An *int64* Tensor of shape *[batch_size, max_time]* containing the token indexes.

- **"length":** An *int* Tensor of shape *[batch_size]* containing the length of each sequence in the batch (including BOS and EOS if added).

If `'variable_utterance'` is set to *True* in *hparams*, the resulting dataset has elements with four fields:

- **"text":** A string Tensor of shape *[batch_size, max_utterance, max_time]*, where *max_utterance* is either the maximum number of utterances in each elements of the batch, or `max_utterance_cnt` as specified in *hparams*.
- **"text_ids":** An *int64* Tensor of shape *[batch_size, max_utterance, max_time]* containing the token indexes.
- **"length":** An *int* Tensor of shape *[batch_size, max_utterance]* containing the length of each sequence in the batch.
- **"utterance_cnt":** An *int* Tensor of shape *[batch_size]* containing the number of utterances of each element in the batch.

The above field names can be accessed through *text_name*, *text_id_name*, *length_name*, and *utterance_cnt_name*, respectively.

Example

```
hparams={
    'dataset': { 'files': 'data.txt', 'vocab_file': 'vocab.txt' },
    'batch_size': 1
}
data = MonoTextData(hparams)
iterator = DataIterator(data)
batch = iterator.get_next()

iterator.switch_to_dataset(sess) # initializes the dataset
batch_ = sess.run(batch)
# batch_ == {
#    'text': [['<BOS>', 'example', 'sequence', '<EOS>']],
#    'text_ids': [[1, 5, 10, 2]],
#    'length': [4]
# }
```

static default_hparams()

Returns a dicitionary of default hyperparameters:

```
{
    # (1) Hyperparams specific to text dataset
    "dataset": {
        "files": [],
        "compression_type": None,
        "vocab_file": "",
        "embedding_init": {},
        "delimiter": " ",
        "max_seq_length": None,
        "length_filter_mode": "truncate",
        "pad_to_max_seq_length": False,
        "bos_token": "<BOS>"
        "eos_token": "<EOS>"
        "other_transformations": [],
        "variable_utterance": False,
        "utterance_delimiter": "|||",
```

(continues on next page)

(continued from previous page)

```
        "max_utterance_cnt": 5,
        "data_name": None,
    }
    # (2) General hyperparams
    "num_epochs": 1,
    "batch_size": 64,
    "allow_smaller_final_batch": True,
    "shuffle": True,
    "shuffle_buffer_size": None,
    "shard_and_shuffle": False,
    "num_parallel_calls": 1,
    "prefetch_buffer_size": 0,
    "max_dataset_size": -1,
    "seed": None,
    "name": "mono_text_data",
    # (3) Bucketing
    "bucket_boundaries": [],
    "bucket_batch_sizes": None,
    "bucket_length_fn": None,
}
```

Here:

1. For the hyperparameters in the `"dataset"` field:

    "files" [str or list] A (list of) text file path(s).

    Each line contains a single text sequence.

    "compression_type" [str, optional] One of "" (no compression), "ZLIB", or "GZIP".

    "vocab_file": str Path to vocabulary file. Each line of the file should contain one vocabulary token.

    Used to create an instance of *Vocab*.

    "embedding_init" [dict] The hyperparameters for pre-trained embedding loading and initialization.

    The structure and default values are defined in *texar.data.Embedding.default_hparams()*.

    "delimiter" [str] The delimiter to split each line of the text files into tokens.

    "max_seq_length" [int, optional] Maximum length of output sequences. Data samples exceeding the length will be truncated or discarded according to `"length_filter_mode"`. The length does not include any added `"bos_token"` or `"eos_token"`. If *None* (default), no filtering is performed.

    "length_filter_mode" [str] Either "truncate" or "discard". If "truncate" (default), tokens exceeding the `"max_seq_length"` will be truncated. If "discard", data samples longer than the `"max_seq_length"` will be discarded.

    "pad_to_max_seq_length" [bool] If *True*, pad all data instances to length `"max_seq_length"`. Raises error if `"max_seq_length"` is not provided.

    "bos_token" [str] The Begin-Of-Sequence token prepended to each sequence.

    Set to an empty string to avoid prepending.

    "eos_token" [str] The End-Of-Sequence token appended to each sequence.

Set to an empty string to avoid appending.

"other_transformations" [list] A list of transformation functions or function names/paths to further transform each single data instance.

(More documentations to be added.)

"variable_utterance" [bool] If *True*, each line of the text file is considered to contain multiple sequences (utterances) separated by `"utterance_delimiter"`.

For example, in dialog data, each line can contain a series of dialog history utterances. See the example in *examples/hierarchical_dialog* for a use case.

"utterance_delimiter" [str] The delimiter to split over utterance level. Should not be the same with `"delimiter"`. Used only when `"variable_utterance"``==True`.

"max_utterance_cnt" [int] Maximally allowed number of utterances in a data instance. Extra utterances are truncated out.

"data_name" [str] Name of the dataset.

2. For the **general** hyperparameters, see *texar.data.DataBase.default_hparams()* for details.

3. **Bucketing** is to group elements of the dataset together by length and then pad and batch. (See more at bucket_by_sequence_length). For bucketing hyperparameters:

"bucket_boundaries" [list] An int list containing the upper length boundaries of the buckets.

Set to an empty list (default) to disable bucketing.

"bucket_batch_sizes" [list] An int list containing batch size per bucket. Length should be *len(bucket_boundaries) + 1*.

If *None*, every bucket whill have the same batch size specified in *batch_size*.

"bucket_length_fn" [str or callable] Function maps dataset element to *tf.int32* scalar, determines the length of the element.

This can be a function, or the name or full module path to the function. If function name is given, the function must be in the `texar.custom` module.

If *None* (default), length is determined by the number of tokens (including BOS and EOS if added) of the element.

list_items()

Returns the list of item names that the data can produce.

Returns A list of strings.

dataset

The dataset, an instance of TF dataset.

dataset_size()

Returns the number of data instances in the data files.

Note that this is the total data count in the raw files, before any filtering and truncation.

vocab

The vocabulary, an instance of *Vocab*.

embedding_init_value

The *Tensor* containing the embedding value loaded from file. *None* if embedding is not specified.

text_name

The name of text tensor, "text" by default.

length_name
The name of length tensor, "length" by default.

text_id_name
The name of text index tensor, "text_ids" by default.

utterance_cnt_name
The name of utterance count tensor, "utterance_cnt" by default.

batch_size
The batch size.

hparams
A *HParams* instance of the data hyperparameters.

name
Name of the module.

num_epochs
Number of epochs.

PairedTextData

class texar.data.**PairedTextData**(*hparams*)
Text data processor that reads parallel source and target text. This can be used in, e.g., seq2seq models.

Parameters **hparams** (*dict*) – Hyperparameters. See *default_hparams()* for the defaults.

By default, the processor reads raw data files, performs tokenization, batching and other pre-processing steps, and results in a TF Dataset whose element is a python *dict* including six fields:

- **"source_text":** A string Tensor of shape *[batch_size, max_time]* containing the **raw** text toknes of source sequences. *max_time* is the length of the longest sequence in the batch. Short sequences in the batch are padded with **empty string**. By default only EOS token is appended to each sequence. Out-of-vocabulary tokens are **NOT** replaced with UNK.
- **"source_text_ids":** An *int64* Tensor of shape *[batch_size, max_time]* containing the token indexes of source sequences.
- **"source_length":** An *int* Tensor of shape *[batch_size]* containing the length of each source sequence in the batch (including BOS and/or EOS if added).
- **"target_text":** A string Tensor as "source_text" but for target sequences. By default both BOS and EOS are added.
- **"target_text_ids":** An *int64* Tensor as "source_text_ids" but for target sequences.
- **"target_length":** An *int* Tensor of shape *[batch_size]* as "source_length" but for target sequences.

If `'variable_utterance'` is set to *True* in `'source_dataset'` and/or `'target_dataset'` of *hparams*, the corresponding fields "source_*" and/or "target_*" are respectively changed to contain variable utterance text data, as in *MonoTextData*.

The above field names can be accessed through *source_text_name*, *source_text_id_name*, *source_length_name*, *source_utterance_cnt_name*, and those prefixed with *target_*, respectively.

Example

```
hparams={
    'source_dataset': {'files': 's', 'vocab_file': 'vs'},
    'target_dataset': {'files': ['t1', 't2'], 'vocab_file': 'vt'},
    'batch_size': 1
}
data = PairedTextData(hparams)
iterator = DataIterator(data)
batch = iterator.get_next()

iterator.switch_to_dataset(sess) # initializes the dataset
batch_ = sess.run(batch)
# batch_ == {
#    'source_text': [['source', 'sequence', '<EOS>']],
#    'source_text_ids': [[5, 10, 2]],
#    'source_length': [3]
#    'target_text': [['<BOS>', 'target', 'sequence', '1', '<EOS>']],
#    'target_text_ids': [[1, 6, 10, 20, 2]],
#    'target_length': [5]
# }
```

static default_hparams()

Returns a dicitionary of default hyperparameters.

```
{
    # (1) Hyperparams specific to text dataset
    "source_dataset": {
        "files": [],
        "compression_type": None,
        "vocab_file": "",
        "embedding_init": {},
        "delimiter": " ",
        "max_seq_length": None,
        "length_filter_mode": "truncate",
        "pad_to_max_seq_length": False,
        "bos_token": None,
        "eos_token": "<EOS>",
        "other_transformations": [],
        "variable_utterance": False,
        "utterance_delimiter": "|||",
        "max_utterance_cnt": 5,
        "data_name": "source",
    },
    "target_dataset": {
        # ...
        # Same fields are allowed as in "source_dataset" with the
        # same default values, except the
        # following new fields/values:
        "bos_token": "<BOS>"
        "vocab_share": False,
        "embedding_init_share": False,
        "processing_share": False,
        "data_name": "target"
    }
    # (2) General hyperparams
    "num_epochs": 1,
    "batch_size": 64,
    "allow_smaller_final_batch": True,
```

(continues on next page)

(continued from previous page)

```
    "shuffle": True,
    "shuffle_buffer_size": None,
    "shard_and_shuffle": False,
    "num_parallel_calls": 1,
    "prefetch_buffer_size": 0,
    "max_dataset_size": -1,
    "seed": None,
    "name": "paired_text_data",
    # (3) Bucketing
    "bucket_boundaries": [],
    "bucket_batch_sizes": None,
    "bucket_length_fn": None,
}
```

Here:

1. Hyperparameters in the `"source_dataset"` and attr:*"target_dataset"* fields have the same definition as those in *texar.data.MonoTextData.default_hparams()*, for source and target text, respectively.

For the new hyperparameters in "target_dataset":

"vocab_share" [bool] Whether to share the vocabulary of source. If *True*, the vocab file of target is ignored.

"embedding_init_share" [bool] Whether to share the embedding initial value of source. If *True*, `"embedding_init"` of target is ignored.

`"vocab_share"` must be true to share the embedding initial value.

"processing_share" [bool] Whether to share the processing configurations of source, including "delimiter", "bos_token", "eos_token", and "other_transformations".

2. For the **general** hyperparameters, see *texar.data.DataBase.default_hparams()* for details.

3. For **bucketing** hyperparameters, see *texar.data.MonoTextData.default_hparams()* for details, except that the default bucket_length_fn is the maximum sequence length of source and target sequences.

list_items()
Returns the list of item names that the data can produce.

Returns A list of strings.

dataset
The dataset.

dataset_size()
Returns the number of data instances in the dataset.

Note that this is the total data count in the raw files, before any filtering and truncation.

vocab
A pair instances of *Vocab* that are source and target vocabs, respectively.

source_vocab
The source vocab, an instance of *Vocab*.

target_vocab
The target vocab, an instance of *Vocab*.

source_embedding_init_value
The *Tensor* containing the embedding value of source data loaded from file. *None* if embedding is not specified.

target_embedding_init_value
The *Tensor* containing the embedding value of target data loaded from file. *None* if embedding is not specified.

embedding_init_value()
A pair of *Tensor* containing the embedding values of source and target data loaded from file.

source_text_name
The name of the source text tensor, "source_text" by default.

source_length_name
The name of the source length tensor, "source_length" by default.

source_text_id_name
The name of the source text index tensor, "source_text_ids" by default.

source_utterance_cnt_name
The name of the source text utterance count tensor, "source_utterance_cnt" by default.

target_text_name
The name of the target text tensor, "target_text" bt default.

target_length_name
The name of the target length tensor, "target_length" by default.

target_text_id_name
The name of the target text index tensor, "target_text_ids" by default.

target_utterance_cnt_name
The name of the target text utterance count tensor, "target_utterance_cnt" by default.

text_name
The name of text tensor, "text" by default.

length_name
The name of length tensor, "length" by default.

text_id_name
The name of text index tensor, "text_ids" by default.

utterance_cnt_name
The name of the text utterance count tensor, "utterance_cnt" by default.

batch_size
The batch size.

hparams
A *HParams* instance of the data hyperparameters.

name
Name of the module.

num_epochs
Number of epochs.

ScalarData

class texar.data.**ScalarData**(*hparams*)

Scalar data where each line of the files is a scalar (int or float), e.g., a data label.

Parameters **hparams** (*dict*) – Hyperparameters. See *default_hparams()* for the defaults.

The processor reads and processes raw data and results in a TF dataset whose element is a python *dict* including one field. The field name is specified in `hparams["dataset"]["data_name"]`. If not specified, the default name is *"data"*. The field name can be accessed through *data_name*.

This field is a Tensor of shape *[batch_size]* containing a batch of scalars, of either int or float type as specified in *hparams*.

Example

```
hparams={
    'dataset': { 'files': 'data.txt', 'data_name': 'label' },
    'batch_size': 2
}
data = ScalarData(hparams)
iterator = DataIterator(data)
batch = iterator.get_next()

iterator.switch_to_dataset(sess) # initializes the dataset
batch_ = sess.run(batch)
# batch_ == {
#     'label': [2, 9]
# }
```

static default_hparams()

Returns a dicitionary of default hyperparameters.

```
{
    # (1) Hyperparams specific to scalar dataset
    "dataset": {
        "files": [],
        "compression_type": None,
        "data_type": "int",
        "other_transformations": [],
        "data_name": None,
    }
    # (2) General hyperparams
    "num_epochs": 1,
    "batch_size": 64,
    "allow_smaller_final_batch": True,
    "shuffle": True,
    "shuffle_buffer_size": None,
    "shard_and_shuffle": False,
    "num_parallel_calls": 1,
    "prefetch_buffer_size": 0,
    "max_dataset_size": -1,
    "seed": None,
    "name": "scalar_data",
}
```

Here:

1. For the hyperparameters in the `"dataset"` field:

   "files" [str or list] A (list of) file path(s).

   Each line contains a single scalar number.

   "compression_type" [str, optional] One of "" (no compression), "ZLIB", or "GZIP".

   "data_type" [str] The scalar type. Currently supports "int" and "float".

   "other_transformations" [list] A list of transformation functions or function names/paths to further transform each single data instance.

   (More documentations to be added.)

   "data_name" [str] Name of the dataset.

2. For the **general** hyperparameters, see *texar.data.DataBase.default_hparams()* for details.

list_items()
Returns the list of item names that the data can produce.

Returns A list of strings.

dataset
The dataset.

dataset_size()
Returns the number of data instances in the dataset.

Note that this is the total data count in the raw files, before any filtering and truncation.

data_name
The name of the data tensor, "data" by default if not specified in *hparams*.

batch_size
The batch size.

hparams
A *HParams* instance of the data hyperparameters.

name
Name of the module.

num_epochs
Number of epochs.

MultiAlignedData

class texar.data.**MultiAlignedData**(*hparams*)
Data consisting of multiple aligned parts.

Parameters **hparams** (*dict*) – Hyperparameters. See *default_hparams()* for the defaults.

The processor can read any number of parallel fields as specified in the "datasets" list of *hparams*, and result in a TF Dataset whose element is a python *dict* containing data fields from each of the specified datasets. Fields from a text dataset have names prefixed by its "data_name". Fields from a scalar dataset are specified by its "data_name".

Example

```
hparams={
    'datasets': [
        {'files': 'a.txt', 'vocab_file': 'v.a', 'data_name': 'x'},
        {'files': 'b.txt', 'vocab_file': 'v.b', 'data_name': 'y'},
        {'files': 'c.txt', 'data_type': 'int', 'data_name': 'z'}
    ]
    'batch_size': 1
}
data = MultiAlignedData(hparams)
iterator = DataIterator(data)
batch = iterator.get_next()

iterator.switch_to_dataset(sess) # initializes the dataset
batch_ = sess.run(batch)
# batch_ == {
#     'x_text': [['<BOS>', 'x', 'sequence', '<EOS>']],
#     'x_text_ids': [['1', '5', '10', '2']],
#     'x_length': [4]
#     'y_text': [['<BOS>', 'y', 'sequence', '1', '<EOS>']],
#     'y_text_ids': [['1', '6', '10', '20', '2']],
#     'y_length': [5],
#     'z': [1000]
# }
```

static default_hparams()

Returns a dicitionary of default hyperparameters.

```
{
    # (1) Hyperparams specific to text dataset
    "datasets": []
    # (2) General hyperparams
    "num_epochs": 1,
    "batch_size": 64,
    "allow_smaller_final_batch": True,
    "shuffle": True,
    "shuffle_buffer_size": None,
    "shard_and_shuffle": False,
    "num_parallel_calls": 1,
    "prefetch_buffer_size": 0,
    "max_dataset_size": -1,
    "seed": None,
    "name": "multi_aligned_data",
}
```

Here:

1. "datasets" is a list of *dict* each of which specifies a text or scalar dataset. The `"data_name"` field of each dataset is used as the name prefix of the data fields from the respective dataset. The `"data_name"` field of each dataset should not be the same.

- For scalar dataset, the allowed hyperparameters and default values are the same as the "dataset" field of *texar.data.ScalarData.default_hparams()*. Note that `"data_type"` must be explicily specified (either "int" or "float").
- For text dataset, the allowed hyperparameters and default values are the same as the "dataset" filed of *texar.data.MonoTextData.default_hparams()*, with several extra hyperparameters:

"data_type" [str] The type of the dataset, one of {"text", "int", "float"}. If set to "int" or "float", the dataset is considered to be a scalar dataset. If not specified or set to "text", the dataset is considered to be a text dataset.

"vocab_share_with" [int, optional] Share the vocabulary of a preceding text dataset with the specified index in the list (starting from 0). The specified dataset must be a text dataset, and must have an index smaller than the current dataset.

If specified, the vocab file of current dataset is ignored. Default is *None* which disables the vocab sharing.

"embedding_init_share_with": int, optional Share the embedding initial value of a preceding text dataset with the specified index in the list (starting from 0). The specified dataset must be a text dataset, and must have an index smaller than the current dataset.

If specified, the `"embedding_init"` field of the current dataset is ignored. Default is *None* which disables the initial value sharing.

"processing_share_with" [int, optional] Share the processing configurations of a preceding text dataset with the specified index in the list (starting from 0). The specified dataset must be a text dataset, and must have an index smaller than the current dataset.

If specified, relevant field of the current dataset are ignored, including "delimiter", "bos_token", "eos_token", and "other_transformations". Default is *None* which disables the processing sharing.

2. For the **general** hyperparameters, see *texar.data.DataBase.default_hparams()* for details.

list_items()
Returns the list of item names that the data can produce.

Returns A list of strings.

dataset
The dataset.

dataset_size()
Returns the number of data instances in the dataset.

Note that this is the total data count in the raw files, before any filtering and truncation.

vocab(*name_or_id*)
Returns the *Vocab* of text dataset by its name or id. *None* if the dataset is not of text type.

Parameters **name_or_id** (*str or int*) – Data name or the index of text dataset.

embedding_init_value(*name_or_id*)
Returns the *Tensor* of embedding init value of the dataset by its name or id. *None* if the dataset is not of text type.

text_name(*name_or_id*)
The name of text tensor of text dataset by its name or id. If the dataaet is not of text type, returns *None*.

length_name(*name_or_id*)
The name of length tensor of text dataset by its name or id. If the dataset is not of text type, returns *None*.

text_id_name(*name_or_id*)
The name of length tensor of text dataset by its name or id. If the dataset is not of text type, returns *None*.

utterance_cnt_name(*name_or_id*)
The name of utterance count tensor of text dataset by its name or id. If the dataset is not variable utterance text data, returns *None*.

data_name
The name of the data tensor of scalar dataset by its name or id.. If the dataset is not a scalar data, returns *None*.

batch_size
The batch size.

hparams
A *HParams* instance of the data hyperparameters.

name
Name of the module.

num_epochs
Number of epochs.

TextDataBase

class texar.data.**TextDataBase**(*hparams*)
Base class inheritted by all text data classes.

static default_hparams()
Returns a dictionary of default hyperparameters.

See the specific subclasses for the details.

3.2.4 Data Iterators

DataIteratorBase

class texar.data.**DataIteratorBase**(*datasets*)
Base class for all data iterator classes to inherit. A data iterator is a wrapper of tf.data.Iterator, and can switch between and iterate through **multiple** datasets.

Parameters **datasets** – Datasets to iterates through. This can be:

- A single instance of tf.data.Dataset or instance of subclass of *DataBase*.
- A *dict* that maps dataset name to instance of tf.data.Dataset or subclass of *DataBase*.
- A *list* of instances of subclasses of *texar.data.DataBase*. The name of instances (*texar.data.DataBase.name*) must be unique.

num_datasets
Number of datasets.

dataset_names
A list of dataset names.

DataIterator

class texar.data.**DataIterator**(*datasets*)
Data iterator that switches and iterates through multiple datasets.

This is a wrapper of TF reinitializble iterator.

Parameters **datasets** – Datasets to iterates through. This can be:

- A single instance of tf.data.Dataset or instance of subclass of *DataBase*.

- A *dict* that maps dataset name to instance of tf.data.Dataset or subclass of *DataBase*.
- A *list* of instances of subclasses of *texar.data.DataBase*. The name of instances (*texar.data.DataBase.name*) must be unique.

Example

```
train_data = MonoTextData(hparams_train)
test_data = MonoTextData(hparams_test)
iterator = DataIterator({'train': train_data, 'test': test_data})
batch = iterator.get_next()

sess = tf.Session()

for _ in range(200): # Run 200 epochs of train/test
    # Starts iterating through training data from the beginning
    iterator.switch_to_dataset(sess, 'train')
    while True:
        try:
            train_batch_ = sess.run(batch)
        except tf.errors.OutOfRangeError:
            print("End of training epoch.")
    # Starts iterating through test data from the beginning
    iterator.switch_to_dataset(sess, 'test')
    while True:
        try:
            test_batch_ = sess.run(batch)
        except tf.errors.OutOfRangeError:
            print("End of test epoch.")
```

switch_to_dataset(*sess*, *dataset_name=None*)
Re-initializes the iterator of a given dataset and starts iterating over the dataset (from the beginning).

Parameters

- **sess** – The current tf session.
- **dataset_name** (*optional*) – Name of the dataset. If not provided, there must be only one Dataset.

get_next()
Returns the next element of the activated dataset.

TrainTestDataIterator

class texar.data.**TrainTestDataIterator**(*train=None*, *val=None*, *test=None*)
Data iterator that alternatives between train, val, and test datasets.

`train`, `val`, and `test` can be instance of either tf.data.Dataset or subclass of *DataBase*. At least one of them must be provided.

This is a wrapper of *DataIterator*.

Parameters

- **train** (*optional*) – Training data.
- **val** (*optional*) – Validation data.

- **test** (*optional*) – Test data.

Example

```
train_data = MonoTextData(hparams_train)
val_data = MonoTextData(hparams_val)
iterator = TrainTestDataIterator(train=train_data, val=val_data)
batch = iterator.get_next()

sess = tf.Session()

for _ in range(200): # Run 200 epochs of train/val
    # Starts iterating through training data from the beginning
    iterator.switch_to_train_data(sess)
    while True:
        try:
            train_batch_ = sess.run(batch)
        except tf.errors.OutOfRangeError:
            print("End of training epoch.")
    # Starts iterating through val data from the beginning
    iterator.switch_to_val_dataset(sess)
    while True:
        try:
            val_batch_ = sess.run(batch)
        except tf.errors.OutOfRangeError:
            print("End of val epoch.")
```

switch_to_train_data(*sess*)
Starts to iterate through training data (from the beginning).

Parameters **sess** – The current tf session.

switch_to_val_data(*sess*)
Starts to iterate through val data (from the beginning).

Parameters **sess** – The current tf session.

switch_to_test_data(*sess*)
Starts to iterate through test data (from the beginning).

Parameters **sess** – The current tf session.

FeedableDataIterator

class texar.data.**FeedableDataIterator**(*datasets*)
Data iterator that iterates through **multiple** datasets and switches between datasets.

The iterator can switch to a dataset and resume from where we left off last time we visited the dataset. This is a wrapper of TF feedable iterator.

Parameters **datasets** – Datasets to iterates through. This can be:

- A single instance of tf.data.Dataset or instance of subclass of *DataBase*.
- A *dict* that maps dataset name to instance of tf.data.Dataset or subclass of *DataBase*.
- A *list* of instances of subclasses of *texar.data.DataBase*. The name of instances (*texar.data.DataBase.name*) must be unique.

Example

```
train_data = MonoTextData(hparams={'num_epochs': 200, ...})
test_data = MonoTextData(hparams_test)
iterator = FeedableDataIterator({'train': train_data,
                                 'test': test_data})
batch = iterator.get_next()

sess = tf.Session()

def _eval_epoch(): # Iterate through test data for one epoch
    # Initialize and start from beginning of test data
    iterator.initialize_dataset(sess, 'test')
    while True:
        try:
            fetch_dict = { # Read from test data
                iterator.handle: Iterator.get_handle(sess, 'test')
            }
            test_batch_ = sess.run(batch, feed_dict=feed_dict)
        except tf.errors.OutOfRangeError:
            print("End of val epoch.")

# Initialize and start from beginning of training data
iterator.initialize_dataset(sess, 'train')
step = 0
while True:
    try:
        fetch_dict = { # Read from training data
            iterator.handle: Iterator.get_handle(sess, 'train')
        }
        train_batch_ = sess.run(batch, fetch_dict=fetch_dict)

        step +=1
        if step % 200 == 0: # Evaluate periodically
            _eval_epoch()
    except tf.errors.OutOfRangeError:
        print("End of training.")
```

get_handle(*sess*, *dataset_name=None*)

Returns a dataset handle used to feed the *handle* placeholder to fetch data from the dataset.

Parameters

- **sess** – The current tf session.
- **dataset_name** (*optional*) – Name of the dataset. If not provided, there must be only one Dataset.

Returns A string handle to be fed to the *handle* placeholder.

Example

```
next_element = iterator.get_next()
train_handle = iterator.get_handle(sess, 'train')
# Gets the next training element
ne_ = sess.run(next_element,
               feed_dict={iterator.handle: train_handle})
```

restart_dataset(*sess*, *dataset_name=None*)
Restarts datasets so that next iteration will fetch data from the beginning of the datasets.

Parameters

- **sess** – The current tf session.
- **dataset_name** (*optional*) – A dataset name or a list of dataset names that specifies which dataset(s) to restart. If *None*, all datasets are restart.

initialize_dataset(*sess*, *dataset_name=None*)
Initializes datasets. A dataset must be initialized before being used.

Parameters

- **sess** – The current tf session.
- **dataset_name** (*optional*) – A dataset name or a list of dataset names that specifies which dataset(s) to initialize. If *None*, all datasets are initialized.

get_next()
Returns the next element of the activated dataset.

handle
The handle placeholder that can be fed with a dataset handle to fetch data from the dataset.

TrainTestFeedableDataIterator

class texar.data.**TrainTestFeedableDataIterator**(*train=None*, *val=None*, *test=None*)
Feedable data iterator that alternatives between train, val, and test datasets.

This is a wrapper of *FeedableDataIterator*. The iterator can switch to a dataset and resume from where it was left off when it was visited last time.

`train`, `val`, and `test` can be instance of either tf.data.Dataset or subclass of *DataBase*. At least one of them must be provided.

Parameters

- **train** (*optional*) – Training data.
- **val** (*optional*) – Validation data.
- **test** (*optional*) – Test data.

Example

```
train_data = MonoTextData(hparams={'num_epochs': 200, ...})
test_data = MonoTextData(hparams_test)
iterator = TrainTestFeedableDataIterator(train=train_data,
                                         test=test_data)
batch = iterator.get_next()

sess = tf.Session()

def _eval_epoch(): # Iterate through test data for one epoch
    # Initialize and start from beginning of test data
    iterator.initialize_test_dataset(sess)
    while True:
        try:
```

(continues on next page)

(continued from previous page)

```
            fetch_dict = { # Read from test data
                iterator.handle: Iterator.get_test_handle(sess)
            }
            test_batch_ = sess.run(batch, feed_dict=feed_dict)
        except tf.errors.OutOfRangeError:
            print("End of test epoch.")

# Initialize and start from beginning of training data
iterator.initialize_train_dataset(sess)
step = 0
while True:
    try:
        fetch_dict = { # Read from training data
            iterator.handle: Iterator.get_train_handle(sess)
        }
        train_batch_ = sess.run(batch, fetch_dict=fetch_dict)

        step +=1
        if step % 200 == 0: # Evaluate periodically
            _eval_epoch()
    except tf.errors.OutOfRangeError:
        print("End of training.")
```

get_train_handle(*sess*)

Returns the handle of the training dataset. The handle can be used to feed the `handle` placeholder to fetch training data.

Parameters **sess** – The current tf session.

Returns A string handle to be fed to the `handle` placeholder.

Example

```
next_element = iterator.get_next()
train_handle = iterator.get_train_handle(sess)
# Gets the next training element
ne_ = sess.run(next_element,
               feed_dict={iterator.handle: train_handle})
```

get_val_handle(*sess*)

Returns the handle of the validation dataset. The handle can be used to feed the `handle` placeholder to fetch validation data.

Parameters **sess** – The current tf session.

Returns A string handle to be fed to the `handle` placeholder.

get_test_handle(*sess*)

Returns the handle of the test dataset. The handle can be used to feed the `handle` placeholder to fetch test data.

Parameters **sess** – The current tf session.

Returns A string handle to be fed to the `handle` placeholder.

restart_train_dataset(*sess*)

Restarts the training dataset so that next iteration will fetch data from the beginning of the training dataset.

Parameters **sess** – The current tf session.

restart_val_dataset(*sess*)
Restarts the validation dataset so that next iteration will fetch data from the beginning of the validation dataset.

Parameters **sess** – The current tf session.

restart_test_dataset(*sess*)
Restarts the test dataset so that next iteration will fetch data from the beginning of the test dataset.

Parameters **sess** – The current tf session.

3.2.5 Data Utils

random_shard_dataset

texar.data.**random_shard_dataset**(*dataset_size*, *shard_size*, *seed=None*)
Returns a dataset transformation function that randomly shards a dataset.

maybe_tuple

texar.data.**maybe_tuple**(*data*)
Returns *tuple(data)* if `data` contains more than 1 elements.

Used to wrap *map_func* inputs.

make_partial

texar.data.**make_partial**(*fn*, **args*, ***kwargs*)
Returns a new function with single argument by freezing other arguments of `fn`.

maybe_download

texar.data.**maybe_download**(*urls*, *path*, *filenames=None*, *extract=False*)
Downloads a set of files.

Parameters

- **urls** – A (list of) urls to download files.
- **path** (*str*) – The destination path to save the files.
- **filenames** – A (list of) strings of the file names. If given, must have the same length with `urls`. If *None*, filenames are extracted from `urls`.
- **extract** (*bool*) – Whether to extract compressed files.

Returns A list of paths to the downloaded files.

read_words

texar.data.**read_words**(*filename*, *newline_token=None*)
Reads word from a file.

Parameters

- **filename** (*str*) – Path to the file.
- **newline_token** (*str, optional*) – The token to replace the original newline token "\n". For example, *newline_token=tx.data.SpecialTokens.EOS*. If *None*, no replacement is performed.

Returns A list of words.

make_vocab

texar.data.**make_vocab**(*filenames*, *max_vocab_size=-1*, *newline_token=None*, *return_type='list'*, *return_count=False*)

Builds vocab of the files.

Parameters

- **filenames** (*str*) – A (list of) files.
- **max_vocab_size** (*int*) – Maximum size of the vocabulary. Low frequency words that exceeding the limit will be discarded. Set to *-1* (default) if no truncation is wanted.
- **newline_token** (*str, optional*) – The token to replace the original newline token "\n". For example, *newline_token=tx.data.SpecialTokens.EOS*. If *None*, no replacement is performed.
- **return_type** (*str*) – Either "list" or "dict". If "list" (default), this function returns a list of words sorted by frequency. If "dict", this function returns a dict mapping words to their index sorted by frequency.
- **return_count** (*bool*) – Whether to return word counts. If *True* and `return_type` is "dict", then a count dict is returned, which is a mapping from words to their frequency.

Returns

- If `return_count` is False, returns a list or dict containing the vocabulary words.
- If `return_count` if True, returns a pair of list or dict *(a, b)*, where *a* is a list or dict containing the vocabulary words, *b* is a list of dict containing the word counts.

count_file_lines

texar.data.**count_file_lines**(*filenames*)

Counts the number of lines in the file(s).

make_chained_transformation

texar.data.**make_chained_transformation**(*tran_fns*, **args*, ***kwargs*)

Returns a dataset transformation function that applies a list of transformations sequentially.

Parameters

- **tran_fns** (*list*) – A list of dataset transformation function.
- ***args** – Extra arguments for each of the transformation function.
- ****kwargs** – Extra keyword arguments for each of the transformation function.

Returns A transformation function to be used in tf.data.Dataset.map.

make_combined_transformation

texar.data.**make_combined_transformation**(*tran_fns*, *name_prefix=None*, **args*, ***kwargs*)

Returns a dataset transformation function that applies transformations to each component of the data.

The data to be transformed must be a tuple of the same length of `tran_fns`.

Parameters

- **tran_fns** (*list*) – A list of elements where each element is a transformation function or a list of transformation functions.
- **name_prefix** (*list, optional*) – Prefix to the field names of each component of the data, to prevent fields with the same name in different components from overriding each other. If not *None*, must be of the same length of `tran_fns`.
- ***args** – Extra arguments for each of the transformation function.
- ****kwargs** – Extra keyword arguments for each of the transformation function.

Returns A transformation function to be used in tf.data.Dataset.map.

3.3 Core

3.3.1 Cells

default_rnn_cell_hparams

texar.core.**default_rnn_cell_hparams**()

Returns a *dict* of RNN cell hyperparameters and their default values.

```
{
    "type": "LSTMCell",
    "kwargs": {
        "num_units": 256
    },
    "num_layers": 1,
    "dropout": {
        "input_keep_prob": 1.0,
        "output_keep_prob": 1.0,
        "state_keep_prob": 1.0,
        "variational_recurrent": False,
        "input_size": []
    },
    "residual": False,
    "highway": False,
}
```

Here:

"type" [str or cell class or cell instance] The RNN cell type. This can be

- The string name or full module path of a cell class. If class name is provided, the class must be in module tf.nn.rnn_cell, tf.contrib.rnn, or `texar.custom`.
- A cell class.
- An instance of a cell class. This is not valid if "num_layers" > 1.

For example

```
"type": "LSTMCell" # class name
"type": "tensorflow.contrib.rnn.Conv1DLSTMCell" # module path
"type": "my_module.MyCell" # module path
"type": tf.nn.rnn_cell.GRUCell # class
"type": BasicRNNCell(num_units=100) # cell instance
"type": MyCell(...) # cell instance
```

"kwargs" [dict] Keyword arguments for the constructor of the cell class. A cell is created by `cell_class(**kwargs)`, where *cell_class* is specified in "type" above.

Ignored if "type" is a cell instance.

"num_layers" [int] Number of cell layers. Each layer is a cell created as above, with the same hyperparameters specified in "kwargs".

"dropout" [dict] Dropout applied to the cell in **each** layer. See DropoutWrapper for details of the hyperparameters. If all "*_keep_prob" = 1, no dropout is applied.

Specifically, if "variational_recurrent" = *True*, the same dropout mask is applied across all time steps per run call. If *True*, "input_size" is required, which is a list of input size of each cell layer. The input size of a cell layer is the last dimension size of its input tensor. For example, the input size of the first layer is usually the dimension of word embeddings, while the input size of subsequent layers are usually the *num_units* of the preceding-layer cell. E.g.,

```
# Assume embedding_dim = 100
"type": "LSTMCell",
"kwargs": { "num_units": 123 },
"num_layers": 3,
"dropout": {
    "output_keep_prob": 0.5,
    "variational_recurrent": True,
    "input_size": [100, 123, 123]
}
```

"residual" [bool] If *True*, apply residual connection on the inputs and outputs of cell in **each** layer except the first layer. Ignored if "num_layers" = 1.

"highway" [bool] If True, apply highway connection on the inputs and outputs of cell in each layer except the first layer. Ignored if "num_layers" = 1.

get_rnn_cell

`texar.core.`**`get_rnn_cell`**(*hparams=None*, *mode=None*)

Creates an RNN cell.

See *`default_rnn_cell_hparams()`* for all hyperparameters and default values.

Parameters

- **`hparams`** (*`dict or HParams, optional`*) – Cell hyperparameters. Missing hyperparameters are set to default values.
- **`mode`** (*`optional`*) – A Tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. If *None*, dropout will be controlled by *`texar.global_mode()`*.

Returns A cell instance.

Raises

- **ValueError** – If hparams["num_layers"]>1 and hparams["type"] is a class instance.
- **ValueError** – The cell is not an RNNCell instance.

get_rnn_cell_trainable_variables

texar.core.**get_rnn_cell_trainable_variables**(*cell*)

Returns the list of trainable variables of an RNN cell.

Parameters **cell** – an instance of RNNCell.

Returns trainable variables of the cell.

Return type list

3.3.2 Layers

get_layer

texar.core.**get_layer**(*hparams*)

Makes a layer instance.

The layer must be an instance of tf.layers.Layer.

Parameters **hparams** (*dict or HParams*) – Hyperparameters of the layer, with structure:

```
{
    "type": "LayerClass",
    "kwargs": {
        # Keyword arguments of the layer class
        # ...
    }
}
```

Here:

"type" [str or layer class or layer instance] The layer type. This can be

- The string name or full module path of a layer class. If the class name is provided, the class must be in module tf.layers, texar.core, or texar.custom.
- A layer class.
- An instance of a layer class.

For example

```
"type": "Conv1D" # class name
"type": "texar.core.MaxReducePooling1D" # module path
"type": "my_module.MyLayer" # module path
"type": tf.layers.Conv2D # class
"type": Conv1D(filters=10, kernel_size=2) # cell instance
"type": MyLayer(...) # cell instance
```

"kwargs" [dict] A dictionary of keyword arguments for constructor of the layer class. Ignored if "type" is a layer instance.

- Arguments named "activation" can be a callable, or a *str* of the name or module path to the activation function.

- Arguments named "*_regularizer" and "*_initializer" can be a class instance, or a *dict* of hyperparameters of respective regularizers and initializers. See
- Arguments named "*_constraint" can be a callable, or a *str* of the name or full path to the constraint function.

Returns A layer instance. If hparams["type"] is a layer instance, returns it directly.

Raises

- **`ValueError`** – If `hparams` is *None*.
- **`ValueError`** – If the resulting layer is not an instance of tf.layers.Layer.

MaxReducePooling1D

class texar.core.**MaxReducePooling1D**(*data_format='channels_last'*, *name=None*, ***kwargs*)

A subclass of tf.layers.Layer. Max Pooling layer for 1D inputs. The same as MaxPooling1D except that the pooling dimension is entirely reduced (i.e., *pool_size=input_length*).

AverageReducePooling1D

class texar.core.**AverageReducePooling1D**(*data_format='channels_last'*, *name=None*, ***kwargs*)

A subclass of tf.layers.Layer. Average Pooling layer for 1D inputs. The same as AveragePooling1D except that the pooling dimension is entirely reduced (i.e., *pool_size=input_length*).

get_pooling_layer_hparams

texar.core.**get_pooling_layer_hparams**(*hparams*)

Creates pooling layer hparams *dict* usable for *get_layer()*.

If the `hparams` sets *'pool_size'* to *None*, the layer will be changed to the respective reduce-pooling layer. For example, `tf.layers.MaxPooling1D` is replaced with *MaxReducePooling1D*.

MergeLayer

class texar.core.**MergeLayer**(*layers=None*, *mode='concat'*, *axis=1*, *trainable=True*, *name=None*, ***kwargs*)

A subclass of tf.layers.Layer. A layer that consists of multiple layers in parallel. Input is fed to each of the parallel layers, and the outputs are merged with a specified mode.

Parameters

- **`layers`** (*list, optional*) – A list of tf.layers.Layer instances, or a list of hyperparameter dicts each of which specifies type and kwargs of each layer (see the *hparams* argument of *get_layer()*).

  If *None*, this layer degenerates to a merging operator that merges inputs directly.

- **`mode`** (*str*) – Mode of the merge op. This can be:
  - `'concat'`: Concatenates layer outputs along one axis. Tensors must have the same shape except for the dimension specified in *axis*, which can have different sizes.
  - `'elemwise_sum'`: Outputs element-wise sum.
  - `'elemwise_mul'`: Outputs element-wise product.

- `'sum'`: Computes the sum of layer outputs along the dimension given by *axis*. E.g., given *axis=1*, two tensors of shape *[a, b]* and *[a, c]* respectively will result in a merged tensor of shape *[a]*.
- `'mean'`: Computes the mean of layer outputs along the dimension given in *axis*.
- `'prod'`: Computes the product of layer outputs along the dimension given in *axis*.
- `'max'`: Computes the maximum of layer outputs along the dimension given in *axis*.
- `'min'`: Computes the minimum of layer outputs along the dimension given in *axis*.
- `'and'`: Computes the *logical and* of layer outputs along the dimension given in *axis*.
- `'or'`: Computes the *logical or* of layer outputs along the dimension given in *axis*.
- `'logsumexp'`: Computes log(sum(exp(elements across the dimension of layer outputs)))

• **axis** (*int*) – The axis to use in merging. Ignored in modes `'elemwise_sum'` and `'elemwise_mul'`.

• **trainable** (*bool*) – Whether the layer should be trained.

• **name** (*str, optional*) – Name of the layer.

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Parameters **input_shape** – Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns An input shape tuple.

call(*inputs*)

This is where the layer's logic lives.

Parameters

• **inputs** – Input tensor, or list/tuple of input tensors.

• ****kwargs** – Additional keyword arguments.

Returns A tensor or list/tuple of tensors.

layers

The list of parallel layers.

SequentialLayer

class texar.core.**SequentialLayer**(*layers*, *trainable=True*, *name=None*, ***kwargs*)

A subclass of tf.layers.Layer. A layer that consists of multiple layers connected sequentially.

Parameters **layers** (*list*) – A list of tf.layers.Layer instances, or a list of hyperparameter dicts each of which specifying type and kwargs of each layer (see the *hparams* argument of *get_layer()*). The layers are connected sequentially.

compute_output_shape(*input_shape*)

Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

Parameters **input_shape** – Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

Returns An input shape tuple.

call(*inputs*, *mode=None*)
This is where the layer's logic lives.

Parameters

- **inputs** – Input tensor, or list/tuple of input tensors.
- ****kwargs** – Additional keyword arguments.

Returns A tensor or list/tuple of tensors.

layers
The list of layers connected sequentially.

default_regularizer_hparams

texar.core.**default_regularizer_hparams**()
Returns the hyperparameters and their default values of a variable regularizer:

```
{
    "type": "L1L2",
    "kwargs": {
        "l1": 0.,
        "l2": 0.
    }
}
```

The default value corresponds to L1L2 and, with *(l1=0, l2=0)*, disables regularization.

get_regularizer

texar.core.**get_regularizer**(*hparams=None*)
Returns a variable regularizer instance.

See *default_regularizer_hparams()* for all hyperparameters and default values.

The "type" field can be a subclass of Regularizer, its string name or module path, or a class instance.

Parameters **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparameters are set to default values.

Returns A Regularizer instance. *None* if hparams is *None* or taking the default hyperparameter value.

Raises **ValueError** – The resulting regularizer is not an instance of Regularizer.

get_initializer

texar.core.**get_initializer**(*hparams=None*)
Returns an initializer instance.

Parameters **hparams** (*dict or HParams, optional*) – Hyperparameters with the structure

```
{
    "type": "initializer_class_or_function",
    "kwargs": {
        #...
    }
}
```

The "type" field can be a initializer class, its name or module path, or class instance. If class name is provided, the class must be from one the following modules: tf.initializers, tf.keras.initializers, tf, and `texar.custom`. The class is created by `initializer_class(**kwargs)`. If a class instance is given, "kwargs" is ignored and can be omitted.

Besides, the "type" field can also be an initialization function called with `initialization_fn(**kwargs)`. In this case "type" can be the function, or its name or module path. If function name is provided, the function must be from one of the above modules or module *tf.contrib.layers*. If no keyword argument is required, "kwargs" can be omitted.

Returns An initializer instance. *None* if `hparams` is *None*.

get_activation_fn

`texar.core.`**`get_activation_fn`**(*fn_name='identity'*, *kwargs=None*)

Returns an activation function *fn* with the signature *output = fn(input)*.

If the function specified by `fn_name` has more than one arguments without default values, then all these arguments except the input feature argument must be specified in `kwargs`. Arguments with default values can also be specified in `kwargs` to take values other than the defaults. In this case a partial function is returned with the above signature.

Parameters

- **`fn_name`** (*`str or callable`*) – An activation function, or its name or module path. The function can be:
  - Built-in function defined in tf or tf.nn, e.g., tf.identity.
  - User-defined activation functions in module `texar.custom`.
  - External activation functions. Must provide the full module path, e.g., "my_module.my_activation_fn".
- **`kwargs`** (*`optional`*) – A *dict* or instance of *`HParams`* containing the keyword arguments of the activation function.

Returns An activation function. *None* if `fn_name` is *None*.

get_constraint_fn

`texar.core.`**`get_constraint_fn`**(*fn_name='NonNeg'*)

Returns a constraint function.

The function must follow the signature: `w_ = constraint_fn(w)`.

Parameters **`fn_name`** (*`str or callable`*) – The name or full path to a constraint function, or the function itself.

The function can be:

- Built-in constraint functions defined in modules tf.keras.constraints (e.g., NonNeg) or tf or tf.nn (e.g., activation functions).
- User-defined function in `texar.custom`.
- Externally defined function. Must provide the full path, e.g., *"my_module.my_constraint_fn"*.

If a callable is provided, then it is returned directly.

Returns The constraint function. *None* if `fn_name` is *None*.

default_conv1d_kwargs

`texar.core.`**`default_conv1d_kwargs`**`()`

Returns the default keyword argument values of the constructor of 1D-convolution layer class tf.layers.Conv1D.

```
{
    "filters": 100,
    "kernel_size": 3,
    "strides": 1,
    "padding": 'valid',
    "data_format": 'channels_last',
    "dilation_rate": 1
    "activation": "identity",
    "use_bias": True,
    "kernel_initializer": {
        "type": "glorot_uniform_initializer",
        "kwargs": {}
    },
    "bias_initializer": {
        "type": "zeros_initializer",
        "kwargs": {}
    },
    "kernel_regularizer": {
        "type": "L1L2",
        "kwargs": {
            "l1": 0.,
            "l2": 0.
        }
    },
    "bias_regularizer": {
        # same as in "kernel_regularizer"
        # ...
    },
    "activity_regularizer": {
        # same as in "kernel_regularizer"
        # ...
    },
    "kernel_constraint": None,
    "bias_constraint": None,
    "trainable": True,
    "name": None
}
```

default_dense_kwargs

texar.core.**default_dense_kwargs**()
Returns the default keyword argument values of the constructor of the dense layer class tf.layers.Dense.

```
{
    "units": 256,
    "activation": "identity",
    "use_bias": True,
    "kernel_initializer": {
        "type": "glorot_uniform_initializer",
        "kwargs": {}
    },
    "bias_initializer": {
        "type": "zeros_initializer",
        "kwargs": {}
    },
    "kernel_regularizer": {
        "type": "L1L2",
        "kwargs": {
            "l1": 0.,
            "l2": 0.
        }
    },
    "bias_regularizer": {
        # same as in "kernel_regularizer"
        # ...
    },
    "activity_regularizer": {
        # same as in "kernel_regularizer"
        # ...
    },
    "kernel_constraint": None,
    "bias_constraint": None,
    "trainable": True,
    "name": None
}
```

3.3.3 Optimization

default_optimization_hparams

texar.core.**default_optimization_hparams**()
Returns a *dict* of default hyperparameters of training op and their default values

```
{
    "optimizer": {
        "type": "AdamOptimizer",
        "kwargs": {
            "learning_rate": 0.001
        }
    },
    "learning_rate_decay": {
        "type": "",
        "kwargs": {},
        "min_learning_rate": 0.,
```

(continues on next page)

(continued from previous page)

```
        "start_decay_step": 0,
        "end_decay_step": inf
    },
    "gradient_clip": {
        "type": "",
        "kwargs": {}
    },
    "gradient_noise_scale": None,
    "name": None
}
```

Here:

"optimizer" [dict] Hyperparameters of a tf.train.Optimizer.

- **"type"** specifies the optimizer class. This can be
  - The string name or full module path of an optimizer class. If the class name is provided, the class must be in module tf.train, tf.contrib.opt or `texar.custom`, `texar.core.optimization`
  - An optimizer class.
  - An instance of an optimizer class.

  For example

  ```
  "type": "AdamOptimizer" # class name
  "type": "my_module.MyOptimizer" # module path
  "type": tf.contrib.opt.AdamWOptimizer # class
  "type": my_module.MyOptimizer # class
  "type": GradientDescentOptimizer(learning_rate=0.1) # instance
  "type": MyOptimizer(...) # instance
  ```

- **"kwargs"** is a *dict* specifying keyword arguments for creating the optimizer class instance, with `opt_class(**kwargs)`. Ignored if "type" is a class instance.

"learning_rate_decay" [dict] Hyperparameters of learning rate decay function. The learning rate starts decay from `"start_decay_step"` and keeps unchanged after `"end_decay_step"` or reaching `"min_learning_rate"`.

The decay function is specified in "type" and "kwargs".

- "type" can be a decay function or its name or module path. If function name is provided, it must be from module tf.train or `texar.custom`, `texar.core.optimization`.
- "kwargs" is a *dict* of keyword arguments for the function excluding arguments named "global_step" and "learning_rate".

The function is called with `lr = decay_fn(learning_rate=lr, global_step=offset_step, **kwargs)`, where *offset_step* is the global step offset as above. The only exception is tf.train.piecewise_constant which is called with `lr = piecewise_constant(x=offset_step, **kwargs)`.

"gradient_clip" [dict] Hyperparameters of gradient clipping. The gradient clipping function takes a list of *(gradients, variables)* tuples and returns a list of *(clipped_gradients, variables)* tuples. Typical examples include tf.clip_by_global_norm, tf.clip_by_value, tf.clip_by_norm, tf.clip_by_average_norm, etc.

"type" specifies the gradient clip function, and can be a function, or its name or mudule path. If function name is provided, the function must be from module tf or `texar.custom`, `texar.core.optimization`.

"kwargs" specifies keyword arguments to the function, except arguments named "t" or "t_list".

The function is called with `clipped_grads(, _) = clip_fn(t_list=grads, **kwargs)` (e.g., for tf.clip_by_global_norm) or `clipped_grads = [clip_fn(t=grad, **kwargs) for grad in grads]` (e.g., for tf.clip_by_value).

"gradient_noise_scale" [float, optional] Adds 0-mean normal noise scaled by this value to gradient.

get_train_op

`texar.core.`**`get_train_op`**(*loss*, *variables=None*, *learning_rate=None*, *global_step=None*, *increment_global_step=True*, *hparams=None*)

Creates a training op.

This is a wrapper of tf.contrib.layers.optimize_loss.

Parameters

- **`loss`** – A scalar Tensor representing the loss to minimize.
- **`variables`** (*optional*) – A list of Variables to optimize. If *None*, all trainable variables are used.
- **`learning_rate`** (*float or Tensor, optional*) – If *None*, learning rate specified in `hparams`, or the default learning rate of the optimizer will be used (if exists).
- **`global_step`** (*optional*) – A scalar int Tensor. Step counter to update on each step unless `increment_global_step` is *False*. Learning rate decay uses `global_step`. If *None*, it will be fetched from the default graph (see tf.train.get_global_step for more details). If it has not been created, no step will be incremented with each weight update.
- **`increment_global_step`** (*bool*) – Whether to increment `global_step`. This is useful if the `global_step` is used in multiple training ops per training step (e.g. to optimize different parts of the model) to avoid incrementing `global_step` more times than necessary.
- **`hparams`** (*dict or HParams, optional*) – hyperparameters. Missing hyperparameters are set to default values automatically. See *default_optimization_hparams()* for all hyperparameters and default values.

Returns (train_op, global_step). If `global_step` is provided, the same `global_step` variable is returned, otherwise a new global step is created and returned.

Return type tuple

get_optimizer_fn

`texar.core.`**`get_optimizer_fn`**(*hparams=None*)

Returns a function *optimizer_fn* of making optimizer instance, along with the optimizer class.

The function has the signiture `optimizer_fn(learning_rate=None) -> optimizer class instance`

See the `"optimizer"` field of *default_optimization_hparams()* for all hyperparameters and default values.

The optimizer class must be a subclass of tf.train.Optimizer.

Parameters **`hparams`** (*dict or HParams, optional*) – hyperparameters. Missing hyperparameters are set to default values automatically.

Returns

- If hparams["type"] is a string or optimizer class, returns *(optimizer_fn, optimizer class)*,
- If hparams["type"] is an optimizer instance, returns *(the optimizer instance, optimizer class)*

get_learning_rate_decay_fn

texar.core.**get_learning_rate_decay_fn**(*hparams=None*)

Creates learning rate decay function based on the hyperparameters.

See the `learning_rate_decay` field in *default_optimization_hparams()* for all hyperparameters and default values.

Parameters **hparams** (*dict or HParams, optional*) – hyperparameters. Missing hyperparameters are set to default values automatically.

Returns If hparams["type"] is specified, returns a function that takes *(learning_rate, step, **kwargs)* and returns a decayed learning rate. If hparams["type"] is empty, returns *None*.

Return type function or None

get_gradient_clip_fn

texar.core.**get_gradient_clip_fn**(*hparams=None*)

Creates a gradient clipping function based on the hyperparameters.

See the `gradient_clip` field in *default_optimization_hparams()* for all hyperparameters and default values.

The gradient clipping function takes a list of *(gradients, variables)* tuples and returns a list of *(clipped_gradients, variables)* tuples. Typical examples include tf.clip_by_global_norm, tf.clip_by_value, tf.clip_by_norm, tf.clip_by_average_norm, etc.

Parameters **hparams** (*dict or HParams, optional*) – hyperparameters. Missing hyperparameters are set to default values automatically.

Returns If hparams["type"] is specified, returns the respective function. If hparams["type"] is empty, returns *None*.

Return type function or *None*

3.3.4 Exploration

EpsilonLinearDecayExploration

class texar.core.**EpsilonLinearDecayExploration**(*hparams=None*)

Decays epsilon linearly.

Parameters **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparameters are set to default values. See *default_hparams()* for the defaults.

static default_hparams()

Returns a *dict* of hyperparameters and their default values.

```
{
    'initial_epsilon': 0.1,
    'final_epsilon': 0.0,
    'decay_timesteps': 20000,
    'start_timestep': 0,
    'name': 'epsilon_linear_decay_exploration',
}
```

This specifies the decay process that starts at "start_timestep" with the value "initial_epsilon", and decays for steps "decay_timesteps" to reach the final epsilon value "final_epsilon".

get_epsilon(*timestep*)
Returns the epsilon value.

Parameters **timestep** (*int*) – The time step.

Returns the epsilon value.

Return type float

ExplorationBase

class texar.core.**ExplorationBase**(*hparams=None*)
Base class inherited by all exploration classes.

Parameters **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparameters are set to default values. See *default_hparams()* for the defaults.

static default_hparams()
Returns a *dict* of hyperparameters and their default values.

```
{
    'name': 'exploration_base'
}
```

get_epsilon(*timestep*)
Returns the epsilon value.

Parameters **timestep** (*int*) – The time step.

Returns the epsilon value.

Return type float

hparams
The hyperparameter.

3.3.5 Replay Memories

DequeReplayMemory

class texar.core.**DequeReplayMemory**(*hparams=None*)
A deque based replay memory that accepts new memory entry and deletes oldest memory entry if exceeding the capacity. Memory entries are accessed in random order.

Parameters **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparameters are set to default values. See *default_hparams()* for the defaults.

static default_hparams()
Returns a *dict* of hyperparameters and their default values.

```
{
    'capacity': 80000,
    'name': 'deque_replay_memory',
}
```

Here:

"capacity" [int] Maximum size of memory kept. Deletes oldest memories if exceeds the capacity.

add(*element*)
Appends element to the memory and deletes old memory if exceeds the capacity.

get(*size*)
Randomly samples *size* entries from the memory. Returns a list.

last()
Returns the latest element in the memeory.

size()
Returns the current size of the memory.

ReplayMemoryBase

class texar.core.**ReplayMemoryBase**(*hparams=None*)
Base class of replay memory inheritted by all replay memory classes.

Parameters **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparameters are set to default values. See *default_hparams()* for the defaults.

static default_hparams()
Returns a *dict* of hyperparameters and their default values.

```
{
    'name': 'replay_memory'
}
```

add(*element*)
Inserts a memory entry

get(*size*)
Pops a memory entry.

last()
Returns the latest element in the memeory.

size()
Returns the current size of the memory.

3.4 Modules

3.4.1 ModuleBase

class texar.**ModuleBase**(*hparams=None*)
Base class inherited by modules that create Variables and are configurable through hyperparameters.

A Texar module inheriting *ModuleBase* has following key features:

- **Convenient variable re-use**: A module instance creates its own sets of variables, and automatically re-uses its variables on subsequent calls. Hence TF variable/name scope is transparent to users. For example:

```
encoder = UnidirectionalRNNEncoder(hparams) # create instance
output_1 = encoder(inputs_1) # variables are created
output_2 = encoder(inputs_2) # variables are re-used

print(encoder.trainable_variables) # access trainable variables
# [ ... ]
```

- **Configurable through hyperparameters**: Each module defines allowed hyperparameters and default values. Hyperparameters not specified by users will take default values.
- **Callable**: As the above example, a module instance is "called" with input tensors and returns output tensors. Every call of a module will add ops to the Graph to perform the module's logic.

Parameters **hparams** (*dict, optional*) – Hyperparameters of the module. See *default_hparams()* for the structure and default values.

_build(**args*, ***kwargs*)
Subclass must implement this method to build the logic.

Parameters

- ***args** – Arguments.
- ****kwargs** – Keyword arguments.

Returns Output Tensor(s).

static default_hparams()
Returns a *dict* of hyperparameters of the module with default values. Used to replace the missing values of input *hparams* during module construction.

```
{
    "name": "module"
}
```

variable_scope
The variable scope of the module.

name
The uniquified name of the module.

trainable_variables
The list of trainable variables of the module.

hparams
An *HParams* instance. The hyperparameters of the module.

3.4.2 Embedders

WordEmbedder

class texar.modules.**WordEmbedder**(*init_value=None*, *vocab_size=None*, *hparams=None*)
Simple word embedder that maps indexes into embeddings. The indexes can be soft (e.g., distributions over vocabulary).

Either `init_value` or *vocab_size* is required. If both are given, there must be *init_value.shape[0]==vocab_size*.

Parameters

- **init_value** (*optional*) – A *Tensor* or numpy array that contains the initial value of embeddings. It is typically of shape *[vocab_size] + embedding-dim*. Embedding can have dimensionality > 1.

  If *None*, embedding is initialized as specified in `hparams["initializer"]`. Otherwise, the `"initializer"` and `"dim"` hyperparameters in `hparams` are ignored.

- **vocab_size** (*int, optional*) – The vocabulary size. Required if `init_value` is not given.

- **hparams** (*dict, optional*) – Embedder hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

See *_build()* for the inputs and outputs of the embedder.

Example

```
ids = tf.random_uniform(shape=[32, 10], maxval=10, dtype=tf.int64)
soft_ids = tf.random_uniform(shape=[32, 10, 100])

embedder = WordEmbedder(vocab_size=100, hparams={'dim': 256})
ids_emb = embedder(ids=ids) # shape: [32, 10, 256]
soft_ids_emb = embedder(soft_ids=soft_ids) # shape: [32, 10, 256]
```

```
## Use with Texar data module
hparams={
    'dataset': {
        'embedding_init': {'file': 'word2vec.txt'}
        ...
    },
}
data = MonoTextData(data_params)
iterator = DataIterator(data)
batch = iterator.get_next()

# Use data vocab size
embedder_1 = WordEmbedder(vocab_size=data.vocab.size)
emb_1 = embedder_1(batch['text_ids'])

# Use pre-trained embedding
embedder_2 = WordEmbedder(init_value=data.embedding_init_value)
emb_2 = embedder_2(batch['text_ids'])
```

_build(*ids=None*, *soft_ids=None*, *mode=None*, ***kwargs*)

Embeds (soft) ids.

Either `ids` or `soft_ids` must be given, and they must not be given at the same time.

Parameters

- **ids** (*optional*) – An integer tensor containing the ids to embed.

- **soft_ids** (*optional*) – A tensor of weights (probabilities) used to mix the embedding vectors.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. If *None*, dropout is controlled by `texar.global_mode()`.
- **kwargs** – Additional keyword arguments for tf.nn.embedding_lookup besides `params` and `ids`.

Returns

If `ids` is given, returns a Tensor of shape *shape(ids) + embedding-dim*. For example, if *shape(ids) = [batch_size, max_time]* and *shape(embedding) = [vocab_size, emb_dim]*, then the return tensor has shape *[batch_size, max_time, emb_dim]*.

If `soft_ids` is given, returns a Tensor of shape *shape(soft_ids)[:-1] + embdding-dim*. For example, if *shape(soft_ids) = [batch_size, max_time, vocab_size]* and *shape(embedding) = [vocab_size, emb_dim]*, then the return tensor has shape *[batch_size, max_time, emb_dim]*.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "dim": 100,
    "dropout_rate": 0,
    "dropout_strategy": 'element',
    "trainable": True,
    "initializer": {
        "type": "random_uniform_initializer",
        "kwargs": {
            "minval": -0.1,
            "maxval": 0.1,
            "seed": None
        }
    },
    "regularizer": {
        "type": "L1L2",
        "kwargs": {
            "l1": 0.,
            "l2": 0.
        }
    },
    "name": "word_embedder",
}
```

Here:

"dim" [int or list] Embedding dimension. Can be a list of integers to yield embeddings with dimensionality > 1.

Ignored if `init_value` is given to the embedder constructor.

"dropout_rate" [float] The dropout rate between 0 and 1. E.g., *dropout_rate=0.1* would drop out 10% of the embedding. Set to 0 to disable dropout.

"dropout_strategy" [str] The dropout strategy. Can be one of the following

- `"element"`: The regular strategy that drops individual elements of embedding vectors.
- `"item"`: Drops individual items (e.g., words) entirely. E.g., for the word sequence 'the simpler the better', the strategy can yield '_ simpler the better', where the first *the* is dropped.

- `"item_type"`: Drops item types (e.g., word types). E.g., for the above sequence, the strategy can yield '_ simpler _ better', where the word type 'the' is dropped. The dropout will never yield '_ simpler the better' as in the 'item' strategy.

"trainable" [bool] Whether the embedding is trainable.

"initializer" [dict or None] Hyperparameters of the initializer for embedding values. See *get_initializer()* for the details. Ignored if `init_value` is given to the embedder constructor.

"regularizer" [dict] Hyperparameters of the regularizer for embedding values. See *get_regularizer()* for the details.

"name" [str] Name of the embedding variable.

embedding
The embedding tensor, of shape *[vocab_size] + dim*.

dim
The embedding dimension.

vocab_size
The vocabulary size.

PositionEmbedder

class texar.modules.**PositionEmbedder**(*init_value=None*, *position_size=None*, *hparams=None*)

Simple position embedder that maps position indexes into embeddings via lookup.

Either `init_value` or *position_size* is required. If both are given, there must be *init_value.shape[0]==position_size*.

Parameters

- **init_value** (*optional*) – A *Tensor* or numpy array that contains the initial value of embeddings. It is typically of shape *[position_size, embedding dim]*

  If *None*, embedding is initialized as specified in `hparams["initializer"]`. Otherwise, the `"initializer"` and `"dim"` hyperparameters in `hparams` are ignored.

- **position_size** (*int, optional*) – The number of possible positions, e.g., the maximum sequence length. Required if `init_value` is not given.

- **hparams** (*dict, optional*) – Embedder hyperparameters. If it is not specified, the default hyperparameter setting is used. See *default_hparams* for the sturcture and default values.

_build(*positions=None*, *sequence_length=None*, *mode=None*, ***kwargs*)
Embeds the positions.

Either `position` or `sequence_length` is required:

- If both are given, `sequence_length` is used to mask out embeddings of those time steps beyond the respective sequence lengths.
- If only `sequence_length` is given, then positions from *0* to *sequence_length-1* are embedded.

Parameters

- **positions** (*optional*) – An integer tensor containing the position ids to embed.

- **sequence_length** (*optional*) – An integer tensor of shape *[batch_size]*. Time steps beyond the respective sequence lengths will have zero-valued embeddings.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. If *None*, dropout will be controlled by *texar.global_mode()*.
- **kwargs** – Additional keyword arguments for tf.nn.embedding_lookup besides `params` and `ids`.

Returns A *Tensor* of shape *shape(inputs) + embedding dimension*.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "dim": 100,
    "initializer": {
        "type": "random_uniform_initializer",
        "kwargs": {
            "minval": -0.1,
            "maxval": 0.1,
            "seed": None
        }
    },
    "regularizer": {
        "type": "L1L2",
        "kwargs": {
            "l1": 0.,
            "l2": 0.
        }
    },
    "dropout_rate": 0,
    "trainable": True,
    "name": "position_embedder"
}
```

The hyperparameters have the same meaning as those in *texar.modules.WordEmbedder.default_hparams()*.

embedding

The embedding tensor.

dim

The embedding dimension.

position_size

The position size, i.e., maximum number of positions.

SinusoidsPositionEmbedder

class texar.modules.**SinusoidsPositionEmbedder**(*hparams=None*)

Sinusoid position embedder that maps position indexes into embeddings via sinusoid calculation. This module does not have trainable parameters. Used in, e.g., *TransformerEncoder*.

Each channel of the input Tensor is incremented by a sinusoid of a different frequency and phase. This allows attention to learn to use absolute and relative positions.

Timing signals should be added to some precursors of both the query and the memory inputs to attention. The use of relative position is possible because sin(x+y) and cos(x+y) can be experessed in terms of y, sin(x) and

cos(x). In particular, we use a geometric sequence of timescales starting with min_timescale and ending with max_timescale. The number of different timescales is equal to dim / 2. For each timescale, we generate the two sinusoidal signals sin(timestep/timescale) and cos(timestep/timescale). All of these sinusoids are concatenated in the dim dimension.

_build(*positions*)
Embeds.

Parameters **positions** (*optional*) – An integer tensor containing the position ids to embed.

Returns A *Tensor* of shape *[1, position_size, dim]*.

default_hparams()
Returns a dictionary of hyperparameters with default values We use a geometric sequence of timescales starting with min_timescale and ending with max_timescale. The number of different timescales is equal to dim/2.

```
{
    'min_timescale': 1.0,
    'max_timescale': 10000.0,
    'dim': 512,
    'name':'sinusoid_posisiton_embedder',
}
```

EmbedderBase

class texar.modules.**EmbedderBase**(*num_embeds=None*, *hparams=None*)
The base embedder class that all embedder classes inherit.

Parameters

- **num_embeds** (*int, optional*) – The number of embedding elements, e.g., the vocabulary size of a word embedder.
- **hparams** (*dict or HParams, optional*) – Embedder hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    "name": "embedder"
}
```

num_embeds
The number of embedding elements.

3.4.3 Encoders

UnidirectionalRNNEncoder

class texar.modules.**UnidirectionalRNNEncoder**(*cell=None*, *cell_dropout_mode=None*, *output_layer=None*, *hparams=None*)
One directional RNN encoder.

Parameters

- **cell** – (RNNCell, optional) If not specified, a cell is created as specified in `hparams["rnn_cell"]`.
- **cell_dropout_mode** (*optional*) – A Tensor taking value of tf.estimator.ModeKeys, which toggles dropout in the RNN cell (e.g., activates dropout in TRAIN mode). If *None*, `global_mode()` is used. Ignored if `cell` is given.
- **output_layer** (*optional*) – An instance of tf.layers.Layer. Applies to the RNN cell output of each step. If *None* (default), the output layer is created as specified in `hparams["output_layer"]`.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See `default_hparams()` for the hyperparameter sturcture and default values.

See `_build()` for the inputs and outputs of the encoder.

Example

```
# Use with embedder
embedder = WordEmbedder(vocab_size, hparams=emb_hparams)
encoder = UnidirectionalRNNEncoder(hparams=enc_hparams)

outputs, final_state = encoder(
    inputs=embedder(data_batch['text_ids']),
    sequence_length=data_batch['length'])
```

_build(*inputs*, *sequence_length=None*, *initial_state=None*, *time_major=False*, *mode=None*, *return_cell_output=False*, *return_output_size=False*, ***kwargs*)

Encodes the inputs.

Parameters

- **inputs** – A 3D Tensor of shape *[batch_size, max_time, dim]*. The first two dimensions `batch_size` and `max_time` are exchanged if `time_major=True` is specified.
- **sequence_length** (*optional*) – A 1D int tensor of shape *[batch_size]*. Sequence lengths of the batch inputs. Used to copy-through state and zero-out outputs when past a batch element's sequence length.
- **initial_state** (*optional*) – Initial state of the RNN.
- **time_major** (*bool*) – The shape format of the `inputs` and `outputs` Tensors. If *True*, these tensors are of shape *[max_time, batch_size, depth]*. If *False* (default), these tensors are of shape *[batch_size, max_time, depth]*.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. Controls output layer dropout if the output layer is specified with `hparams`. If *None* (default), `texar.global_mode()` is used.
- **return_cell_output** (*bool*) – Whether to return the output of the RNN cell. This is the results prior to the output layer.
- **return_output_size** (*bool*) – Whether to return the size of the output (i.e., the results after output layers).
- ****kwargs** – Optional keyword arguments of tf.nn.dynamic_rnn, such as *swap_memory*, *dtype*, *parallel_iterations*, etc.

Returns

- By default (both *return_cell_output* and *return_output_size* are False), returns a pair `(outputs, final_state)`
  - `outputs`: The RNN output tensor by the output layer (if exists) or the RNN cell (otherwise). The tensor is of shape *[batch_size, max_time, output_size]* if *time_major* is False, or *[max_time, batch_size, output_size]* if *time_major* is True. If RNN cell output is a (nested) tuple of Tensors, then the `outputs` will be a (nested) tuple having the same nest structure as the cell output.
  - `final_state`: The final state of the RNN, which is a Tensor of shape *[batch_size] + cell.state_size* or a (nested) tuple of Tensors if *cell.state_size* is a (nested) tuple.
- If *return_cell_output* is True, returns a triple `(outputs, final_state, cell_outputs)`
  - `cell_outputs`: The outputs by the RNN cell prior to the output layer, having the same structure with `outputs` except for the *output_dim*.
- If *return_output_size* is *True*, returns a tuple `(outputs, final_state, output_size)`
  - `output_size`: A (possibly nested tuple of) int representing the size of `outputs`. If a single int or an int array, then *outputs* has shape *[batch/time, time/batch] + output_size*. If a (nested) tuple, then *output_size* has the same structure as with *outputs*.
- If both *return_cell_output* and *return_output_size* are True, returns `(outputs, final_state, cell_outputs, output_size)`.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "rnn_cell": default_rnn_cell_hparams(),
    "output_layer": {
        "num_layers": 0,
        "layer_size": 128,
        "activation": "identity",
        "final_layer_activation": None,
        "other_dense_kwargs": None,
        "dropout_layer_ids": [],
        "dropout_rate": 0.5,
        "variational_dropout": False
    },
    "name": "unidirectional_rnn_encoder"
}
```

Here:

"rnn_cell" [dict] A dictionary of RNN cell hyperparameters. Ignored if *cell* is given to the encoder constructor.

The default value is defined in *default_rnn_cell_hparams()*.

"output_layer" [dict] Output layer hyperparameters. Ignored if *output_layer* is given to the encoder constructor. Includes:

"num_layers" [int] The number of output (dense) layers. Set to 0 to avoid any output layers applied to the cell outputs..

"layer_size" [int or list] The size of each of the output (dense) layers.

If an *int*, each output layer will have the same size. If a list, the length must equal to `num_layers`.

"activation" [str or callable or None] Activation function for each of the output (dense) layer except for the final layer. This can be a function, or its string name or module path. If function name is given, the function must be from module tf.nn or tf. For example

```
"activation": "relu" # function name
"activation": "my_module.my_activation_fn" # module path
"activation": my_module.my_activation_fn # function
```

Default is *None* which maintains a linear activation.

"final_layer_activation" [str or callable or None] The activation function for the final output layer.

"other_dense_kwargs" [dict or None] Other keyword arguments to construct each of the output dense layers, e.g., *use_bias*. See Dense for the keyword arguments.

"dropout_layer_ids" [int or list] The indexes of layers (starting from *0*) whose inputs are applied with dropout. The index = `num_layers` means dropout applies to the final layer output. E.g.,

```
{
    "num_layers": 2,
    "dropout_layer_ids": [0, 2]
}
```

will leads to a series of layers as *-dropout-layer0-layer1-dropout-*.

The dropout mode (training or not) is controlled by the `mode` argument of *`_build()`*.

"dropout_rate" [float] The dropout rate, between 0 and 1. E.g., *"dropout_rate": 0.1* would drop out 10% of elements.

"variational_dropout": bool Whether the dropout mask is the same across all time steps.

"name" [str] Name of the encoder

cell
The RNN cell.

state_size
The state size of encoder cell.

Same as `encoder.cell.state_size`.

output_layer
The output layer.

BidirectionalRNNEncoder

class texar.modules.**BidirectionalRNNEncoder**(*cell_fw=None*, *cell_bw=None*, *cell_dropout_mode=None*, *output_layer_fw=None*, *output_layer_bw=None*, *hparams=None*)

Bidirectional forward-backward RNN encoder.

Parameters

- **cell_fw**(*RNNCell, optional*) – The forward RNN cell. If not given, a cell is created as specified in `hparams["rnn_cell_fw"]`.

- **cell_bw** (*RNNCell, optional*) – The backward RNN cell. If not given, a cell is created as specified in `hparams["rnn_cell_bw"]`.
- **cell_dropout_mode** (*optional*) – A tensor taking value of tf.estimator.ModeKeys, which toggles dropout in the RNN cells (e.g., activates dropout in TRAIN mode). If *None*, *global_mode()* is used. Ignored if respective cell is given.
- **output_layer_fw** (*optional*) – An instance of tf.layers.Layer. Apply to the forward RNN cell output of each step. If *None* (default), the output layer is created as specified in `hparams["output_layer_fw"]`.
- **output_layer_bw** (*optional*) – An instance of tf.layers.Layer. Apply to the backward RNN cell output of each step. If *None* (default), the output layer is created as specified in `hparams["output_layer_bw"]`.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

See *_build()* for the inputs and outputs of the encoder.

Example

```
# Use with embedder
embedder = WordEmbedder(vocab_size, hparams=emb_hparams)
encoder = BidirectionalRNNEncoder(hparams=enc_hparams)

outputs, final_state = encoder(
    inputs=embedder(data_batch['text_ids']),
    sequence_length=data_batch['length'])
# outputs == (outputs_fw, outputs_bw)
# final_state == (final_state_fw, final_state_bw)
```

_build(*inputs*, *sequence_length=None*, *initial_state_fw=None*, *initial_state_bw=None*, *time_major=False*, *mode=None*, *return_cell_output=False*, *return_output_size=False*, ***kwargs*)

Encodes the inputs.

Parameters

- **inputs** – A 3D Tensor of shape *[batch_size, max_time, dim]*. The first two dimensions *batch_size* and *max_time* may be exchanged if *time_major=True* is specified.
- **sequence_length** (*optional*) – A 1D int tensor of shape *[batch_size]*. Sequence lengths of the batch inputs. Used to copy-through state and zero-out outputs when past a batch element's sequence length.
- **initial_state** (*optional*) – Initial state of the RNN.
- **time_major** (*bool*) – The shape format of the `inputs` and `outputs` Tensors. If *True*, these tensors are of shape *[max_time, batch_size, depth]*. If *False* (default), these tensors are of shape *[batch_size, max_time, depth]*.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. Controls output layer dropout if the output layer is specified with `hparams`. If *None* (default), *texar.global_mode()* is used.
- **return_cell_output** (*bool*) – Whether to return the output of the RNN cell. This is the results prior to the output layer.

- ****kwargs** – Optional keyword arguments of tf.nn.dynamic_rnn, such as *swap_memory*, *dtype*, *parallel_iterations*, etc.

Returns

- By default (both *return_cell_output* and *return_output_size* are False), returns a pair `(outputs, final_state)`
  - `outputs`: A tuple *(outputs_fw, outputs_bw)* containing the forward and the backward RNN outputs, each of which is of shape *[batch_size, max_time, output_dim]* if *time_major* is False, or *[max_time, batch_size, output_dim]* if *time_major* is True. If RNN cell output is a (nested) tuple of Tensors, then *outputs_fw* and *outputs_bw* will be a (nested) tuple having the same structure as the cell output.
  - `final_state`: A tuple *(final_state_fw, final_state_bw)* containing the final states of the forward and backward RNNs, each of which is a Tensor of shape *[batch_size] + cell.state_size*, or a (nested) tuple of Tensors if *cell.state_size* is a (nested) tuple.
- If *return_cell_output* is True, returns a triple `(outputs, final_state, cell_outputs)` where
  - `cell_outputs`: A tuple *(cell_outputs_fw, cell_outputs_bw)* containting the outputs by the forward and backward RNN cells prior to the output layers, having the same structure with `outputs` except for the *output_dim*.
- If *return_output_size* is True, returns a tuple `(outputs, final_state, output_size)` where
  - `output_size`: A tupple *(output_size_fw, output_size_bw)* containing the size of *outputs_fw* and *outputs_bw*, respectively. Take **_fw* for example, *output_size_fw* is a (possibly nested tuple of) int. If a single int or an int array, then *outputs_fw* has shape *[batch/time, time/batch] + output_size_fw*. If a (nested) tuple, then *output_size_fw* has the same structure as with *outputs_fw*. The same applies to *output_size_bw*.
- If both *return_cell_output* and *return_output_size* are True, returns `(outputs, final_state, cell_outputs, output_size)`.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "rnn_cell_fw": default_rnn_cell_hparams(),
    "rnn_cell_bw": default_rnn_cell_hparams(),
    "rnn_cell_share_config": True,
    "output_layer_fw": {
        "num_layers": 0,
        "layer_size": 128,
        "activation": "identity",
        "final_layer_activation": None,
        "other_dense_kwargs": None,
        "dropout_layer_ids": [],
        "dropout_rate": 0.5,
        "variational_dropout": False
    },
    "output_layer_bw": {
        # Same hyperparams and default values as "output_layer_fw"
        # ...
    },
    "output_layer_share_config": True,
```

(continues on next page)

(continued from previous page)

```
    "name": "bidirectional_rnn_encoder"
}
```

Here:

"rnn_cell_fw" [dict] Hyperparameters of the forward RNN cell. Ignored if *cell_fw* is given to the encoder constructor.

The default value is defined in *default_rnn_cell_hparams()*.

"rnn_cell_bw" [dict] Hyperparameters of the backward RNN cell. Ignored if *cell_bw* is given to the encoder constructor , or if `"rnn_cell_share_config"` is *True*.

The default value is defined in *default_rnn_cell_hparams()*.

"rnn_cell_share_config" [bool] Whether share hyperparameters of the backward cell with the forward cell. Note that the cell parameters (variables) are not shared.

"output_layer_fw" [dict] Hyperparameters of the forward output layer. Ignored if *output_layer_fw* is given to the constructor. See the "output_layer" field of *default_hparams()* for details.

"output_layer_bw" [dict] Hyperparameters of the backward output layer. Ignored if *output_layer_bw* is given to the constructor. Have the same structure and defaults with `"output_layer_fw"`.

Ignored if `"output_layer_share_config"` is True.

"output_layer_share_config" [bool] Whether share hyperparameters of the backward output layer with the forward output layer. Note that the layer parameters (variables) are not shared.

"name" [str] Name of the encoder

cell_fw
The forward RNN cell.

cell_bw
The backward RNN cell.

state_size_fw
The state size of the forward encoder cell.

Same as `encoder.cell_fw.state_size`.

state_size_bw
The state size of the backward encoder cell.

Same as `encoder.cell_bw.state_size`.

output_layer_fw
The output layer of the forward RNN.

output_layer_bw
The output layer of the backward RNN.

HierarchicalRNNEncoder

class texar.modules.**HierarchicalRNNEncoder**(*encoder_major=None*, *encoder_minor=None*, *hparams=None*)

A hierarchical encoder that stacks basic RNN encoders into two layers. Can be used to encode long, structured sequences, e.g. paragraphs, dialog history, etc.

Parameters

- **encoder_major** (*optional*) – An instance of subclass of *RNNEncoderBase* The high-level encoder taking final states from low-level encoder as its inputs. If not specified, an encoder is created as specified in `hparams["encoder_major"]`.
- **encoder_minor** (*optional*) – An instance of subclass of *RNNEncoderBase* The low-level encoder. If not specified, an encoder is created as specified in `hparams["encoder_minor"]`.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

See *_build()* for the inputs and outputs of the encoder.

_build(*inputs*, *order='btu'*, *medium=None*, *sequence_length_major=None*, *sequence_length_minor=None*, ***kwargs*)
Encodes the inputs.

Parameters

- **inputs** – A 4-D tensor of shape *[B, T, U, dim]*, where
  - B: batch_size
  - T: the max length of high-level sequences. E.g., the max number of utterances in dialog history.
  - U: the max length of low-level sequences. E.g., the max length of each utterance in dialog history.
  - dim: embedding dimension

  The order of first three dimensions can be changed according to `order`.
- **order** – A 3-char string containing 'b', 't', and 'u', that specifies the order of inputs dimensions above. Following four can be accepted:
  - **'btu'**: None of the encoders are time-major.
  - **'utb'**: Both encoders are time-major.
  - **'tbu'**: The major encoder is time-major.
  - **'ubt'**: The minor encoder is time-major.
- **medium** (*optional*) – A list of callables that subsequently process the final states of minor encoder and obtain the inputs for the major encoder. If not specified, *flatten()* is used for processing the minor's final states.
- **sequence_length_major** (*optional*) – The *sequence_length* argument sent to major encoder. This is a 1-D Tensor of shape *[B]*.
- **sequence_length_minor** (*optional*) – The *sequence_length* argument sent to minor encoder. It can be either a 1-D Tensor of shape *[B*T]*, or a 2-D Tensor of shape *[B, T]* or *[T, B]* according to `order`.
- ****kwargs** – Other keyword arguments for the major and minor encoders, such as *initial_state*, etc. Note that *sequence_length*, and *time_major* must not be included here. *time_major* is derived from `order` automatically. By default, arguments will be sent to both major and minor encoders. To specify which encoder an argument should be sent to, add '_minor'/'_major' as its suffix.

  Note that *initial_state_minor* must have a batch dimension of size *B*T*. If you have an initial state of batch dimension = *T*, use *tile_initial_state_minor()* to tile it according to *order*.

Returns

A tuple *(outputs, final_state)* by the major encoder.

See the return values of *_build()* method of respective encoder class for details.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "encoder_major_type": "UnidirectionalRNNEncoder",
    "encoder_major_hparams": {},
    "encoder_minor_type": "UnidirectionalRNNEncoder",
    "encoder_minor_hparams": {},
    "config_share": False,
    "name": "hierarchical_encoder_wrapper"
}
```

Here:

"encoder_major_type" [str or class or instance] The high-level encoder. Can be a RNN encoder class, its name or module path, or a class instance. Ignored if *encoder_major* is given to the encoder constructor.

"encoder_major_hparams" [dict] The hyperparameters for the high-level encoder. The high-level encoder is created with `encoder_class(hparams=encoder_major_hparams)`. Ignored if *encoder_major* is given to the encoder constructor, or if "encoder_major_type" is an encoder instance.

"encoder_minor_type" [str or class or instance] The low-level encoder. Can be a RNN encoder class, its name or module path, or a class instance. Ignored if *encoder_minor* is given to the encoder constructor, or if "config_share" is True.

"encoder_minor_hparams" [dict] The hyperparameters for the low-level encoder. The high-level encoder is created with `encoder_class(hparams=encoder_minor_hparams)`. Ignored if *encoder_minor* is given to the encoder constructor, or if "config_share" is True, or if "encoder_minor_type" is an encoder instance.

"config_share": Whether to use encoder_major's hyperparameters to construct encoder_minor.

"name": Name of the encoder.

static tile_initial_state_minor(*initial_state*, *order*, *inputs_shape*)

Tiles an initial state to be used for encoder minor.

The batch dimension of `initial_state` must equal *T*. The state will be copied for *B* times and used to start encoding each low-level sequence. For example, the first utterance in each dialog history in the batch will have the same initial state.

Parameters

- **initial_state** – Initial state with the batch dimension of size *T*.
- **order** (*str*) – The dimension order of inputs. Must be the same as used in *_build()*.
- **inputs_shape** – Shape of *inputs* for *_build()*. Can usually be Obtained with *tf.shape(inputs)*.

Returns A tiled initial state with batch dimension of size *B*T*

static flatten(*x*)

Flattens a cell state by concatenating a sequence of cell states along the last dimension. If the cell states are LSTMStateTuple, only the hidden *LSTMStateTuple.h* is used.

This process is used by default if `medium` is not provided to *_build()*.

encoder_major
The high-level encoder.

encoder_minor
The low-level encoder.

MultiheadAttentionEncoder

class texar.modules.**MultiheadAttentionEncoder**(*hparams=None*)
Multihead Attention Encoder

Parameters **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

_build(*queries*, *memory*, *memory_attention_bias*, *cache=None*, *mode=None*)
Encodes the inputs.

Parameters

- **queries** – A 3d tensor with shape of [batch, length_query, depth_query].
- **memory** – A 3d tensor with shape of [batch, length_key, depth_key].
- **memory_attention_bias** – A 3d tensor with shape of [batch, length_key, num_units].
- **cache** – Memory cache only when inferencing the sentence from sractch.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL* and *PREDICT*. Controls dropout mode. If *None* (default), *texar.global_mode()* is used.

Returns A Tensor of shape *[batch_size, max_time, dim]* containing the encoded vectors.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    "initializer": None,
    'num_heads': 8,
    'output_dim': 512,
    'num_units': 512,
    'dropout_rate': 0.1,
    'use_bias': False,
    "name": "multihead_attention"
}
```

Here:

"initializer" [dict, optional] Hyperparameters of the default initializer that initializes variables created in this module. See *get_initializer()* for details.

"num_heads" [int] Number of heads for attention calculation.

"output_dim" [int] Output dimension of the returned tensor.

"num_units" [int] Hidden dimension of the unsplitted attention space. Should be devicible by *num_heads*.

"dropout_rate": [float] Dropout rate in the attention.

"use_bias": bool Use bias when projecting the key, value and query.

"name" [str] Name of the module.

TransformerEncoder

class texar.modules.**TransformerEncoder**(*hparams=None*)

Transformer encoder that applies multi-head self attention for encoding sequences. Stacked *~texar.modules.encoders.MultiheadAttentionEncoder*, *~texar.modules.FeedForwardNetwork* and residual connections. :param hparams: Hyperparameters. Missing

hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

_build(*inputs*, *sequence_length*, *mode=None*)

Encodes the inputs.

Parameters

- **inputs** – A 3D Tensor of shape *[batch_size, max_time, dim]*, containing the word embeddings of input sequences. Note that the embedding dimension *dim* must equal "dim" in hparams.
- **sequence_length** – A 1D Tensor of shape *[batch_size]*. Input tokens beyond respective sequence lengths are masked out automatically.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. Used to toggle dropout. If *None* (default), *texar.global_mode()* is used.

Returns A Tensor of shape *[batch_size, max_time, dim]* containing the encoded vectors.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "num_blocks": 6,
    "dim": 512,
    'position_embedder_type': 'sinusoids',
    'position_size': None,
    'position_embedder_hparams': None,
    "embedding_dropout": 0.1,
    "residual_dropout": 0.1,
    "poswise_feedforward": default_transformer_poswise_net_hparams,
    'multihead_attention': {
        'name': 'multihead_attention',
        'num_units': 512,
        'output_dim': 512,
        'num_heads': 8,
        'dropout_rate': 0.1,
        'output_dim': 512,
        'use_bias': False,
    },
    "initializer": None,
    "name": "transformer_encoder"
    'use_bert_config': False,
}
```

Here:

"num_blocks" [int] Number of stacked blocks.

"dim" [int] Hidden dimension of the encoders.

"use_bert_config": bool If False, apply the default Transformer Encoder architecture. If True, apply the Transformer Encoder architecture used in BERT. The differences lie in:

1. The Normalization of the input embedding with dimension
2. The attention bias for padding tokens.
3. The residual connections between the internal tensors.

"position_embedder_type": Choose from "sinusoids" or "variables".

"sinusoids": create the position embedding as sinusoids, which is fixed.

"variables": create the position embedding as trainable variables.

"position_size": int The size of position embeddings. Only be used when "position_embedder_type"is "variables".

"position_embedder_hparams" [dict, optional] Hyperparameters of a *PositionEmbedder* as position embedder if "position_embedder_type" is "variables", or Hyperparameters of a *SinusoidsPositionEmbedder* as position embedder if "position_embedder_type" is "sinusoids".

"embedding_dropout" [float] Dropout rate of the input word and position embeddings.

"residual_dropout" [float] Dropout rate of the residual connections.

"poswise_feedforward" [dict,] Hyperparameters for a feed-forward network used in residual connections. Make sure the dimension of the output tensor is equal to *dim*.

See *default_transformer_poswise_net_hparams()* for details.

"multihead_attention": dict, Hyperparameters for the multihead attention strategy. Make sure the "output_dim" in this module is equal to "dim". See :func:

~texar.modules.encoder.MultiheadAttentionEncoder. default_harams for details.

"initializer" [dict, optional] Hyperparameters of the default initializer that initializes variables created in this module. See *get_initializer()* for details.

"name" [str] Name of the module.

Conv1DEncoder

class texar.modules.**Conv1DEncoder**(*hparams=None*)

Simple Conv-1D encoder which consists of a sequence of conv layers followed with a sequence of dense layers.

Wraps *Conv1DNetwork* to be a subclass of *EncoderBase*. Has exact the same functionality with *Conv1DNetwork*.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

The same as *default_hparams()* of *Conv1DNetwork*, except that the default name is 'conv_encoder'.

EncoderBase

class texar.modules.**EncoderBase**(*hparams=None*)
Base class inherited by all encoder classes.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

RNNEncoderBase

class texar.modules.**RNNEncoderBase**(*hparams=None*)
Base class for all RNN encoder classes to inherit.

Parameters **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    "name": "rnn_encoder"
}
```

default_transformer_poswise_net_hparams

texar.modules.**default_transformer_poswise_net_hparams**(*output_dim=512*)
Returns default hyperparameters of a *FeedForwardNetwork* as a pos-wise network used in *TransformerEncoder* and *TransformerDecoder*.

This is a 2-layer dense network with dropout in-between.

```
{
    "layers": [
        {
            "type": "Dense",
            "kwargs": {
                "name": "conv1",
                "units": output_dim*4,
                "activation": "relu",
                "use_bias": True,
            }
        },
        {
            "type": "Dropout",
            "kwargs": {
                "rate": 0.1,
            }
        },
        {
            "type": "Dense",
            "kwargs": {
                "name": "conv2",
                "units": output_dim,
                "use_bias": True,
            }
```

(continues on next page)

(continued from previous page)

```
        }
    ],
    "name": "ffn"
}
```

Parameters **output_dim** (*int*) – The size of output dense layer.

3.4.4 Decoders

RNNDecoderBase

class texar.modules.**RNNDecoderBase**(*cell=None*, *vocab_size=None*, *output_layer=None*, *cell_dropout_mode=None*, *hparams=None*)

Base class inherited by all RNN decoder classes. See *BasicRNNDecoder* for the argumenrts.

See *_build()* for the inputs and outputs of RNN decoders in general.

_build(*decoding_strategy='train_greedy'*, *initial_state=None*, *inputs=None*, *sequence_length=None*, *embedding=None*, *start_tokens=None*, *end_token=None*, *softmax_temperature=None*, *max_decoding_length=None*, *impute_finished=False*, *output_time_major=False*, *input_time_major=False*, *helper=None*, *mode=None*, ***kwargs*)

Performs decoding. This is a shared interface for both *BasicRNNDecoder* and *AttentionRNNDecoder*.

The function provides **3 ways** to specify the decoding method, with varying flexibility:

1. The `decoding_strategy` argument: A string taking value of:

   - **"train_greedy"**: decoding in teacher-forcing fashion (i.e., feeding *ground truth* to decode the next step), and each sample is obtained by taking the *argmax* of the RNN output logits. Arguments `(inputs, sequence_length, input_time_major)` are required for this strategy, and argument `embedding` is optional.
   - **"infer_greedy"**: decoding in inference fashion (i.e., feeding the *generated* sample to decode the next step), and each sample is obtained by taking the *argmax* of the RNN output logits. Arguments `(embedding, start_tokens, end_token)` are required for this strategy, and argument `max_decoding_length` is optional.
   - **"infer_sample"**: decoding in inference fashion, and each sample is obtained by *random sampling* from the RNN output distribution. Arguments `(embedding, start_tokens, end_token)` are required for this strategy, and argument `max_decoding_length` is optional.

   This argument is used only when argument `helper` is *None*.

   Example:

```
embedder = WordEmbedder(vocab_size=data.vocab.size)
decoder = BasicRNNDecoder(vocab_size=data.vocab.size)

# Teacher-forcing decoding
outputs_1, _, _ = decoder(
    decoding_strategy='train_greedy',
    inputs=embedder(data_batch['text_ids']),
    sequence_length=data_batch['length']-1)
```

(continues on next page)

(continued from previous page)

```
# Random sample decoding. Gets 100 sequence samples
outputs_2, _, sequence_length = decoder(
    decoding_strategy='infer_sample',
    start_tokens=[data.vocab.bos_token_id]*100,
    end_token=data.vocab.eos.token_id,
    embedding=embedder,
    max_decoding_length=60)
```

2. The `helper` argument: An instance of subclass of tf.contrib.seq2seq.Helper. This provides a superset of decoding strategies than above, for example:

   - TrainingHelper corresponding to the "train_greedy" strategy.
   - ScheduledEmbeddingTrainingHelper and ScheduledOutputTrainingHelper for scheduled sampling.
   - *`SoftmaxEmbeddingHelper`* and *`GumbelSoftmaxEmbeddingHelper`* for soft decoding and gradient backpropagation.

   This means gives the maximal flexibility of configuring the decoding strategy.

   Example:

```
embedder = WordEmbedder(vocab_size=data.vocab.size)
decoder = BasicRNNDecoder(vocab_size=data.vocab.size)

# Teacher-forcing decoding, same as above with
# `decoding_strategy='train_greedy'`
helper_1 = tf.contrib.seq2seq.TrainingHelper(
    inputs=embedders(data_batch['text_ids']),
    sequence_length=data_batch['length']-1)
outputs_1, _, _ = decoder(helper=helper_1)

# Gumbel-softmax decoding
helper_2 = GumbelSoftmaxEmbeddingHelper(
    embedding=embedder,
    start_tokens=[data.vocab.bos_token_id]*100,
    end_token=data.vocab.eos_token_id,
    tau=0.1)
outputs_2, _, sequence_length = decoder(
    max_decoding_length=60, helper=helper_2)
```

3. `hparams["helper_train"]` and `hparams["helper_infer"]`: Specifying the helper through hyperparameters. Train and infer strategy is toggled based on `mode`. Appriopriate arguments (e.g., `inputs`, `start_tokens`, etc) are selected to construct the helper. Additional arguments for helper constructor can be provided either through `**kwargs`, or through `hparams["helper_train/infer"]["kwargs"]`.

   This means is used only when both `decoding_strategy` and `helper` are *None*.

   Example:

```
h = {
    "helper_infer": {
        "type": "GumbelSoftmaxEmbeddingHelper",
        "kwargs": { "tau": 0.1 }
    }
```

(continues on next page)

(continued from previous page)

```
}
embedder = WordEmbedder(vocab_size=data.vocab.size)
decoder = BasicRNNDecoder(vocab_size=data.vocab.size, hparams=h)

# Gumbel-softmax decoding
output, _, _ = decoder(
    decoding_strategy=None, # Sets to None explicit
    embedding=embedder,
    start_tokens=[data.vocab.bos_token_id]*100,
    end_token=data.vocab.eos_token_id,
    max_decoding_length=60,
    mode=tf.estimator.ModeKeys.PREDICT)
        # PREDICT mode also shuts down dropout
```

Parameters

- **decoding_strategy** (*str*) – A string specifying the decoding strategy. Different arguments are required based on the strategy. Ignored if `helper` is given.
- **initial_state** (*optional*) – Initial state of decoding. If *None* (default), zero state is used.
- **inputs** (*optional*) – Input tensors for teacher forcing decoding. Used when *decoding_strategy* is set to "train_greedy", or when *hparams*-configured helper is used.
  - If `embedding` is *None*, *inputs* is directly fed to the decoder. E.g., in *"train_greedy"* strategy, *inputs* must be a 3D Tensor of shape *[batch_size, max_time, emb_dim]* (or *[max_time, batch_size, emb_dim]* if 'input_time_major'==True).
  - If *embedding* is given, *inputs* is used as index to look up embeddings and feed in the decoder. E.g., if *embedding* is an instance of *WordEmbedder*, then `inputs` is usually a 2D int Tensor *[batch_size, max_time]* (or *[max_time, batch_size]* if 'input_time_major'==True) containing the token indexes.
- **sequence_length** (*optional*) – A 1D int Tensor containing the sequence length of `inputs`. Used when *decoding_strategy="train_greedy"* or *hparams*-configured helper is used.
- **embedding** (*optional*) – A callable that returns embedding vectors of *inputs* (e.g., an instance of subclass of *EmbedderBase*), or the *params* argument of tf.nn.embedding_lookup. If provided, *inputs* (if used) will be passed to *embedding* to fetch the embedding vectors of the inputs. Required when *decoding_strategy="infer_greedy"* or *"infer_sample"*; optional when *decoding_strategy="train_greedy"*.
- **start_tokens** (*optional*) – A int Tensor of shape *[batch_size]*, the start tokens. Used when *decoding_strategy="infer_greedy"* or *"infer_sample"*, or when *hparams*-configured helper is used. Companying with Texar data module, to get *batch_size* samples where batch_size is changing according to the data module, this can be set as *start_tokens=tf.ones_like(batch['length'])*bos_token_id*.
- **end_token** (*optional*) – A int 0D Tensor, the token that marks end of decoding. Used when *decoding_strategy="infer_greedy"* or *"infer_sample"*, or when *hparams*-configured helper is used.
- **softmax_temperature** (*optional*) – A float 0D Tensor, value to divide the logits by before computing the softmax. Larger values (above 1.0) result in more random samples. Must > 0. If *None*, 1.0 is used. Used when *decoding_strategy="infer_sample"*.

- **max_decoding_length** – A int scalar Tensor indicating the maximum allowed number of decoding steps. If *None* (default), either *hparams["max_decoding_length_train"]* or *hparams["max_decoding_length_infer"]* is used according to `mode`.
- **impute_finished** (*bool*) – If *True*, then states for batch entries which are marked as finished get copied through and the corresponding outputs get zeroed out. This causes some slowdown at each time step, but ensures that the final state and outputs have the correct values and that backprop ignores time steps that were marked as finished.
- **output_time_major** (*bool*) – If *True*, outputs are returned as time major tensors. If *False* (default), outputs are returned as batch major tensors.
- **input_time_major** (*optional*) – Whether the `inputs` tensor is time major. Used when *decoding_strategy="train_greedy"* or *hparams*-configured helper is used.
- **helper** (*optional*) – An instance of Helper that defines the decoding strategy. If given, *decoding_strategy* and helper configs in *hparams* are ignored.
- **mode** (*str, optional*) – A string taking value in tf.estimator.ModeKeys. If *TRAIN*, training related hyperparameters are used (e.g., *hparams['max_decoding_length_train']*), otherwise, inference related hyperparameters are used (e.g., *hparams['max_decoding_length_infer']*). If *None* (default), *TRAIN* mode is used.
- ****kwargs** – Other keyword arguments for constructing helpers defined by *hparams["helper_trainn"]* or *hparams["helper_infer"]*.

Returns

(outputs, final_state, sequence_lengths), where

- **'outputs'**: an object containing the decoder output on all time steps.
- **'final_state'**: is the cell state of the final time step.
- **'sequence_lengths'**: is an int Tensor of shape *[batch_size]* containing the length of each sample.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

The hyperparameters are the same as in *default_hparams()* of *BasicRNNDecoder*, except that the default "name" here is "rnn_decoder".

batch_size

The batch size of input values.

cell

The RNN cell.

zero_state(*batch_size*, *dtype*)

Zero state of the RNN cell. Equivalent to `decoder.cell.zero_state`.

state_size

The state size of decoder cell. Equivalent to `decoder.cell.state_size`.

hparams

An *HParams* instance. The hyperparameters of the module.

name

The uniquified name of the module.

trainable_variables

The list of trainable variables of the module.

variable_scope
The variable scope of the module.

vocab_size
The vocab size.

output_layer
The output layer.

BasicRNNDecoder

class texar.modules.**BasicRNNDecoder**(*cell=None*, *cell_dropout_mode=None*, *vocab_size=None*, *output_layer=None*, *hparams=None*)

Basic RNN decoder.

Parameters

- **cell** (*RNNCell, optional*) – An instance of RNNCell. If *None* (default), a cell is created as specified in *hparams*.
- **cell_dropout_mode** (*optional*) – A Tensor taking value of tf.estimator.ModeKeys, which toggles dropout in the RNN cell (e.g., activates dropout in TRAIN mode). If *None*, *global_mode()* is used. Ignored if *cell* is given.
- **vocab_size** (*int, optional*) – Vocabulary size. Required if *output_layer* is *None*.
- **output_layer** (*optional*) – An instance of tf.layers.Layer, or tf.identity. Apply to the RNN cell output to get logits. If *None*, a dense layer is used with output dimension set to *vocab_size*. Set *output_layer=tf.identity* if you do not want to have an output layer after the RNN cell outputs.
- **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

See *_build()* for the inputs and outputs of the decoder. The decoder returns *(outputs, final_state, sequence_lengths)*, where *outputs* is an instance of *BasicRNNDecoderOutput*.

Example

```
embedder = WordEmbedder(vocab_size=data.vocab.size)
decoder = BasicRNNDecoder(vocab_size=data.vocab.size)

# Training loss
outputs, _, _ = decoder(
    decoding_strategy='train_greedy',
    inputs=embedder(data_batch['text_ids']),
    sequence_length=data_batch['length']-1)

loss = tx.losses.sequence_sparse_softmax_cross_entropy(
    labels=data_batch['text_ids'][:, 1:],
    logits=outputs.logits,
    sequence_length=data_batch['length']-1)

# Inference sample
outputs, _, _ = decoder(
```

(continues on next page)

(continued from previous page)

```
    decoding_strategy='infer_sample',
    start_tokens=[data.vocab.bos_token_id]*100,
    end_token=data.vocab.eos.token_id,
    embedding=embedder,
    max_decoding_length=60,
    mode=tf.estimator.ModeKeys.PREDICT)

sample_id = sess.run(outputs.sample_id)
sample_text = tx.utils.map_ids_to_strs(sample_id, data.vocab)
print(sample_text)
# [
#   the first sequence sample .
#   the second sequence sample .
#   ...
# ]
```

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "rnn_cell": default_rnn_cell_hparams(),
    "max_decoding_length_train": None,
    "max_decoding_length_infer": None,
    "helper_train": {
        "type": "TrainingHelper",
        "kwargs": {}
    }
    "helper_infer": {
        "type": "SampleEmbeddingHelper",
        "kwargs": {}
    }
    "name": "basic_rnn_decoder"
}
```

Here:

"rnn_cell" [dict] A dictionary of RNN cell hyperparameters. Ignored if *cell* is given to the decoder constructor. The default value is defined in `default_rnn_cell_hparams()`.

"max_decoding_length_train": int or None Maximum allowed number of decoding steps in training mode. If *None* (default), decoding is performed until fully done, e.g., encountering the <EOS> token. Ignored if *max_decoding_length* is given when calling the decoder.

"max_decoding_length_infer" [int or None] Same as "max_decoding_length_train" but for inference mode.

"helper_train" [dict] The hyperparameters of the helper used in training. "type" can be a helper class, its name or module path, or a helper instance. If a class name is given, the class must be from module tf.contrib.seq2seq, `texar.modules`, or `texar.custom`. This is used only when both *decoding_strategy* and *helper* augments are *None* when calling the decoder. See *_build()* for more details.

"helper_infer": dict Same as "helper_train" but during inference mode.

"name" [str] Name of the decoder.

The default value is "basic_rnn_decoder".

batch_size
The batch size of input values.

cell
The RNN cell.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

output_layer
The output layer.

state_size
The state size of decoder cell. Equivalent to `decoder.cell.state_size`.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

vocab_size
The vocab size.

zero_state(*batch_size*, *dtype*)
Zero state of the RNN cell. Equivalent to `decoder.cell.zero_state`.

BasicRNNDecoderOutput

class texar.modules.**BasicRNNDecoderOutput**
The outputs of basic RNN decoder that include both RNN outputs and sampled ids at each step. This is also used to store results of all the steps after decoding the whole sequence.

logits
The outputs of RNN (at each step/of all steps) by applying the output layer on cell outputs. E.g., in *BasicRNNDecoder* with default hyperparameters, this is a Tensor of shape *[batch_size, max_time, vocab_size]* after decoding the whole sequence.

sample_id
The sampled results (at each step/of all steps). E.g., in BasicRNNDecoder with decoding strategy of train_greedy, this is a Tensor of shape *[batch_size, max_time]* containing the sampled token indexes of all steps.

cell_output
The output of RNN cell (at each step/of all steps). This is the results prior to the output layer. E.g., in BasicRNNDecoder with default hyperparameters, this is a Tensor of shape *[batch_size, max_time, cell_output_size]* after decoding the whole sequence.

AttentionRNNDecoder

class texar.modules.**AttentionRNNDecoder**(*memory*, *memory_sequence_length=None*, *cell=None*, *cell_dropout_mode=None*, *vocab_size=None*, *output_layer=None*, *cell_input_fn=None*, *hparams=None*)
RNN decoder with attention mechanism.

Parameters

- **memory** – The memory to query, e.g., the output of an RNN encoder. This tensor should be shaped *[batch_size, max_time, dim]*.
- **memory_sequence_length** (*optional*) – A tensor of shape *[batch_size]* containing the sequence lengths for the batch entries in memory. If provided, the memory tensor rows are masked with zeros for values past the respective sequence lengths.
- **cell** (*RNNCell, optional*) – An instance of *RNNCell*. If *None*, a cell is created as specified in *hparams*.
- **cell_dropout_mode** (*optional*) – A Tensor taking value of tf.estimator.ModeKeys, which toggles dropout in the RNN cell (e.g., activates dropout in TRAIN mode). If *None*, *global_mode()* is used. Ignored if *cell* is given.
- **vocab_size** (*int, optional*) – Vocabulary size. Required if *output_layer* is *None*.
- **output_layer** (*optional*) – An instance of tf.layers.Layer, or tf.identity. Apply to the RNN cell output to get logits. If *None*, a dense layer is used with output dimension set to *vocab_size*. Set *output_layer=tf.identity* if you do not want to have an output layer after the RNN cell outputs.
- **cell_input_fn** (*callable, optional*) – A callable that produces RNN cell inputs. If *None* (default), the default is used: *lambda inputs, attention: tf.concat([inputs, attention], -1)*, which cancats regular RNN cell inputs with attentions.
- **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

See *_build()* for the inputs and outputs of the decoder. The decoder returns *(outputs, final_state, sequence_lengths)*, where *outputs* is an instance of *AttentionRNNDecoderOutput*.

Example

```
# Encodes the source
enc_embedder = WordEmbedder(data.source_vocab.size, ...)
encoder = UnidirectionalRNNEncoder(...)

enc_outputs, _ = encoder(
    inputs=enc_embedder(data_batch['source_text_ids']),
    sequence_length=data_batch['source_length'])

# Decodes while attending to the source
dec_embedder = WordEmbedder(vocab_size=data.target_vocab.size, ...)
decoder = AttentionRNNDecoder(
    memory=enc_outputs,
    memory_sequence_length=data_batch['source_length'],
    vocab_size=data.target_vocab.size)

outputs, _, _ = decoder(
    decoding_strategy='train_greedy',
    inputs=dec_embedder(data_batch['target_text_ids']),
    sequence_length=data_batch['target_length']-1)
```

static default_hparams()

Returns a dictionary of hyperparameters with default values:

Common hyperparameters are the same as in *BasicRNNDecoder.default_hparams()*. Additional hyperparameters are for attention mechanism configuration.

```
{
    "attention": {
        "type": "LuongAttention",
        "kwargs": {
            "num_units": 256,
        },
        "attention_layer_size": None,
        "alignment_history": False,
        "output_attention": True,
    },
    # The following hyperparameters are the same as with
    # `BasicRNNDecoder`
    "rnn_cell": default_rnn_cell_hparams(),
    "max_decoding_length_train": None,
    "max_decoding_length_infer": None,
    "helper_train": {
        "type": "TrainingHelper",
        "kwargs": {}
    }
    "helper_infer": {
        "type": "SampleEmbeddingHelper",
        "kwargs": {}
    }
    "name": "attention_rnn_decoder"
}
```

Here:

"attention" [dict] Attention hyperparameters, including:

"type" [str or class or instance] The attention type. Can be an attention class, its name or module path, or a class instance. The class must be a subclass of TF AttentionMechanism. If class name is given, the class must be from modules tf.contrib.seq2seq or `texar.custom`.

Example:

```
# class name
"type": "LuongAttention"
"type": "BahdanauAttention"
# module path
"type": "tf.contrib.seq2seq.BahdanauMonotonicAttention"
"type": "my_module.MyAttentionMechanismClass"
# class
"type": tf.contrib.seq2seq.LuongMonotonicAttention
# instance
"type": LuongAttention(...)
```

"kwargs" [dict] keyword arguments for the attention class constructor. Arguments `memory` and `memory_sequence_length` should **not** be specified here because they are given to the decoder constructor. Ignored if "type" is an attention class instance. For example

Example:

```
"type": "LuongAttention",
"kwargs": {
    "num_units": 256,
```

(continues on next page)

(continued from previous page)

```
    "probability_fn": tf.nn.softmax
}
```

Here "probability_fn" can also be set to the string name or module path to a probability function.

"attention_layer_size" [int or None] The depth of the attention (output) layer. The context and cell output are fed into the attention layer to generate attention at each time step. If *None* (default), use the context as attention at each time step.

"alignment_history": bool whether to store alignment history from all time steps in the final output state. (Stored as a time major *TensorArray* on which you must call *stack()*.)

"output_attention": bool If *True* (default), the output at each time step is the attention value. This is the behavior of Luong-style attention mechanisms. If *False*, the output at each time step is the output of *cell*. This is the beahvior of Bhadanau-style attention mechanisms. In both cases, the *attention* tensor is propagated to the next time step via the state and is used there. This flag only controls whether the attention mechanism is propagated up to the next cell in an RNN stack or to the top RNN output.

zero_state(*batch_size*, *dtype*)
Returns zero state of the basic cell. Equivalent to `decoder.cell._cell.zero_state`.

wrapper_zero_state(*batch_size*, *dtype*)
Returns zero state of the attention-wrapped cell. Equivalent to `decoder.cell.zero_state`.

state_size
The state size of the basic cell. Equivalent to `decoder.cell._cell.state_size`.

wrapper_state_size
The state size of the attention-wrapped cell. Equivalent to `decoder.cell.state_size`.

batch_size
The batch size of input values.

cell
The RNN cell.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

output_layer
The output layer.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

vocab_size
The vocab size.

AttentionRNNDecoderOutput

class texar.modules.**AttentionRNNDecoderOutput**

The outputs of attention RNN decoders that additionally include attention results.

logits

The outputs of RNN (at each step/of all steps) by applying the output layer on cell outputs. E.g., in *AttentionRNNDecoder*, this is a Tensor of shape *[batch_size, max_time, vocab_size]* after decoding.

sample_id

The sampled results (at each step/of all steps). E.g., in *AttentionRNNDecoder* with decoding strategy of train_greedy, this is a Tensor of shape *[batch_size, max_time]* containing the sampled token indexes of all steps.

cell_output

The output of RNN cell (at each step/of all steps). This is the results prior to the output layer. E.g., in AttentionRNNDecoder with default hyperparameters, this is a Tensor of shape *[batch_size, max_time, cell_output_size]* after decoding the whole sequence.

attention_scores

A single or tuple of ‘Tensor‘(s) containing the alignments emitted (at the previous time step/of all time steps) for each attention mechanism.

attention_context

The attention emitted (at the previous time step/of all time steps).

beam_search_decode

texar.modules.**beam_search_decode**(*decoder_or_cell*, *embedding*, *start_tokens*, *end_token*, *beam_width*, *initial_state=None*, *tiled_initial_state=None*, *output_layer=None*, *length_penalty_weight=0.0*, *max_decoding_length=None*, *output_time_major=False*, ***kwargs*)

Performs beam search sampling decoding.

Parameters

- **decoder_or_cell** – An instance of subclass of *RNNDecoderBase*, or an instance of RNNCell. The decoder or RNN cell to perform decoding.
- **embedding** – A callable that takes a vector tensor of indexes (e.g., an instance of subclass of *EmbedderBase*), or the `params` argument for tf.nn.embedding_lookup.
- **start_tokens** – *int32* vector shaped *[batch_size]*, the start tokens.
- **end_token** – *int32* scalar, the token that marks end of decoding.
- **beam_width** (*int*) – Python integer, the number of beams.
- **initial_state** (*optional*) – Initial state of decoding. If *None* (default), zero state is used.

  The state must **not** be tiled with tile_batch. If you have an already-tiled initial state, use `tiled_initial_state` instead.

  In the case of attention RNN decoder, *initial_state* must **not** be an AttentionWrapperState. Instead, it must be a state of the wrapped *RNNCell*, which state will be wrapped into *AttentionWrapperState* automatically.

  Ignored if `tiled_initial_state` is given.

- **tiled_initial_state** (*optional*) – Initial state that has been tiled (typicaly with tile_batch) so that the batch dimension has size *batch_size * beam_width*.

  In the case of attention RNN decoder, this can be either a state of the wrapped *RNNCell*, or an *AttentionWrapperState*.

  If not given, `initial_state` is used.

- **output_layer** (*optional*) – A Layer instance to apply to the RNN output prior to storing the result or sampling. If *None* and `decoder_or_cell` is a decoder, the decoder's output layer will be used.
- **length_penalty_weight** – Float weight to penalize length. Disabled with *0.0* (default).
- **max_decoding_length** (*optional*) – A int scalar Tensor indicating the maximum allowed number of decoding steps. If *None* (default), decoding will continue until the end token is encountered.
- **output_time_major** (*bool*) – If *True*, outputs are returned as time major tensors. If *False* (default), outputs are returned as batch major tensors.
- ****kwargs** – Other keyword arguments for dynamic_decode except argument *maximum_iterations* which is set to `max_decoding_length`.

Returns

A tuple *(outputs, final_state, sequence_length)*, where

- outputs: An instance of FinalBeamSearchDecoderOutput.
- final_state: An instance of BeamSearchDecoderState.
- sequence_length: A Tensor of shape *[batch_size]* containing the lengths of samples.

Example

```
## Beam search with basic RNN decoder

embedder = WordEmbedder(vocab_size=data.vocab.size)
decoder = BasicRNNDecoder(vocab_size=data.vocab.size)

outputs, _, _, = beam_search_decode(
    decoder_or_cell=decoder,
    embedding=embedder,
    start_tokens=[data.vocab.bos_token_id] * 100,
    end_token=data.vocab.eos_token_id,
    beam_width=5,
    max_decoding_length=60)

sample_ids = sess.run(outputs.predicted_ids)
sample_text = tx.utils.map_ids_to_strs(sample_id[:,:,0], data.vocab)
print(sample_text)
# [
#   the first sequence sample .
#   the second sequence sample .
#   ...
# ]
```

```
## Beam search with attention RNN decoder

# Encodes the source
enc_embedder = WordEmbedder(data.source_vocab.size, ...)
encoder = UnidirectionalRNNEncoder(...)

enc_outputs, enc_state = encoder(
    inputs=enc_embedder(data_batch['source_text_ids']),
    sequence_length=data_batch['source_length'])

# Decodes while attending to the source
dec_embedder = WordEmbedder(vocab_size=data.target_vocab.size, ...)
decoder = AttentionRNNDecoder(
    memory=enc_outputs,
    memory_sequence_length=data_batch['source_length'],
    vocab_size=data.target_vocab.size)

# Beam search
outputs, _, _, = beam_search_decode(
    decoder_or_cell=decoder,
    embedding=dec_embedder,
    start_tokens=[data.vocab.bos_token_id] * 100,
    end_token=data.vocab.eos_token_id,
    beam_width=5,
    initial_state=enc_state,
    max_decoding_length=60)
```

TransformerDecoder

class texar.modules.**TransformerDecoder**(*embedding*, *hparams=None*)

Transformer decoder that applies multi-head attention for sequence decoding. Stacked *~texar.modules.encoders.MultiheadAttentionEncoder* for encoder-decoder attention and self attention, *~texar.modules.FeedForwardNetwork* and residual connections.

Use the passed *embedding* variable as the parameters of the transform layer from output to logits.

Parameters

- **embedding** – A Tensor of shape *[vocab_size, dim]* containing the word embeddng. The Tensor is used as the decoder output layer.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

_build(*memory*, *memory_sequence_length=None*, *memory_attention_bias=None*, *inputs=None*, *sequence_length=None*, *decoding_strategy='train_greedy'*, *beam_width=1*, *alpha=0*, *start_tokens=None*, *end_token=None*, *max_decoding_length=None*, *mode=None*)

Performs decoding.

The decoder supports 4 decoding strategies. For the first 3 strategies, set `decoding_strategy` to the respective string.

- **"train_greedy"**: decoding in teacher-forcing fashion (i.e., feeding ground truth to decode the next step), and for each step sample is obtained by taking the *argmax* of logits. Argument `inputs` is required for this strategy. `sequence_length` is optional.
- **"infer_greedy"**: decoding in inference fashion (i.e., feeding *generated* sample to decode the next step), and for each step sample is obtained by taking the *argmax* of logits.

Arguments `(start_tokens, end_token)` are required for this strategy, and argument `max_decoding_length` is optional.

- **"infer_sample"**: decoding in inference fashion, and for each step sample is obtained by *random sampling* from the logits. Arguments `(start_tokens, end_token)` are required for this strategy, and argument `max_decoding_length` is optional.
- **Beam Search**: set `beam_width` to > 1 to use beam search decoding. Arguments `(start_tokens, end_token)` are required, and argument `max_decoding_length` is optional.

Parameters

- **memory** – The memory to attend, e.g., the output of an RNN encoder. A Tensor of shape *[batch_size, memory_max_time, dim]*.
- **memory_sequence_length** (*optional*) – A Tensor of shape *[batch_size]* containing the sequence lengths for the batch entries in memory. Used to create attention bias of `memory_attention_bias` is not given. Ignored if *memory_attention_bias* is provided.
- **memory_attention_bias** (*optional*) – A Tensor of shape *[batch_size, num_heads, memory_max_time, dim]*. An attention bias typically sets the value of a padding position to a large negative value for masking. If not given, `memory_sequence_length` is used to automatically create an attention bias.
- **inputs** (*optional*) – Input tensor for teacher forcing decoding, of shape *[batch_size, target_max_time, emb_dim]* containing the target sequence word embeddings. Used when `decoding_strategy` is set to "train_greedy".
- **sequence_length** (*optional*) – A Tensor of shape *[batch_size]*, containing the sequence length of `inputs`. Tokens beyond the respective sequence length are masked out. Used when `decoding_strategy` is set to "train_greedy".
- **decoding_strategy** (*str*) – A string specifying the decoding strategy, including "train_greedy", "infer_greedy", "infer_sample". Different arguments are required based on the strategy. See above for details. Ignored if `beam_width` > 1.
- **beam_width** (*int*) – Set to > 1 to use beam search.
- **alpha** (*float*) – Length penalty coefficient. Refer to https://arxiv.org/abs/1609.08144 for more details.
- **start_tokens** (*optional*) – An int Tensor of shape *[batch_size]*, containing the start tokens. Used when *decoding_strategy* = "infer_greedy" or "infer_sample", or *beam_width* > 1.
- **end_token** (*optional*) – An int 0D Tensor, the token that marks end of decoding. Used when *decoding_strategy* = "infer_greedy" or "infer_sample", or *beam_width* > 1.
- **max_decoding_length** (*optional*) – An int scalar Tensor indicating the maximum allowed number of decoding steps. If *None* (default), use "max_decoding_length" defined in `hparams`. Ignored in "train_greedy" decoding.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. Controls dropout mode. If *None* (default), *texar.global_mode()* is used.

Returns

- For **"train_greedy"** decoding, returns an instance of *TransformerDecoderOutput* which contains *sample_id* and *logits*.

- For **"infer_greedy"** and **"infer_sample"** decoding, returns a tuple *(outputs, sequence_lengths)*, where *outputs* is an instance of `TransformerDecoderOutput` as in "train_greedy", and *sequence_lengths* is a Tensor of shape *[batch_size]* containing the length of each sample.
- For **beam_search** decoding, returns a *dict* containing keys "sample_id" and "log_prob".
  - **"sample_id"** is an int Tensor of shape *[batch_size, max_time, beam_width]* containing generated token indexes. *sample_id[:,:,0]* is the highest-probable sample.
  - **"log_porb"** is a float Tensor of shape *[batch_size, beam_width]* containing the log probability of each sequence sample.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    # Same as in TransformerEncoder
    "num_blocks": 6,
    "dim": 512,
    "position_embedder_hparams": None,
    "embedding_dropout": 0.1,
    "residual_dropout": 0.1,
    "poswise_feedforward": default_transformer_poswise_net_hparams,
    "multihead_attention": {
        "num_units": 512,
        "num_heads": 8,
    },
    "initializer": None,
    # Additional for TransformerDecoder
    "embedding_tie": True,
    "output_layer_bias": False,
    "max_decoding_length": 1e10,
    "name": "transformer_decoder"
}
```

Here:

"num_blocks" [int] Number of stacked blocks.

"dim" [int] Hidden dimension of the encoder.

"position_embedder_hparams" [dict, optional] Hyperparameters of a `SinusoidsPositionEmbedder` as position embedder. If *None*, the `default_hparams()` is used.

"embedding_dropout": float Dropout rate of the input word and position embeddings.

"residual_dropout" [float] Dropout rate of the residual connections.

"poswise_feedforward" [dict,] Hyperparameters for a feed-forward network used in residual connections. Make sure the dimension of the output tensor is equal to *dim*.

See `default_transformer_poswise_net_hparams()` for details.

"multihead_attention": dict, Hyperparameters for the multihead attention strategy. Make sure the *output_dim* in this module is equal to *dim*.

See :func: *~texar.modules.encoder.MultiheadAttentionEncoder. default_harams* for details.

‘

"initializer" [dict, optional] Hyperparameters of the default initializer that initializes variables created in this module. See *get_initializer()* for details.

"embedding_tie" [bool] Whether to use the word embedding matrix as the output layer that computes logits. If *False*, an additional dense layer is created.

"output_layer_bias" [bool] Whether to use bias to the output layer.

"max_decoding_length" [int] The maximum allowed number of decoding steps. Set to a very large number of avoid the length constraint. Ignored if provided in *_build()* or "train_greedy" decoding is used.

Length penalty coefficient. Refer to https://arxiv.org/abs/1609.08144 for more details.

"name" [str] Name of the module.

TransformerDecoderOutput

class texar.modules.**TransformerDecoderOutput**

The output of *TransformerDecoder*.

logits

A float Tensor of shape *[batch_size, max_time, vocab_size]* containing the logits.

sample_id

An int Tensor of shape *[batch_size, max_time]* containing the sampled token indexes.

SoftmaxEmbeddingHelper

class texar.modules.**SoftmaxEmbeddingHelper**(*embedding*, *start_tokens*, *end_token*, *tau*, *stop_gradient=False*, *use_finish=True*)

A helper that feeds softmax probabilities over vocabulary to the next step. Uses the softmax probability vector to pass through word embeddings to get the next input (i.e., a mixed word embedding).

A subclass of Helper. Used as a helper to *RNNDecoderBase* _build() in inference mode.

Parameters

- **embedding** – An embedding argument (params) for tf.nn.embedding_lookup, or an instance of subclass of *texar.modules.EmbedderBase*. Note that other callables are not acceptable here.
- **start_tokens** – An int tensor shaped *[batch_size]*. The start tokens.
- **end_token** – An int scalar tensor. The token that marks end of decoding.
- **tau** – A float scalar tensor, the softmax temperature.
- **stop_gradient** (*bool*) – Whether to stop the gradient backpropagation when feeding softmax vector to the next step.
- **use_finish** (*bool*) – Whether to stop decoding once *end_token* is generated. If *False*, decoding will continue until *max_decoding_length* of the decoder is reached.

batch_size

Batch size of tensor returned by *sample*.

Returns a scalar int32 tensor.

sample_ids_dtype

DType of tensor returned by *sample*.

Returns a DType.

sample_ids_shape
Shape of tensor returned by *sample*, excluding the batch dimension.

Returns a *TensorShape*.

initialize(*name=None*)
Returns *(initial_finished, initial_inputs)*.

sample(*time*, *outputs*, *state*, *name=None*)
Returns *sample_id* which is softmax distributions over vocabulary with temperature *tau*. Shape = *[batch_size, vocab_size]*

next_inputs(*time*, *outputs*, *state*, *sample_ids*, *name=None*)
Returns *(finished, next_inputs, next_state)*.

GumbelSoftmaxEmbeddingHelper

class texar.modules.**GumbelSoftmaxEmbeddingHelper**(*embedding*, *start_tokens*, *end_token*, *tau*, *straight_through=False*, *stop_gradient=False*, *use_finish=True*)

A helper that feeds gumbel softmax sample to the next step. Uses the gumbel softmax vector to pass through word embeddings to get the next input (i.e., a mixed word embedding).

A subclass of Helper. Used as a helper to *RNNDecoderBase* `_build()` in inference mode.

Same as *SoftmaxEmbeddingHelper* except that here gumbel softmax (instead of softmax) is used.

Parameters

- **embedding** – An embedding argument (`params`) for tf.nn.embedding_lookup, or an instance of subclass of *texar.modules.EmbedderBase*. Note that other callables are not acceptable here.
- **start_tokens** – An int tensor shaped *[batch_size]*. The start tokens.
- **end_token** – An int scalar tensor. The token that marks end of decoding.
- **tau** – A float scalar tensor, the softmax temperature.
- **straight_through** (*bool*) – Whether to use straight through gradient between time steps. If *True*, a single token with highest probability (i.e., greedy sample) is fed to the next step and gradient is computed using straight through. If *False* (default), the soft gumbel-softmax distribution is fed to the next step.
- **stop_gradient** (*bool*) – Whether to stop the gradient backpropagation when feeding softmax vector to the next step.
- **use_finish** (*bool*) – Whether to stop decoding once *end_token* is generated. If *False*, decoding will continue until *max_decoding_length* of the decoder is reached.

sample(*time*, *outputs*, *state*, *name=None*)
Returns *sample_id* of shape *[batch_size, vocab_size]*. If *straight_through* is False, this is gumbel softmax distributions over vocabulary with temperature *tau*. If *straight_through* is True, this is one-hot vectors of the greedy samples.

get_helper

texar.modules.**get_helper**(*helper_type*, *inputs=None*, *sequence_length=None*, *embedding=None*, *start_tokens=None*, *end_token=None*, ***kwargs*)

Creates a Helper instance.

Parameters

- **helper_type** – A Helper class, its name or module path, or a class instance. If a class instance is given, it is returned directly.
- **inputs** (*optional*) – Inputs to the RNN decoder, e.g., ground truth tokens for teacher forcing decoding.
- **sequence_length** (*optional*) – A 1D int Tensor containing the sequence length of `inputs`.
- **embedding** (*optional*) – A callable that takes a vector tensor of indexes (e.g., an instance of subclass of *EmbedderBase*), or the *params* argument for *embedding_lookup* (e.g., the embedding Tensor).
- **start_tokens** (*optional*) – A int Tensor of shape *[batch_size]*, the start tokens.
- **end_token** (*optional*) – A int 0D Tensor, the token that marks end of decoding.
- ****kwargs** – Additional keyword arguments for constructing the helper.

Returns A helper instance.

3.4.5 Connectors

ConnectorBase

class texar.modules.**ConnectorBase**(*output_size*, *hparams=None*)

Base class inherited by all connector classes. A connector is to transform inputs into outputs with any specified structure and shape. For example, tranforming the final state of an encoder to the initial state of a decoder, and performing stochastic sampling in between as in Variational Autoencoders (VAEs).

Parameters

- **output_size** – Size of output **excluding** the batch dimension. For example, set *output_size* to *dim* to generate output of shape *[batch_size, dim]*. Can be an *int*, a tuple of *int*, a Tensorshape, or a tuple of TensorShapes. For example, to transform inputs to have decoder state size, set *output_size=decoder.state_size*.
- **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

output_size

The output size.

hparams

An *HParams* instance. The hyperparameters of the module.

name

The uniquified name of the module.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

ConstantConnector

class texar.modules.**ConstantConnector**(*output_size*, *hparams=None*)
Creates a constant Tensor or (nested) tuple of Tensors that contains a constant value.

Parameters

- **output_size** – Size of output **excluding** the batch dimension. For example, set *output_size* to *dim* to generate output of shape *[batch_size, dim]*. Can be an *int*, a tuple of *int*, a Tensorshape, or a tuple of TensorShapes. For example, to transform inputs to have decoder state size, set *output_size=decoder.state_size*.
- **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

This connector does not have trainable parameters. See *_build()* for the inputs and outputs of the connector.

Example

```
connector = Connector(cell.state_size)
zero_state = connector(batch_size=64, value=0.)
one_state = connector(batch_size=64, value=1.)
```

_build(*batch_size*, *value=None*)
Creates output tensor(s) that has the given value.

Parameters

- **batch_size** – An *int* or *int* scalar Tensor, the batch size.
- **value** (*optional*) – A scalar, the value that the output tensor(s) has. If *None*, "value" in *hparams* is used.

Returns A (structure of) tensor whose structure is the same as *output_size*, with value speicified by *value* or *hparams*.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    "value": 0.,
    "name": "constant_connector"
}
```

Here:

"value" [float] The constant scalar that the output tensor(s) has. Ignored if *value* is given to *_build()*.

"name" [str] Name of the connector.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

output_size
The output size.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

ForwardConnector

class texar.modules.**ForwardConnector**(*output_size*, *hparams=None*)
Transforms inputs to have specified structure.

Parameters

- **output_size** – Size of output **excluding** the batch dimension. For example, set *output_size* to *dim* to generate output of shape *[batch_size, dim]*. Can be an *int*, a tuple of *int*, a Tensorshape, or a tuple of TensorShapes. For example, to transform inputs to have decoder state size, set *output_size=decoder.state_size*.
- **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

This connector does not have trainable parameters. See *_build()* for the inputs and outputs of the connector.

The input to the connector must have the same structure with *output_size*, or must have the same number of elements and be re-packable into the structure of *output_size*. Note that if input is or contains a *dict* instance, the keys will be sorted to pack in deterministic order (See pack_sequence_as for more details).

Example

```
cell = LSTMCell(num_units=256)
# cell.state_size == LSTMStateTuple(c=256, h=256)

connector = ForwardConnector(cell.state_size)
output = connector([tensor_1, tensor_2])
# output == LSTMStateTuple(c=tensor_1, h=tensor_2)
```

_build(*inputs*)
Transforms inputs to have the same structure as with *output_size*. Values of the inputs are not changed.

`inputs` must either have the same structure, or have the same number of elements with *output_size*.

Parameters **inputs** – The input (structure of) tensor to pass forward.

Returns A (structure of) tensors that re-packs *inputs* to have the specified structure of *output_size*.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    "name": "forward_connector"
}
```

Here:

"name" [str] Name of the connector.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

output_size
The output size.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

MLPTransformConnector

class texar.modules.**MLPTransformConnector**(*output_size*, *hparams=None*)
Transforms inputs with an MLP layer and packs the results into the specified structure and size.

Parameters

- **output_size** – Size of output **excluding** the batch dimension. For example, set *output_size* to *dim* to generate output of shape *[batch_size, dim]*. Can be an *int*, a tuple of *int*, a Tensorshape, or a tuple of TensorShapes. For example, to transform inputs to have decoder state size, set *output_size=decoder.state_size*.
- **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

See *_build()* for the inputs and outputs of the connector.

The input to the connector can have arbitrary structure and size.

Example

```
cell = LSTMCell(num_units=256)
# cell.state_size == LSTMStateTuple(c=256, h=256)

connector = MLPTransformConnector(cell.state_size)
inputs = tf.zeros([64, 10])
output = connector(inputs)
# output == LSTMStateTuple(c=tensor_of_shape_(64, 256),
#                          h=tensor_of_shape_(64, 256))
```

```
## Use to connect encoder and decoder with different state size
encoder = UnidirectionalRNNEncoder(...)
_, final_state = encoder(inputs=...)

decoder = BasicRNNDecoder(...)
connector = MLPTransformConnector(decoder.state_size)

_ = decoder(
```

(continues on next page)

(continued from previous page)

```
    initial_state=connector(final_state),
    ...)
```

_build(*inputs*)

Transforms inputs with an MLP layer and packs the results to have the same structure as specified by *output_size*.

Parameters **inputs** – Input (structure of) tensors to be transformed. Must be a Tensor of shape *[batch_size, ...]* or a (nested) tuple of such Tensors. That is, the first dimension of (each) tensor must be the batch dimension.

Returns A Tensor or a (nested) tuple of Tensors of the same structure of *output_size*.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "activation_fn": "identity",
    "name": "mlp_connector"
}
```

Here:

"activation_fn" [str or callable] The activation function applied to the outputs of the MLP transformation layer. Can be a function, or its name or module path.

"name" [str] Name of the connector.

hparams

An *HParams* instance. The hyperparameters of the module.

name

The uniquified name of the module.

output_size

The output size.

trainable_variables

The list of trainable variables of the module.

variable_scope

The variable scope of the module.

ReparameterizedStochasticConnector

class texar.modules.**ReparameterizedStochasticConnector**(*output_size*, *hparams=None*)

Samples from a distribution with reparameterization trick, and transforms samples into specified size.

Reparameterization allows gradients to be back-propagated through the stochastic samples. Used in, e.g., Variational Autoencoders (VAEs).

Parameters

- **output_size** – Size of output **excluding** the batch dimension. For example, set *output_size* to *dim* to generate output of shape *[batch_size, dim]*. Can be an *int*, a tuple of *int*, a Tensorshape, or a tuple of TensorShapes. For example, to transform inputs to have decoder state size, set *output_size=decoder.state_size*.

- **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

Example

```
cell = LSTMCell(num_units=256)
# cell.state_size == LSTMStateTuple(c=256, h=256)

connector = ReparameterizedStochasticConnector(cell.state_size)

kwargs = {
    'loc': tf.zeros([batch_size, 10]),
    'scale_diag': tf.ones([batch_size, 10])
}
output, sample = connector(distribution_kwargs=kwargs)
# output == LSTMStateTuple(c=tensor_of_shape_(batch_size, 256),
#                          h=tensor_of_shape_(batch_size, 256))
# sample == Tensor([batch_size, 10])


kwargs = {
    'loc': tf.zeros([10]),
    'scale_diag': tf.ones([10])
}
output_, sample_ = connector(distribution_kwargs=kwargs,
                             num_samples=batch_size_)
# output_ == LSTMStateTuple(c=tensor_of_shape_(batch_size_, 256),
#                           h=tensor_of_shape_(batch_size_, 256))
# sample == Tensor([batch_size_, 10])
```

_build(*distribution='MultivariateNormalDiag'*, *distribution_kwargs=None*, *transform=True*, *num_samples=None*)

Samples from a distribution and optionally performs transformation with an MLP layer.

The distribution must be reparameterizable, i.e., *distribution.reparameterization_type* = *FULLY_REPARAMETERIZED*.

Parameters

- **distribution** – A instance of subclass of TF Distribution, or tensorflow_probability Distribution, Can be a class, its name or module path, or a class instance.
- **distribution_kwargs** (*dict, optional*) – Keyword arguments for the distribution constructor. Ignored if *distribution* is a class instance.
- **transform** (*bool*) – Whether to perform MLP transformation of the distribution samples. If *False*, the structure/shape of a sample must match *output_size*.
- **num_samples** (*optional*) – An *int* or *int* Tensor. Number of samples to generate. If not given, generate a single sample. Note that if batch size has already been included in *distribution*'s dimensionality, *num_samples* should be left as *None*.

Returns

A tuple (output, sample), where

- output: A Tensor or a (nested) tuple of Tensors with the same structure and size of

output_size. The batch dimension equals num_samples if specified, or is determined by the distribution dimensionality.

- sample: The sample from the distribution, prior to transformation.

Raises

- **ValueError** – If distribution cannot be reparametrized.
- **ValueError** – The output does not match *output_size*.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "activation_fn": "identity",
    "name": "reparameterized_stochastic_connector"
}
```

Here:

"activation_fn" [str] The activation function applied to the outputs of the MLP transformation layer. Can be a function, or its name or module path.

"name" [str] Name of the connector.

hparams

An *HParams* instance. The hyperparameters of the module.

name

The uniquified name of the module.

output_size

The output size.

trainable_variables

The list of trainable variables of the module.

variable_scope

The variable scope of the module.

StochasticConnector

class texar.modules.**StochasticConnector**(*output_size*, *hparams=None*)

Samples from a distribution and transforms samples into specified size.

The connector is the same as *ReparameterizedStochasticConnector*, except that here reparameterization is disabled, and thus the gradients cannot be back-propagated through the stochastic samples.

Parameters

- **output_size** – Size of output **excluding** the batch dimension. For example, set *output_size* to *dim* to generate output of shape *[batch_size, dim]*. Can be an *int*, a tuple of *int*, a Tensorshape, or a tuple of TensorShapes. For example, to transform inputs to have decoder state size, set *output_size=decoder.state_size*.
- **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

_build(*distribution='MultivariateNormalDiag'*, *distribution_kwargs=None*, *transform=False*, *num_samples=None*)

Samples from a distribution and optionally performs transformation with an MLP layer.

The inputs and outputs are the same as *ReparameterizedStochasticConnector* except that the distribution does not need to be reparameterizable, and gradient cannot be back-propagate through the samples.

Parameters

- **distribution** – A instance of subclass of TF Distribution, or tensorflow_probability Distribution. Can be a class, its name or module path, or a class instance.
- **distribution_kwargs** (*dict, optional*) – Keyword arguments for the distribution constructor. Ignored if *distribution* is a class instance.
- **transform** (*bool*) – Whether to perform MLP transformation of the distribution samples. If *False*, the structure/shape of a sample must match *output_size*.
- **num_samples** (*optional*) – An *int* or *int* Tensor. Number of samples to generate. If not given, generate a single sample. Note that if batch size has already been included in *distribution*'s dimensionality, *num_samples* should be left as *None*.

Returns

A tuple (output, sample), where

- output: A Tensor or a (nested) tuple of Tensors with the same structure and size of *output_size*. The batch dimension equals `num_samples` if specified, or is determined by the distribution dimensionality.
- sample: The sample from the distribution, prior to transformation.

Raises **ValueError** – The output does not match *output_size*.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "activation_fn": "identity",
    "name": "stochastic_connector"
}
```

Here:

"activation_fn" [str] The activation function applied to the outputs of the MLP transformation layer. Can be a function, or its name or module path.

"name" [str] Name of the connector.

hparams

An *HParams* instance. The hyperparameters of the module.

name

The uniquified name of the module.

output_size

The output size.

trainable_variables

The list of trainable variables of the module.

variable_scope

The variable scope of the module.

3.4.6 Classifiers

Conv1DClassifier

class texar.modules.**Conv1DClassifier**(*hparams=None*)

Simple Conv-1D classifier. This is a combination of the *Conv1DEncoder* with a classification layer.

Parameters **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

Example

```
clas = Conv1DClassifier(hparams={'num_classes': 10})

inputs = tf.random_uniform([64, 20, 256])
logits, pred = clas(inputs)
# logits == Tensor of shape [64, 10]
# pred   == Tensor of shape [64]
```

_build(*inputs*, *sequence_length=None*, *dtype=None*, *mode=None*)

Feeds the inputs through the network and makes classification.

The arguments are the same as in *Conv1DEncoder*.

The predictions of binary classification ("num_classes"=1) and multi-way classification ("num_classes">1) are different, as explained below.

Parameters

- **inputs** – The inputs to the network, which is a 3D tensor. See *Conv1DEncoder* for more details.
- **sequence_length** (*optional*) – An int tensor of shape *[batch_size]* containing the length of each element in `inputs`. If given, time steps beyond the length will first be masked out before feeding to the layers.
- **dtype** (*optional*) – Type of the inputs. If not provided, infers from inputs automatically.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. If *None*, *texar.global_mode()* is used.

Returns

A tuple *(logits, pred)*, where

- **'logits'** is a Tensor of shape *[batch_size, num_classes]* for *num_classes* >1, and *[batch_size]* for *num_classes* =1 (i.e., binary classification).
- **'pred'** is the prediction, a Tensor of shape *[batch_size]* and type *tf.int64*. For binary classification, the standard sigmoid function is used for prediction, and the class labels are *{0, 1}*.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    # (1) Same hyperparameters as in Conv1DEncoder
    ...

    # (2) Additional hyperparameters
    "num_classes": 2,
    "logit_layer_kwargs": {
        "use_bias": False
    },
    "name": "conv1d_classifier"
}
```

Here:

1. Same hyperparameters as in *Conv1DEncoder*. See the *default_hparams()*. An instance of Conv1DEncoder is created for feature extraction.

2. Additional hyperparameters:

"num_classes" [int] Number of classes:

- If **'> 0'**, an additional Dense layer is appended to the encoder to compute the logits over classes.
- If **'<= 0'**, no dense layer is appended. The number of classes is assumed to be the final dense layer size of the encoder.

"logit_layer_kwargs" [dict] Keyword arguments for the logit Dense layer constructor, except for argument "units" which is set to "num_classes". Ignored if no extra logit layer is appended.

"name" [str] Name of the classifier.

trainable_variables
The list of trainable variables of the module.

num_classes
The number of classes.

nn
The classifier neural network.

has_layer(*layer_name*)
Returns *True* if the network with the name exists. Returns *False* otherwise.

Parameters **layer_name** (*str*) – Name of the layer.

layer_by_name(*layer_name*)
Returns the layer with the name. Returns 'None' if the layer name does not exist.

Parameters **layer_name** (*str*) – Name of the layer.

layers_by_name
A dictionary mapping layer names to the layers.

layers
A list of the layers.

layer_names
A list of uniquified layer names.

layer_outputs_by_name(*layer_name*)
Returns the output tensors of the layer with the specified name. Returns *None* if the layer name does not exist.

Parameters **layer_name** (*str*) – Name of the layer.

layer_outputs
A list containing output tensors of each layer.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

variable_scope
The variable scope of the module.

UnidirectionalRNNClassifier

class texar.modules.**UnidirectionalRNNClassifier**(*cell=None*, *cell_dropout_mode=None*, *output_layer=None*, *hparams=None*)
One directional RNN classifier. This is a combination of the *UnidirectionalRNNEncoder* with a classification layer. Both step-wise classification and sequence-level classification are supported, specified in *hparams*.

Arguments are the same as in *UnidirectionalRNNEncoder*.

Parameters

- **cell** – (RNNCell, optional) If not specified, a cell is created as specified in `hparams["rnn_cell"]`.
- **cell_dropout_mode** (*optional*) – A Tensor taking value of tf.estimator.ModeKeys, which toggles dropout in the RNN cell (e.g., activates dropout in TRAIN mode). If *None*, *global_mode()* is used. Ignored if `cell` is given.
- **output_layer** (*optional*) – An instance of tf.layers.Layer. Applies to the RNN cell output of each step. If *None* (default), the output layer is created as specified in `hparams["output_layer"]`.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

_build(*inputs*, *sequence_length=None*, *initial_state=None*, *time_major=False*, *mode=None*, ***kwargs*)
Feeds the inputs through the network and makes classification.

The arguments are the same as in *UnidirectionalRNNEncoder*.

Parameters

- **inputs** – A 3D Tensor of shape *[batch_size, max_time, dim]*. The first two dimensions *batch_size* and *max_time* may be exchanged if *time_major=True* is specified.
- **sequence_length** (*optional*) – A 1D int tensor of shape *[batch_size]*. Sequence lengths of the batch inputs. Used to copy-through state and zero-out outputs when past a batch element's sequence length.
- **initial_state** (*optional*) – Initial state of the RNN.

- **time_major** (*bool*) – The shape format of the `inputs` and `outputs` Tensors. If *True*, these tensors are of shape *[max_time, batch_size, depth]*. If *False* (default), these tensors are of shape *[batch_size, max_time, depth]*.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. Controls output layer dropout if the output layer is specified with *hparams*. If *None* (default), *texar.global_mode()* is used.
- **return_cell_output** (*bool*) – Whether to return the output of the RNN cell. This is the results prior to the output layer.
- ****kwargs** – Optional keyword arguments of tf.nn.dynamic_rnn, such as *swap_memory*, *dtype*, *parallel_iterations*, etc.

Returns

A tuple *(logits, pred)*, containing the logits over classes and the predictions, respectively.

- If "clas_strategy"=="final_time" or "all_time"
  - If "num_classes"==1, *logits* and *pred* are of both shape *[batch_size]*
  - If "num_classes">1, *logits* is of shape *[batch_size, num_classes]* and *pred* is of shape *[batch_size]*.
- If "clas_strategy"=="time_wise",
  - If "num_classes"==1, *logits* and *pred* are of both shape *[batch_size, max_time]*
  - If "num_classes">1, *logits* is of shape *[batch_size, max_time, num_classes]* and *pred* is of shape *[batch_size, max_time]*.
  - If *time_major* is *True*, the batch and time dimensions are exchanged.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    # (1) Same hyperparameters as in UnidirectionalRNNEncoder
    ...

    # (2) Additional hyperparameters
    "num_classes": 2,
    "logit_layer_kwargs": None,
    "clas_strategy": "final_time",
    "max_seq_length": None,
    "name": "unidirectional_rnn_classifier"
}
```

Here:

1. Same hyperparameters as in *UnidirectionalRNNEncoder*. See the *default_hparams()*. An instance of UnidirectionalRNNEncoder is created for feature extraction.

2. Additional hyperparameters:

"num_classes" [int] Number of classes:

- If **'> 0'**, an additional Dense layer is appended to the encoder to compute the logits over classes.
- If **'<= 0'**, no dense layer is appended. The number of classes is assumed to be the final dense layer size of the encoder.

"logit_layer_kwargs" [dict] Keyword arguments for the logit Dense layer constructor, except for argument "units" which is set to "num_classes". Ignored if no extra logit layer is appended.

"clas_strategy" [str] The classification strategy, one of:

- **"final_time"**: Sequence-leve classification based on the output of the final time step. One sequence has one class.
- **"all_time"**: Sequence-level classification based on the output of all time steps. One sequence has one class.
- **"time_wise"**: Step-wise classfication, i.e., make classification for each time step based on its output.

"max_seq_length" [int, optional] Maximum possible length of input sequences. Required if "clas_strategy" is "all_time".

"name" [str] Name of the classifier.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

num_classes
The number of classes, specified in *hparams*.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

3.4.7 Networks

FeedForwardNetworkBase

class texar.modules.**FeedForwardNetworkBase**(*hparams=None*)
Base class inherited by all feed-forward network classes.

Parameters **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

See _build() for the inputs and outputs.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    "name": "NN"
}
```

append_layer(*layer*)
Appends a layer to the end of the network. The method is only feasible before _build is called.

Parameters **layer** – A tf.layers.Layer instance, or a dict of layer hyperparameters.

has_layer(*layer_name*)
Returns *True* if the network with the name exists. Returns *False* otherwise.

Parameters **layer_name** (*str*) – Name of the layer.

layer_by_name(*layer_name*)
Returns the layer with the name. Returns 'None' if the layer name does not exist.

Parameters **layer_name** (*str*) – Name of the layer.

layers_by_name
A dictionary mapping layer names to the layers.

layers
A list of the layers.

layer_names
A list of uniquified layer names.

layer_outputs_by_name(*layer_name*)
Returns the output tensors of the layer with the specified name. Returns *None* if the layer name does not exist.

Parameters **layer_name** (*str*) – Name of the layer.

layer_outputs
A list containing output tensors of each layer.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

FeedForwardNetwork

class texar.modules.**FeedForwardNetwork**(*layers=None*, *hparams=None*)
Feed-forward neural network that consists of a sequence of layers.

Parameters

- **layers** (*list, optional*) – A list of Layer instances composing the network. If not given, layers are created according to *hparams*.
- **hparams** (*dict, optional*) – Embedder hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

See *_build()* of *FeedForwardNetworkBase* for the inputs and outputs.

Example

```
hparams = { # Builds a two-layer dense NN
    "layers": [
        { "type": "Dense", "kwargs": { "units": 256 },
        { "type": "Dense", "kwargs": { "units": 10 }
    ]
}
nn = FeedForwardNetwork(hparams=hparams)

inputs = tf.random_uniform([64, 100])
outputs = nn(inputs)
# outputs == Tensor of shape [64, 10]
```

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    "layers": [],
    "name": "NN"
}
```

Here:

"layers" [list] A list of layer hyperparameters. See *get_layer()* for the details of layer hyperparameters.

"name" [str] Name of the network.

append_layer(*layer*)
Appends a layer to the end of the network. The method is only feasible before _build is called.

Parameters **layer** – A tf.layers.Layer instance, or a dict of layer hyperparameters.

has_layer(*layer_name*)
Returns *True* if the network with the name exists. Returns *False* otherwise.

Parameters **layer_name** (*str*) – Name of the layer.

hparams
An *HParams* instance. The hyperparameters of the module.

layer_by_name(*layer_name*)
Returns the layer with the name. Returns 'None' if the layer name does not exist.

Parameters **layer_name** (*str*) – Name of the layer.

layer_names
A list of uniquified layer names.

layer_outputs
A list containing output tensors of each layer.

layer_outputs_by_name(*layer_name*)
Returns the output tensors of the layer with the specified name. Returns *None* if the layer name does not exist.

Parameters **layer_name** (*str*) – Name of the layer.

layers
A list of the layers.

layers_by_name
A dictionary mapping layer names to the layers.

name
The uniquified name of the module.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

Conv1DNetwork

class texar.modules.**Conv1DNetwork**(*hparams=None*)
Simple Conv-1D network which consists of a sequence of conv layers followed with a sequence of dense layers.

Parameters **hparams** (*dict, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

See *_build()* for the inputs and outputs. The inputs must be a 3D Tensor of shape *[batch_size, length, channels]* (default), or *[batch_size, channels, length]* (if *data_format* is set to *'channels_last'* through *hparams*). For example, for sequence classification, *length* corresponds to time steps, and *channels* corresponds to embedding dim.

Example

```
nn = Conv1DNetwork() # Use the default structure

inputs = tf.random_uniform([64, 20, 256])
outputs = nn(inputs)
# outputs == Tensor of shape [64, 128], cuz the final dense layer
# has size 128.
```

_build(*inputs*, *sequence_length=None*, *dtype=None*, *mode=None*)
Feeds forward inputs through the network layers and returns outputs.

Parameters

- **inputs** – The inputs to the network, which is a 3D tensor.
- **sequence_length** (*optional*) – An int tensor of shape *[batch_size]* containing the length of each element in `inputs`. If given, time steps beyond the length will first be masked out before feeding to the layers.
- **dtype** (*optional*) – Type of the inputs. If not provided, infers from inputs automatically.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. If *None*, *texar.global_mode()* is used.

Returns The output of the final layer.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    # (1) Conv layers
    "num_conv_layers": 1,
    "filters": 128,
```

(continues on next page)

(continued from previous page)

```
    "kernel_size": [3, 4, 5],
    "conv_activation": "relu",
    "conv_activation_kwargs": None,
    "other_conv_kwargs": None,
    # (2) Pooling layers
    "pooling": "MaxPooling1D",
    "pool_size": None,
    "pool_strides": 1,
    "other_pool_kwargs": None,
    # (3) Dense layers
    "num_dense_layers": 1,
    "dense_size": 128,
    "dense_activation": "identity",
    "dense_activation_kwargs": None,
    "final_dense_activation": None,
    "final_dense_activation_kwargs": None,
    "other_dense_kwargs": None,
    # (4) Dropout
    "dropout_conv": [1],
    "dropout_dense": [],
    "dropout_rate": 0.75,
    # (5) Others
    "name": "conv1d_network",
}
```

Here:

1. For **convolutional** layers:

   "num_conv_layers" [int] Number of convolutional layers.

   "filters" [int or list] The number of filters in the convolution, i.e., the dimensionality of the output space. If "num_conv_layers" > 1, "filters" must be a list of "num_conv_layers" integers.

   "kernel_size" [int or list] Lengths of 1D convolution windows.

   - If "num_conv_layers" == 1, this can be a list of arbitrary number of *int* denoting different sized conv windows. The number of filters of each size is specified by "filters". For example, the default values will create 3 sets of filters, each of which has kernel size of 3, 4, and 5, respectively, and has filter number 128.
   - If "num_conv_layers" > 1, this must be a list of length "num_conv_layers". Each element can be an *int* or a list of arbitrary number of *int* denoting the kernel size of respective layer.

   "conv_activation": str or callable Activation function applied to the output of the convolutional layers. Set to "indentity" to maintain a linear activation. See *get_activation_fn()* for more details.

   "conv_activation_kwargs" [dict, optional] Keyword arguments for conv layer activation functions. See *get_activation_fn()* for more details.

   "other_conv_kwargs" [dict, optional] Other keyword arguments for tf.layers.Conv1D constructor, e.g., "data_format", "padding", etc.

2. For **pooling** layers:

   "pooling" [str or class or instance] Pooling layer after each of the convolutional layer(s). Can a pooling layer class, its name or module path, or a class instance.

"pool_size" [int or list, optional] Size of the pooling window. If an *int*, all pooling layer will have the same pool size. If a list, the list length must equal "num_conv_layers". If *None* and the pooling type is either MaxPooling or AveragePooling, the pool size will be set to input size. That is, the output of the pooling layer is a single unit.

"pool_strides" [int or list, optional] Strides of the pooling operation. If an *int*, all pooling layer will have the same stride. If a list, the list length must equal "num_conv_layers".

"other_pool_kwargs" [dict, optional] Other keyword arguments for pooling layer class constructor.

3. For **dense** layers (note that here dense layers always follow conv and pooling layers):

   "num_dense_layers" [int] Number of dense layers.

   "dense_size" [int or list] Number of units of each dense layers. If an *int*, all dense layers will have the same size. If a list of *int*, the list length must equal "num_dense_layers".

   "dense_activation" [str or callable] Activation function applied to the output of the dense layers **except** the last dense layer output . Set to "indentity" to maintain a linear activation. See *get_activation_fn()* for more details.

   "dense_activation_kwargs" [dict, optional] Keyword arguments for dense layer activation functions before the last dense layer. See *get_activation_fn()* for more details.

   "final_dense_activation" [str or callable] Activation function applied to the output of the **last** dense layer. Set to *None* or "indentity" to maintain a linear activation. See *get_activation_fn()* for more details.

   "final_dense_activation_kwargs" [dict, optional] Keyword arguments for the activation function of last dense layer. See *get_activation_fn()* for more details.

   "other_dense_kwargs" [dict, optional] Other keyword arguments for Dense layer class constructor.

4. For **dropouts**:

   "dropout_conv" [int or list] The indexes of conv layers (starting from *0*) whose **inputs** are applied with dropout. The index = `num_conv_layers` means dropout applies to the final conv layer output. E.g.,

   ```
   {
       "num_conv_layers": 2,
       "dropout_conv": [0, 2]
   }
   ```

   will leads to a series of layers as *-dropout-conv0-conv1-dropout-*.

   The dropout mode (training or not) is controlled by the `mode` argument of *_build()*.

   "dropout_dense" [int or list] Same as "dropout_conv" but applied to dense layers (index starting from *0*).

   "dropout_rate" [float] The dropout rate, between 0 and 1. E.g., *"dropout_rate": 0.1* would drop out 10% of elements.

5. Others:

   "name" [str] Name of the network.

append_layer(*layer*)

Appends a layer to the end of the network. The method is only feasible before *_build* is called.

Parameters **layer** – A tf.layers.Layer instance, or a dict of layer hyperparameters.

has_layer(*layer_name*)
Returns *True* if the network with the name exists. Returns *False* otherwise.

Parameters **layer_name** (*str*) – Name of the layer.

hparams
An *HParams* instance. The hyperparameters of the module.

layer_by_name(*layer_name*)
Returns the layer with the name. Returns 'None' if the layer name does not exist.

Parameters **layer_name** (*str*) – Name of the layer.

layer_names
A list of uniquified layer names.

layer_outputs
A list containing output tensors of each layer.

layer_outputs_by_name(*layer_name*)
Returns the output tensors of the layer with the specified name. Returns *None* if the layer name does not exist.

Parameters **layer_name** (*str*) – Name of the layer.

layers
A list of the layers.

layers_by_name
A dictionary mapping layer names to the layers.

name
The uniquified name of the module.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

3.4.8 Memory

MemNetBase

class texar.modules.**MemNetBase**(*raw_memory_dim*, *input_embed_fn=None*, *output_embed_fn=None*, *query_embed_fn=None*, *hparams=None*)
Base class inherited by all memory network classes.

Parameters

- **raw_memory_dim** (*int*) – Dimension size of raw memory entries (before embedding). For example, if a raw memory entry is a word, this is the **vocabulary size** (imagine a one-hot representation of word). If a raw memory entry is a dense vector, this is the dimension size of the vector.
- **input_embed_fn** (*optional*) – A callable that embeds raw memory entries as inputs. This corresponds to the *A* embedding operation in (Sukhbaatar et al.) If not provided, a default embedding operation is created as specified in *hparams*. See *get_default_embed_fn()* for details.

- **output_embed_fn** (*optional*) – A callable that embeds raw memory entries as outputs. This corresponds to the *C* embedding operation in (Sukhbaatar et al.) If not provided, a default embedding operation is created as specified in *hparams*. See *get_default_embed_fn()* for details.
- **query_embed_fn** (*optional*) – A callable that embeds query. This corresponds to the *B* embedding operation in (Sukhbaatar et al.). If not provided and "use_B" is True in *hparams*, a default embedding operation is created as specified in *hparams*. See *get_default_embed_fn()* for details. Notice: If you'd like to customize this callable, please follow the same number and style of dimensions as in *input_embed_fn* or *output_embed_fn*, and assume that the 2nd dimension of its input and output (which corresponds to *memory_size*) is 1.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

get_default_embed_fn(*memory_size*, *embed_fn_hparams*)

Creates a default embedding function. Can be used for A, C, or B operation.

For B operation (i.e., query_embed_fn), *memory_size* must be 1.

The function is a combination of both memory embedding and temporal embedding, with the combination method specified by "combine_mode" in the *embed_fn_hparams*.

Parameters **embed_fn_hparams** (*dict or HParams*) – Hyperparameter of the embedding function. See `default_memnet_embed_fn()` for details.

Returns

A tuple *(embed_fn, memory_dim)*, where

- **'memory_dim'** is the dimension of memory entry embedding, inferred from `embed_fn_hparams`.
  - If *combine_mode* == 'add', *memory_dim* is the embedder dimension.
  - If *combine_mode* == 'concat', *memory_dim* is the sum of the memory embedder dimension and the temporal embedder dimension.
- **'embed_fn'** is an embedding function that takes in memory and returns memory embedding. Specifically, the function has signature `memory_embedding=embed_fn(memory=None, soft_memory=None)` where one of *memory* and *soft_memory* is provided (but not both).

  param memory An *int* Tensor of shape *[batch_size, memory_size]* containing memory indexes used for embedding lookup.

  param soft_memory A Tensor of shape *[batch_size, memory_size, raw_memory_dim]* containing soft weights used to mix the embedding vectors.

  returns A Tensor of shape *[batch_size, memory_size, memory_dim]* containing the memory entry embeddings.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "n_hops": 1,
    "memory_dim": 100,
    "relu_dim": 50,
```

(continues on next page)

(continued from previous page)

```
    "memory_size": 100,
    "A": default_embed_fn_hparams,
    "C": default_embed_fn_hparams,
    "B": default_embed_fn_hparams,
    "use_B": False,
    "use_H": False,
    "dropout_rate": 0,
    "variational": False,
    "name": "memnet",
}
```

Here:

"n_hops" [int] Number of hops.

"memory_dim" [int] Memory dimension, i.e., the dimension size of a memory entry embedding. Ignored if at least one of the embedding functions is created according to *hparams*. In this case *memory_dim* is inferred from the created embed_fn.

"relu_dim" [int] Number of elements in *memory_dim* that have relu at the end of each hop. Should be not less than 0 and not more than :attr'memory_dim'.

"memory_size" [int] Number of entries in memory.

For example, the number of sentences {x_i} in Fig.1(a) of (Sukhbaatar et al.) End-To-End Memory Networks.

"use_B" [bool] Whether to create the query embedding function. Ignored if *query_embed_fn* is given to the constructor.

"use_H" [bool] Whether to perform a linear transformation with matrix *H* at the end of each A-C layer.

"dropout_rate" [float] The dropout rate to apply to the output of each hop. Should be between 0 and 1. E.g., *dropout_rate=0.1* would drop out 10% of the units.

"variational" [bool] Whether to share dropout masks after each hop.

memory_size
The memory size.

raw_memory_dim
The dimension of memory element (or vocabulary size).

memory_dim
The dimension of embedded memory and all vectors in hops.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

MemNetRNNLike

class texar.modules.**MemNetRNNLike**(*raw_memory_dim*, *input_embed_fn=None*, *output_embed_fn=None*, *query_embed_fn=None*, *hparams=None*)

An implementation of multi-layer end-to-end memory network, with RNN-like weight tying described in (Sukhbaatar et al.) End-To-End Memory Networks .

See *get_default_embed_fn()* for default embedding functions. Customized embedding functions must follow the same signature.

Parameters

- **raw_memory_dim** (*int*) – Dimension size of raw memory entries (before embedding). For example, if a raw memory entry is a word, this is the **vocabulary size** (imagine a one-hot representation of word). If a raw memory entry is a dense vector, this is the dimension size of the vector.
- **input_embed_fn** (*optional*) – A callable that embeds raw memory entries as inputs. This corresponds to the *A* embedding operation in (Sukhbaatar et al.) If not provided, a default embedding operation is created as specified in *hparams*. See *get_default_embed_fn()* for details.
- **output_embed_fn** (*optional*) – A callable that embeds raw memory entries as outputs. This corresponds to the *C* embedding operation in (Sukhbaatar et al.) If not provided, a default embedding operation is created as specified in *hparams*. See *get_default_embed_fn()* for details.
- **query_embed_fn** (*optional*) – A callable that embeds query. This corresponds to the *B* embedding operation in (Sukhbaatar et al.). If not provided and "use_B" is True in *hparams*, a default embedding operation is created as specified in *hparams*. See *get_default_embed_fn()* for details. For customized query_embed_fn, note that the function must follow the signature of the default embed_fn where *memory_size* must be 1.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    "n_hops": 1,
    "memory_dim": 100,
    "relu_dim": 50,
    "memory_size": 100,
    "A": default_embed_fn_hparams,
    "C": default_embed_fn_hparams,
    "B": default_embed_fn_hparams,
    "use_B": False,
    "use_H": True,
    "dropout_rate": 0,
    "variational": False,
    "name": "memnet_rnnlike",
}
```

Here:

"n_hops" [int] Number of hops.

"memory_dim" [int] Memory dimension, i.e., the dimension size of a memory entry embedding. Ignored if at least one of the embedding functions is created according to *hparams*. In this case *memory_dim* is inferred from the created embed_fn.

"relu_dim" [int] Number of elements in *memory_dim* that have relu at the end of each hop. Should be not less than 0 and not more than :attr'memory_dim'.

"memory_size" [int] Number of entries in memory.

For example, the number of sentences {x_i} in Fig.1(a) of (Sukhbaatar et al.) End-To-End Memory Networks.

"use_B" [bool] Whether to create the query embedding function. Ignored if *query_embed_fn* is given to the constructor.

"use_H" [bool] Whether to perform a linear transformation with matrix *H* at the end of each A-C layer.

"dropout_rate" [float] The dropout rate to apply to the output of each hop. Should be between 0 and 1. E.g., *dropout_rate=0.1* would drop out 10% of the units.

"variational" [bool] Whether to share dropout masks after each hop.

hparams
An *HParams* instance. The hyperparameters of the module.

memory_dim
The dimension of embedded memory and all vectors in hops.

memory_size
The memory size.

name
The uniquified name of the module.

raw_memory_dim
The dimension of memory element (or vocabulary size).

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

default_memnet_embed_fn_hparams

texar.modules.**default_memnet_embed_fn_hparams**()
Returns a dictionary of hyperparameters with default hparams for `default_embed_fn()`

```
{
    "embedding": {
        "dim": 100
    },
    "temporal_embedding": {
        "dim": 100
    },
    "combine_mode": "add"
}
```

Here:

"embedding" [dict, optional] Hyperparameters for embedding operations. See *default_hparams()* of *WordEmbedder* for details. If *None*, the default hyperparameters are used.

"temporal_embedding" [dict, optional] Hyperparameters for temporal embedding operations. See `default_hparams()` of `PositionEmbedder` for details. If *None*, the default hyperparameters are used.

"combine_mode" [str] Either **'add'** or **'concat'**. If 'add', memory embedding and temporal embedding are added up. In this case the two embedders must have the same dimension. If 'concat', the two embeddings are concated.

3.4.9 Policy

PolicyNetBase

class texar.modules.**PolicyNetBase**(*network=None*, *network_kwargs=None*, *hparams=None*)

Policy net that takes in states and outputs actions.

Parameters

- **network** (*optional*) – A network that takes in state and returns outputs for generating actions. For example, an instance of subclass of `FeedForwardNetworkBase`. If *None*, a network is created as specified in `hparams`.
- **network_kwargs** (*dict, optional*) – Keyword arguments for network constructor. Note that the *hparams* argument for network constructor is specified in the "network_hparams" field of `hparams` and should not be included in *network_kwargs*. Ignored if `network` is given.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See `default_hparams()` for the hyperparameter sturcture and default values.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    'network_type': 'FeedForwardNetwork',
    'network_hparams': {
        'layers': [
            {
                'type': 'Dense',
                'kwargs': {'units': 256, 'activation': 'relu'}
            },
            {
                'type': 'Dense',
                'kwargs': {'units': 256, 'activation': 'relu'}
            },
        ]
    },
    'distribution_kwargs': None,
    'name': 'policy_net',
}
```

Here:

"network_type" [str or class or instance] A network that takes in state and returns outputs for generating actions. This can be a class, its name or module path, or a class instance. Ignored if *network* is given to the constructor.

"network_hparams" [dict] Hyperparameters for the network. With the `network_kwargs` argument to the constructor, a network is created with `network_class(**network_kwargs, hparams=network_hparams)`.

For example, the default values creates a two-layer dense network.

"distribution_kwargs" [dict, optional] Keyword arguments for distribution constructor. A distribution would be created for action sampling.

"name" [str] Name of the policy.

network
The network.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

CategoricalPolicyNet

class texar.modules.**CategoricalPolicyNet**(*action_space=None*, *network=None*, *network_kwargs=None*, *hparams=None*)

Policy net with Categorical distribution for discrete scalar actions.

This is a combination of a network with a top-layer distribution for action sampling.

Parameters

- **action_space** (*optional*) – An instance of *Space* specifying the action space. If not given, an discrete action space *[0, high]* is created with *high* specified in *hparams*.
- **network** (*optional*) – A network that takes in state and returns outputs for generating actions. For example, an instance of subclass of *FeedForwardNetworkBase*. If *None*, a network is created as specified in *hparams*.
- **network_kwargs** (*dict, optional*) – Keyword arguments for network constructor. Note that the *hparams* argument for network constructor is specified in the "network_hparams" field of *hparams* and should not be included in *network_kwargs*. Ignored if *network* is given.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

_build(*inputs*, *mode=None*)

Takes in states and outputs actions.

Parameters

- **inputs** – Inputs to the policy network with the first dimension the batch dimension.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. If *None*, *texar.global_mode()* is used.

Returns A *dict* including fields *"logits"*, *"action"*, and *"dist"*, where

- **"logits"**: A Tensor of shape *[batch_size] + action_space size* used for categorical distribution sampling.
- **"action"**: A Tensor of shape *[batch_size] + action_space.shape*.
- **"dist"**: The Categorical based on the logits.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    'network_type': 'FeedForwardNetwork',
    'network_hparams': {
        'layers': [
            {
                'type': 'Dense',
                'kwargs': {'units': 256, 'activation': 'relu'}
            },
            {
                'type': 'Dense',
                'kwargs': {'units': 256, 'activation': 'relu'}
            },
        ]
    },
    'distribution_kwargs': {
        'dtype': 'int32',
        'validate_args': False,
        'allow_nan_stats': True
    },
    'action_space': 2,
    'make_output_layer': True,
    'name': 'categorical_policy_net'
}
```

Here:

"distribution_kwargs" [dict] Keyword arguments for the Categorical distribution constructor. Arguments *logits* and *probs* should not be included as they are inferred from the inputs. Argument *dtype* can be a string (e.g., *int32*) and will be converted to a corresponding tf dtype.

"action_space" [int] Upper bound of the action space. The resulting action space is all discrete scalar numbers between 0 and the upper bound specified here (both inclusive).

"make_output_layer" [bool] Whether to append a dense layer to the network to transform features to logits for action sampling. If *False*, the final layer output of network must match the action space.

See *default_hparams* for details of other hyperparameters.

action_space

An instance of *Space* specifiying the action space.

hparams

An *HParams* instance. The hyperparameters of the module.

name

The uniquified name of the module.

network

The network.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

3.4.10 Q-Nets

QNetBase

class texar.modules.**QNetBase**(*network=None*, *network_kwargs=None*, *hparams=None*)
Base class inheritted by all Q net classes. A Q net takes in states and outputs Q value of actions.

Parameters

- **network** (*optional*) – A network that takes in state and returns Q values. For example, an instance of subclass of *FeedForwardNetworkBase*. If *None*, a network is created as specified in *hparams*.
- **network_kwargs** (*dict, optional*) – Keyword arguments for network constructor. Note that the *hparams* argument for network constructor is specified in the "network_hparams" field of *hparams* and should not be included in *network_kwargs*. Ignored if *network* is given.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    'network_type': 'FeedForwardNetwork',
    'network_hparams': {
        'layers': [
            {
                'type': 'Dense',
                'kwargs': {'units': 256, 'activation': 'relu'}
            },
            {
                'type': 'Dense',
                'kwargs': {'units': 256, 'activation': 'relu'}
            },
        ]
    },
    'name': 'q_net',
}
```

Here:

"network_type" [str or class or instance] A network that takes in state and returns outputs for generating actions. This can be a class, its name or module path, or a class instance. Ignored if *network* is given to the constructor.

"network_hparams" [dict] Hyperparameters for the network. With the `network_kwargs` argument to the constructor, a network is created with `network_class(**network_kwargs, hparams=network_hparams)`.

For example, the default values creates a two-layer dense network.

"name" [str] Name of the Q net.

network
The network.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

CategoricalPolicyNet

class texar.modules.**CategoricalQNet**(*action_space=None*, *network=None*, *network_kwargs=None*, *hparams=None*)

Q net with categorical scalar action space.

Parameters

- **action_space** (*optional*) – An instance of *Space* specifying the action space. If not given, an discrete action space *[0, high]* is created with *high* specified in *hparams*.
- **network** (*optional*) – A network that takes in state and returns Q values. For example, an instance of subclass of *FeedForwardNetworkBase*. If *None*, a network is created as specified in *hparams*.
- **network_kwargs** (*dict, optional*) – Keyword arguments for network constructor. Note that the *hparams* argument for network constructor is specified in the "network_hparams" field of *hparams* and should not be included in *network_kwargs*. Ignored if *network* is given.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

_build(*inputs*, *mode=None*)

Takes in states and outputs Q values.

Parameters

- **inputs** – Inputs to the Q net with the first dimension the batch dimension.
- **mode** (*optional*) – A tensor taking value in tf.estimator.ModeKeys, including *TRAIN*, *EVAL*, and *PREDICT*. If *None*, *texar.global_mode()* is used.

Returns A *dict* including fields *"qvalues"*. where

- **"qvalues"**: A Tensor of shape *[batch_size] + action_space size* containing Q values of all possible actions.

static default_hparams()

Returns a dictionary of hyperparameters with default values.

```
{
    'network_type': 'FeedForwardNetwork',
    'network_hparams': {
        'layers': [
            {
                'type': 'Dense',
                'kwargs': {'units': 256, 'activation': 'relu'}
            },
            {
                'type': 'Dense',
                'kwargs': {'units': 256, 'activation': 'relu'}
            },
        ]
    },
    'action_space': 2,
    'make_output_layer': True,
    'name': 'q_net'
}
```

Here:

"action_space" [int] Upper bound of the action space. The resulting action space is all discrete scalar numbers between 0 and the upper bound specified here (both inclusive).

"make_output_layer" [bool] Whether to append a dense layer to the network to transform features to Q values. If *False*, the final layer output of network must match the action space.

See *default_hparams* for details of other hyperparameters.

action_space
An instance of *Space* specifiying the action space.

hparams
An *HParams* instance. The hyperparameters of the module.

name
The uniquified name of the module.

network
The network.

trainable_variables
The list of trainable variables of the module.

variable_scope
The variable scope of the module.

3.5 Agents

3.5.1 Sequence Agents

SeqPGAgent

class texar.agents.**SeqPGAgent**(*samples*, *logits*, *sequence_length*, *trainable_variables=None*, *learning_rate=None*, *sess=None*, *hparams=None*)

Policy Gradient agent for sequence prediction.

This is a wrapper of the **training process** that trains a model with policy gradient. Agent itself does not create new trainable variables.

Parameters

- **samples** – An *int* Tensor of shape *[batch_size, max_time]* containing sampled sequences from the model.
- **logits** – A float Tenosr of shape *[batch_size, max_time, vocab_size]* containing the logits of samples from the model.
- **sequence_length** – A Tensor of shape *[batch_size]*. Time steps beyond the respective sequence lengths are masked out.
- **trainable_variables** (*optional*) – Trainable variables of the model to update during training. If *None*, all trainable variables in the graph are used.
- **learning_rate** (*optional*) – Learning rate for policy optimization. If not given, determine the learning rate from *hparams*. See *get_train_op()* for more details.
- **sess** (*optional*) – A tf session. Can be *None* here and set later with *agent.sess = session*.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

Example

```
## Train a decoder with policy gradient
decoder = BasicRNNDecoder(...)
outputs, _, sequence_length = decoder(
    decoding_strategy='infer_sample', ...)

sess = tf.Session()
agent = SeqPGAgent(
    samples=outputs.sample_id,
    logits=outputs.logits,
    sequence_length=sequence_length,
    sess=sess)
while training:
    # Generate samples
    vals = agent.get_samples()
    # Evaluate reward
    sample_text = tx.utils.map_ids_to_strs(vals['samples'], vocab)
    reward_bleu = []
    for y, y_ in zip(ground_truth, sample_text)
        reward_bleu.append(tx.evals.sentence_bleu(y, y_)
    # Update
    agent.observe(reward=reward_bleu)
```

static default_hparams()

Returns a dictionary of hyperparameters with default values:

```
{
    'discount_factor': 0.95,
    'normalize_reward': False,
    'entropy_weight': 0.,
    'loss': {
```

(continues on next page)

(continued from previous page)

```
        'average_across_batch': True,
        'average_across_timesteps': False,
        'sum_over_batch': False,
        'sum_over_timesteps': True,
        'time_major': False
    },
    'optimization': default_optimization_hparams(),
    'name': 'pg_agent',
}
```

Here:

"discount_factor" [float] The discount factor of reward.

"normalize_reward" [bool] Whether to normalize the discounted reward, by *(discounted_reward - mean) / std*. Here *mean* and *std* are over all time steps and all samples in the batch.

"entropy_weight" [float] The weight of entropy loss of the sample distribution, to encourage maximizing the Shannon entropy. Set to 0 to disable the loss.

"loss" [dict] Extra keyword arguments for *pg_loss_with_logits()*, including the reduce arguments (e.g., *average_across_batch*) and *time_major*

"optimization" [dict] Hyperparameters of optimization for updating the policy net. See *default_optimization_hparams()* for details.

"name" [str] Name of the agent.

get_samples(*extra_fetches=None*, *feed_dict=None*)

Returns sequence samples and extra results.

Parameters

- **extra_fetches** (*dict, optional*) – Extra tensors to fetch values, besides *samples* and *sequence_length*. Same as the *fetches* argument of tf.Session.run and tf_main:*partial_run <Session#partial_run>*.
- **feed_dict** (*dict, optional*) – A *dict* that maps tensor to values. Note that all placeholder values used in *get_samples()* and subsequent *observe()* calls should be fed here.

Returns A *dict* with keys **"samples"** and **"sequence_length"** containing the fetched values of *samples* and *sequence_length*, as well as other fetched values as specified in `extra_fetches`.

Example

```
extra_fetches = {'truth_ids': data_batch['text_ids']}
vals = agent.get_samples()
sample_text = tx.utils.map_ids_to_strs(vals['samples'], vocab)
truth_text = tx.utils.map_ids_to_strs(vals['truth_ids'], vocab)
reward = reward_fn_in_python(truth_text, sample_text)
```

observe(*reward*, *train_policy=True*, *compute_loss=True*)

Observes the reward, and updates the policy or computes loss accordingly.

Parameters

- **reward** – A Python array/list of shape *[batch_size]* containing the reward for the samples generated in last call of *get_samples()*.
- **train_policy** (*bool*) – Whether to update the policy model according to the reward.
- **compute_loss** (*bool*) – If *train_policy* is False, whether to compute the policy gradient loss (but does not update the policy).

Returns If *train_policy* or *compute_loss* is True, returns the loss (a python float scalar). Otherwise returns *None*.

sess
The tf session.

pg_loss
The scalar tensor of policy gradient loss.

sequence_length
The tensor of sample sequence length, of shape *[batch_size]*.

samples
The tensor of sequence samples.

hparams
A `HParams` instance. The hyperparameters of the module.

logits
The tensor of sequence logits.

name
The name of the module (not uniquified).

variable_scope
The variable scope of the agent.

3.5.2 Episodic Agents

EpisodicAgentBase

class texar.agents.**EpisodicAgentBase**(*env_config*, *hparams=None*)
Base class inherited by episodic RL agents.

An agent is a wrapper of the **training process** that trains a model with RL algorithms. Agent itself does not create new trainable variables.

An episodic RL agent typically provides 3 interfaces, namely, *reset()*, *get_action()* and *observe()*, and is used as the following example.

Example

```
env = SomeEnvironment(...)
agent = PGAgent(...)

while True:
    # Starts one episode
    agent.reset()
    observ = env.reset()
    while True:
```

(continues on next page)

(continued from previous page)

```
        action = agent.get_action(observ)
        next_observ, reward, terminal = env.step(action)
        agent.observe(reward, terminal)
        observ = next_observ
        if terminal:
            break
```

Parameters

- **env_config** – An instance of *EnvConfig* specifying action space, observation space, and reward range, etc. Use `get_gym_env_config()` to create an EnvConfig from a gym environment.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    "name": "agent"
}
```

reset()
Resets the states to begin new episode.

observe(*reward*, *terminal*, *train_policy=True*, *feed_dict=None*)
Observes experience from environment.

Parameters

- **reward** – Reward of the action. The configuration (e.g., shape) of the reward is defined in *env_config*.
- **terminal** (*bool*) – Whether the episode is terminated.
- **train_policy** (*bool*) – Wether to update the policy for this step.
- **feed_dict** (*dict, optional*) – Any stuffs fed to running the training operator.

get_action(*observ*, *feed_dict=None*)
Gets action according to observation.

Parameters **observ** – Observation from the environment.

Returns action from the policy.

env_config
Environment configuration.

hparams
A `HParams` instance. The hyperparameters of the module.

name
The name of the module (not uniquified).

variable_scope
The variable scope of the agent.

PGAgent

class texar.agents.**PGAgent**(*env_config*, *sess=None*, *policy=None*, *policy_kwargs=None*, *policy_caller_kwargs=None*, *learning_rate=None*, *hparams=None*)

Policy gradient agent for episodic setting. This agent here supports **un-batched** training, i.e., each time generates one action, takes one observation, and updates the policy.

The policy must take in an observation of shape *[1] + observation_shape*, where the first dimension 1 stands for batch dimension, and output a *dict* containing:

- Key **"action"** whose value is a Tensor of shape *[1] + action_shape* containing a single action.
- One of keys "log_prob" or "dist":
  - **"log_prob"**: A Tensor of shape *[1]*, the log probability of the "action".
  - **"dist"**: A tf_main:*tf.distributions.Distribution <distributions/Distribution>* with the *log_prob* interface and *log_prob = dist.log_prob(outputs["action"])*.

Parameters

- **env_config** – An instance of *EnvConfig* specifying action space, observation space, and reward range, etc. Use `get_gym_env_config()` to create an EnvConfig from a gym environment.
- **sess** (*optional*) – A tf session. Can be *None* here and set later with *agent.sess = session*.
- **policy** (*optional*) – A policy net that takes in observation and outputs actions and probabilities. If not given, a policy network is created based on *hparams*.
- **policy_kwargs** (*dict, optional*) – Keyword arguments for policy constructor. Note that the *hparams* argument for network constructor is specified in the "policy_hparams" field of *hparams* and should not be included in *policy_kwargs*. Ignored if *policy* is given.
- **policy_caller_kwargs** (*dict, optional*) – Keyword arguments for calling the policy to get actions. The policy is called with `outputs=policy(inputs=observation, **policy_caller_kwargs)`
- **learning_rate** (*optional*) – Learning rate for policy optimization. If not given, determine the learning rate from *hparams*. See *get_train_op()* for more details.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerter will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

static default_hparams()

Returns a dictionary of hyperparameters with default values:

```
{
    'policy_type': 'CategoricalPolicyNet',
    'policy_hparams': None,
    'discount_factor': 0.95,
    'normalize_reward': False,
    'optimization': default_optimization_hparams(),
    'name': 'pg_agent',
}
```

Here:

"policy_type" [str or class or instance] Policy net. Can be class, its name or module path, or a class instance. If class name is given, the class must be from module `texar.modules` or `texar.custom`. Ignored if a *policy* is given to the agent constructor.

"policy_hparams" [dict, optional] Hyperparameters for the policy net. With the `policy_kwargs` argument to the constructor, a network is created with `policy_class(**policy_kwargs, hparams=policy_hparams)`.

"discount_factor" [float] The discount factor of reward.

"normalize_reward" [bool] Whether to normalize the discounted reward, by *(discounted_reward - mean) / std*.

"optimization" [dict] Hyperparameters of optimization for updating the policy net. See *default_optimization_hparams()* for details.

"name" [str] Name of the agent.

sess
The tf session.

env_config
Environment configuration.

get_action(*observ*, *feed_dict=None*)
Gets action according to observation.

Parameters **observ** – Observation from the environment.

Returns action from the policy.

hparams
A `HParams` instance. The hyperparameters of the module.

name
The name of the module (not uniquified).

observe(*reward*, *terminal*, *train_policy=True*, *feed_dict=None*)
Observes experience from environment.

Parameters

- **reward** – Reward of the action. The configuration (e.g., shape) of the reward is defined in *env_config*.
- **terminal** (*bool*) – Whether the episode is terminated.
- **train_policy** (*bool*) – Wether to update the policy for this step.
- **feed_dict** (*dict, optional*) – Any stuffs fed to running the training operator.

policy
The policy model.

reset()
Resets the states to begin new episode.

variable_scope
The variable scope of the agent.

DQNAgent

class texar.agents.**DQNAgent**(*env_config*, *sess=None*, *qnet=None*, *target=None*, *qnet_kwargs=None*, *qnet_caller_kwargs=None*, *replay_memory=None*, *replay_memory_kwargs=None*, *exploration=None*, *exploration_kwargs=None*, *hparams=None*)

Deep Q learning agent for episodic setting.

A Q learning algorithm consists of several components:

- A **Q-net** takes in a state and returns Q-value for action sampling. See `CategoricalQNet` for an example Q-net class and required interface.
- A **replay memory** manages past experience for Q-net updates. See `DequeReplayMemory` for an example replay memory class and required interface.
- An **exploration** that specifies the exploration strategy used to train the Q-net. See `EpsilonLinearDecayExploration` for an example class and required interface.

Parameters

- **env_config** – An instance of `EnvConfig` specifying action space, observation space, and reward range, etc. Use `get_gym_env_config()` to create an EnvConfig from a gym environment.
- **sess** (*optional*) – A tf session. Can be *None* here and set later with *agent.sess = session*.
- **qnet** (*optional*) – A Q network that predicts Q values given states. If not given, a Q network is created based on `hparams`.
- **target** (*optional*) – A target network to compute target Q values.
- **qnet_kwargs** (*dict, optional*) – Keyword arguments for qnet constructor. Note that the *hparams* argument for network constructor is specified in the "policy_hparams" field of `hparams` and should not be included in *policy_kwargs*. Ignored if `qnet` is given.
- **qnet_caller_kwargs** (*dict, optional*) – Keyword arguments for calling *qnet* to get Q values. The *qnet* is called with `outputs=qnet(inputs=observation, **qnet_caller_kwargs)`
- **replay_memory** (*optional*) – A replay memory instance. If not given, a replay memory is created based on `hparams`.
- **replay_memory_kwargs** (*dict, optional*) – Keyword arguments for replay_memory constructor. Ignored if `replay_memory` is given.
- **exploration** (*optional*) – An exploration instance used in the algorithm. If not given, an exploration instance is created based on `hparams`.
- **exploration_kwargs** (*dict, optional*) – Keyword arguments for exploration class constructor. Ignored if `exploration` is given.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerters will be set to default values. See `default_hparams()` for the hyperparameter sturcture and default values.

static default_hparams()

Returns a dictionary of hyperparameters with default values:

```
{
    'qnet_type': 'CategoricalQNet',
    'qnet_hparams': None,
    'replay_memory_type': 'DequeReplayMemory',
    'replay_memory_hparams': None,
    'exploration_type': 'EpsilonLinearDecayExploration',
    'exploration_hparams': None,
    'optimization': opt.default_optimization_hparams(),
    'target_update_strategy': 'copy',
    'cold_start_steps': 100,
    'sample_batch_size': 32,
    'update_period': 100,
    'discount_factor': 0.95,
    'name': 'dqn_agent'
}
```

Here:

"qnet_type" [str or class or instance] Q-value net. Can be class, its name or module path, or a class instance. If class name is given, the class must be from module `texar.modules` or `texar.custom`. Ignored if a *qnet* is given to the agent constructor.

"qnet_hparams" [dict, optional] Hyperparameters for the Q net. With the `qnet_kwargs` argument to the constructor, a network is created with `qnet_class(**qnet_kwargs, hparams=qnet_hparams)`.

"replay_memory_type" [str or class or instance] Replay memory class. Can be class, its name or module path, or a class instance. If class name is given, the class must be from module `texar.core` or `texar.custom`. Ignored if a *replay_memory* is given to the agent constructor.

"replay_memory_hparams" [dict, optional] Hyperparameters for the replay memory. With the `replay_memory_kwargs` argument to the constructor, a network is created with `replay_memory_class( **replay_memory_kwargs, hparams=replay_memory_hparams)`.

"exploration_type" [str or class or instance] Exploration class. Can be class, its name or module path, or a class instance. If class name is given, the class must be from module `texar.core` or `texar.custom`. Ignored if a *exploration* is given to the agent constructor.

"exploration_hparams" [dict, optional] Hyperparameters for the exploration class. With the `exploration_kwargs` argument to the constructor, a network is created with `exploration_class( **exploration_kwargs, hparams=exploration_hparams)`.

"optimization" [dict] Hyperparameters of optimization for updating the Q-net. See *`default_optimization_hparams()`* for details.

"cold_start_steps": int In the beginning, Q-net is not trained in the first few steps.

"sample_batch_size": int The number of samples taken in replay memory when training.

"target_update_strategy": string

- If **"copy"**, the target network is assigned with the parameter of Q-net every `"update_period"` steps.
- If **"tau"**, target will be updated by assigning as

` (1 - 1/update_period) * target + 1/update_period * qnet `

"update_period": int Frequecy of updating the target network, i.e., updating the target once for every "update_period" steps.

"discount_factor" [float] The discount factor of reward.

"name" [str] Name of the agent.

env_config
Environment configuration.

get_action(*observ*, *feed_dict=None*)
Gets action according to observation.

Parameters **observ** – Observation from the environment.

Returns action from the policy.

hparams
A `HParams` instance. The hyperparameters of the module.

name
The name of the module (not uniquified).

observe(*reward*, *terminal*, *train_policy=True*, *feed_dict=None*)
Observes experience from environment.

Parameters

- **reward** – Reward of the action. The configuration (e.g., shape) of the reward is defined in *env_config*.
- **terminal** (*bool*) – Whether the episode is terminated.
- **train_policy** (*bool*) – Wether to update the policy for this step.
- **feed_dict** (*dict, optional*) – Any stuffs fed to running the training operator.

reset()
Resets the states to begin new episode.

sess
The tf session.

variable_scope
The variable scope of the agent.

ActorCriticAgent

class texar.agents.**ActorCriticAgent**(*env_config*, *sess=None*, *actor=None*, *actor_kwargs=None*, *critic=None*, *critic_kwargs=None*, *hparams=None*)

Actor-critic agent for episodic setting.

An actor-critic algorithm consists of several components:

- **Actor** is the policy to optimize. As a temporary implementation, here by default we use a *PGAgent* instance that wraps a *policy net* and provides proper interfaces to perform the role of an actor.
- **Critic** that provides learning signals to the actor. Again, as a temporary implemetation, here by default we use a *DQNAgent* instance that wraps a *Q net* and provides proper interfaces to perform the role of a critic.

Parameters

- **env_config** – An instance of *EnvConfig* specifying action space, observation space, and reward range, etc. Use `get_gym_env_config()` to create an EnvConfig from a gym environment.
- **sess** (*optional*) – A tf session. Can be *None* here and set later with *agent.sess = session*.
- **actor** (*optional*) – An instance of *PGAgent* that performs as actor in the algorithm. If not provided, an actor is created based on *hparams*.
- **actor_kwargs** (*dict, optional*) – Keyword arguments for actor constructor. Note that the *hparams* argument for actor constructor is specified in the "actor_hparams" field of *hparams* and should not be included in *actor_kwargs*. Ignored if `actor` is given.
- **critic** (*optional*) – An instance of *DQNAgent* that performs as critic in the algorithm. If not provided, a critic is created based on *hparams*.
- **critic_kwargs** (*dict, optional*) – Keyword arguments for critic constructor. Note that the *hparams* argument for critic constructor is specified in the "critic_hparams" field of *hparams* and should not be included in *critic_kwargs*. Ignored if `critic` is given.
- **hparams** (*dict or HParams, optional*) – Hyperparameters. Missing hyperparamerters will be set to default values. See *default_hparams()* for the hyperparameter sturcture and default values.

static default_hparams()

Returns a dictionary of hyperparameters with default values:

```
{
    'actor_type': 'PGAgent',
    'actor_hparams': None,
    'critic_type': 'DQNAgent',
    'critic_hparams': None,
    'name': 'actor_critic_agent'
}
```

Here:

"actor_type" [str or class or instance] Actor. Can be class, its name or module path, or a class instance. If class name is given, the class must be from module `texar.agents` or `texar.custom`. Ignored if a *actor* is given to the agent constructor.

"actor_kwargs" [dict, optional] Hyperparameters for the actor class. With the `actor_kwargs` argument to the constructor, an actor is created with `actor_class(**actor_kwargs, hparams=actor_hparams)`.

"critic_type" [str or class or instance] Critic. Can be class, its name or module path, or a class instance. If class name is given, the class must be from module `texar.agents` or `texar.custom`. Ignored if a *critic* is given to the agent constructor.

"critic_kwargs" [dict, optional] Hyperparameters for the critic class. With the `critic_kwargs` argument to the constructor, an critic is created with `critic_class(**critic_kwargs, hparams=critic_hparams)`.

"name" [str] Name of the agent.

get_action(*observ*, *feed_dict=None*)

Gets action according to observation.

Parameters **observ** – Observation from the environment.

Returns action from the policy.

env_config
Environment configuration.

hparams
A `HParams` instance. The hyperparameters of the module.

name
The name of the module (not uniquified).

observe(*reward*, *terminal*, *train_policy=True*, *feed_dict=None*)
Observes experience from environment.

Parameters

- **reward** – Reward of the action. The configuration (e.g., shape) of the reward is defined in *env_config*.
- **terminal** (*bool*) – Whether the episode is terminated.
- **train_policy** (*bool*) – Wether to update the policy for this step.
- **feed_dict** (*dict, optional*) – Any stuffs fed to running the training operator.

reset()
Resets the states to begin new episode.

sess
The tf session.

variable_scope
The variable scope of the agent.

3.5.3 Agent Utils

Space

class texar.agents.**Space**(*shape=None*, *low=None*, *high=None*, *dtype=None*)
Observation and action spaces. Describes valid actions and observations. Similar to gym.Space.

Parameters

- **shape** (*optional*) – Shape of the space, a tuple. If not given, infers from *low* and *high*.
- **low** (*optional*) – Lower bound (inclusive) of each dimension of the space. Must have shape as specified by *shape*, and of the same shape with with *high* (if given). If *None*, set to *-inf* for each dimension.
- **high** (*optional*) – Upper bound (inclusive) of each dimension of the space. Must have shape as specified by *shape*, and of the same shape with with *low* (if given). If *None*, set to *inf* for each dimension.
- **dtype** (*optional*) – Data type of elements in the space. If not given, infers from *low* (if given) or set to *float*.

Example

```
s = Space(low=0, high=10, dtype=np.int32)
#s.contains(2) == True
#s.contains(10) == True
#s.contains(11) == False
#s.shape == ()

s2 = Space(shape=(2,2), high=np.ones([2,2]), dtype=np.float)
#s2.low == [[-inf, -inf], [-inf, -inf]]
#s2.high == [[1., 1.], [1., 1.]]
```

contains(*x*)
Checks if x is contained in the space. Returns a *bool*.

shape
Shape of the space.

low
Lower bound of the space.

high
Upper bound of the space.

dtype
Data type of the element.

EnvConfig

class texar.agents.**EnvConfig**(*action_space*, *observ_space*, *reward_range*)
Configurations of an environment.

Parameters

- **action_space** – An instance of *Space* or gym.Space, the action space.
- **observ_space** – An instance of *Space* or gym.Space, the observation space.
- **reward_range** – A tuple corresponding to the min and max possible rewards, e.g., *reward_range=(-1.0, 1.0)*.

convert_gym_space

get_gym_env_config

3.6 Loss Functions

3.6.1 MLE Loss

sequence_softmax_cross_entropy

texar.losses.**sequence_softmax_cross_entropy**(*labels*, *logits*, *sequence_length*, *average_across_batch=True*, *average_across_timesteps=False*, *sum_over_batch=False*, *sum_over_timesteps=True*, *time_major=False*, *stop_gradient_to_label=False*, *name=None*)

Computes softmax cross entropy for each time step of sequence predictions.

Parameters

- **labels** – Target class distributions.
  - If `time_major` is *False* (default), this must be a Tensor of shape *[batch_size, max_time, num_classes]*.
  - If *time_major* is *True*, this must be a Tensor of shape *[max_time, batch_size, num_classes]*.

  Each row of *labels* should be a valid probability distribution, otherwise, the computation of the gradient will be incorrect.
- **logits** – Unscaled log probabilities. This must have the shape of *[max_time, batch_size, num_classes]* or *[batch_size, max_time, num_classes]* according to the value of *time_major*.
- **sequence_length** – A Tensor of shape *[batch_size]*. Time steps beyond the respective sequence lengths will have zero losses.
- **average_across_timesteps** (*bool*) – If set, average the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **average_across_batch** (*bool*) – If set, average the loss across the batch dimension. Must not set *average_across_batch*' and *sum_over_batch* at the same time.
- **sum_over_timesteps** (*bool*) – If set, sum the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **sum_over_batch** (*bool*) – If set, sum the loss across the batch dimension. Must not set *average_across_batch* and *sum_over_batch* at the same time.
- **time_major** (*bool*) – The shape format of the inputs. If *True*, `labels` and `logits` must have shape *[max_time, batch_size, ...]*. If *False* (default), they must have shape *[batch_size, max_time, ...]*.
- **stop_gradient_to_label** (*bool*) – If set, gradient propagation to `labels` will be disabled.
- **name** (*str, optional*) – A name for the operation.

Returns

A Tensor containing the loss, of rank 0, 1, or 2 depending on the arguments `{average_across}/{sum_over}_{timesteps}/{batch}`. For example:

- If `sum_over_timesteps` and `average_across_batch` are *True* (default), the return Tensor is of rank 0.
- If `average_across_batch` is *True* and other arguments are *False*, the return Tensor is of shape *[max_time]*.

sequence_sparse_softmax_cross_entropy

`texar.losses.`**`sequence_sparse_softmax_cross_entropy`**(*labels*, *logits*, *sequence_length*, *average_across_batch=True*, *average_across_timesteps=False*, *sum_over_batch=False*, *sum_over_timesteps=True*, *time_major=False*, *name=None*)

Computes sparse softmax cross entropy for each time step of sequence predictions.

Parameters

- **`labels`** – Target class indexes. I.e., classes are mutually exclusive (each entry is in exactly one class).
  - If `time_major` is *False* (default), this must be a Tensor of shape *[batch_size, max_time]*.
  - If *time_major* is *True*, this must be a Tensor of shape *[max_time, batch_size]*.
- **`logits`** – Unscaled log probabilities. This must have the shape of *[max_time, batch_size, num_classes]* or *[batch_size, max_time, num_classes]* according to the value of *time_major*.
- **`sequence_length`** – A Tensor of shape *[batch_size]*. Time steps beyond the respective sequence lengths will have zero losses.
- **`average_across_timesteps`** (*bool*) – If set, average the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **`average_across_batch`** (*bool*) – If set, average the loss across the batch dimension. Must not set *average_across_batch*' and *sum_over_batch* at the same time.
- **`sum_over_timesteps`** (*bool*) – If set, sum the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **`sum_over_batch`** (*bool*) – If set, sum the loss across the batch dimension. Must not set *average_across_batch* and *sum_over_batch* at the same time.
- **`time_major`** (*bool*) – The shape format of the inputs. If *True*, `labels` and `logits` must have shape *[max_time, batch_size, ...]*. If *False* (default), they must have shape *[batch_size, max_time, ...]*.
- **`name`** (*str, optional*) – A name for the operation.

Returns

A Tensor containing the loss, of rank 0, 1, or 2 depending on the arguments `{average_across}/{sum_over}_{timesteps}/{batch}`. For example:

- If `sum_over_timesteps` and `average_across_batch` are *True* (default), the return Tensor is of rank 0.

- If `average_across_batch` is *True* and other arguments are *False*, the return Tensor is of shape *[max_time]*.

Example

```
embedder = WordEmbedder(vocab_size=data.vocab.size)
decoder = BasicRNNDecoder(vocab_size=data.vocab.size)
outputs, _, _ = decoder(
    decoding_strategy='train_greedy',
    inputs=embedder(data_batch['text_ids']),
    sequence_length=data_batch['length']-1)

loss = sequence_sparse_softmax_cross_entropy(
    labels=data_batch['text_ids'][:, 1:],
    logits=outputs.logits,
    sequence_length=data_batch['length']-1)
```

sequence_sigmoid_cross_entropy

texar.losses.**sequence_sigmoid_cross_entropy**(*labels*, *logits*, *sequence_length*, *average_across_batch=True*, *average_across_timesteps=False*, *average_across_classes=True*, *sum_over_batch=False*, *sum_over_timesteps=True*, *sum_over_classes=False*, *time_major=False*, *stop_gradient_to_label=False*, *name=None*)

Computes sigmoid cross entropy for each time step of sequence predictions.

Parameters

- **labels** – Target class distributions.
  - If `time_major` is *False* (default), this must be a Tensor of shape *[batch_size, max_time(, num_classes)]*.
  - If *time_major* is *True*, this must be a Tensor of shape *[max_time, batch_size(, num_classes)]*.

  Each row of *labels* should be a valid probability distribution, otherwise, the computation of the gradient will be incorrect.
- **logits** – Unscaled log probabilities having the same shape as with `labels`.
- **sequence_length** – A Tensor of shape *[batch_size]*. Time steps beyond the respective sequence lengths will have zero losses.
- **average_across_timesteps** (*bool*) – If set, average the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **average_across_batch** (*bool*) – If set, average the loss across the batch dimension. Must not set *average_across_batch*' and *sum_over_batch* at the same time.
- **average_across_classes** (*bool*) – If set, average the loss across the class dimension (if exists). Must not set *average_across_classes*' and *sum_over_classes* at the same time. Ignored if `logits` is a 2D Tensor.

- **sum_over_timesteps** (*bool*) – If set, sum the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **sum_over_batch** (*bool*) – If set, sum the loss across the batch dimension. Must not set *average_across_batch* and *sum_over_batch* at the same time.
- **sum_over_classes** (*bool*) – If set, sum the loss across the class dimension. Must not set *average_across_classes* and *sum_over_classes* at the same time. Ignored if `logits` is a 2D Tensor.
- **time_major** (*bool*) – The shape format of the inputs. If *True*, `labels` and `logits` must have shape *[max_time, batch_size, ...]*. If *False* (default), they must have shape *[batch_size, max_time, ...]*.
- **stop_gradient_to_label** (*bool*) – If set, gradient propagation to `labels` will be disabled.
- **name** (*str, optional*) – A name for the operation.

Returns

A Tensor containing the loss, of rank 0, 1, or 2 depending on the arguments `{average_across}/{sum_over}_{timesteps}/{batch}/{classes}`. For example, if the class dimension does not exist, and

- If `sum_over_timesteps` and `average_across_batch` are *True* (default), the return Tensor is of rank 0.
- If `average_across_batch` is *True* and other arguments are *False*, the return Tensor is of shape *[max_time]*.

binary_sigmoid_cross_entropy

`texar.losses.`**binary_sigmoid_cross_entropy**(*pos_logits=None*, *neg_logits=None*, *average_across_batch=True*, *average_across_classes=True*, *sum_over_batch=False*, *sum_over_classes=False*, *return_pos_neg_losses=False*, *name=None*)

Computes sigmoid cross entropy of binary predictions.

Parameters

- **pos_logits** – The logits of predicting positive on positive data. A tensor of shape *[batch_size(, num_classes)]*.
- **neg_logits** – The logits of predicting positive on negative data. A tensor of shape *[batch_size(, num_classes)]*.
- **average_across_batch** (*bool*) – If set, average the loss across the batch dimension. Must not set *average_across_batch*' and *sum_over_batch* at the same time.
- **average_across_classes** (*bool*) – If set, average the loss across the class dimension (if exists). Must not set *average_across_classes*' and *sum_over_classes* at the same time. Ignored if `logits` is a 1D Tensor.
- **sum_over_batch** (*bool*) – If set, sum the loss across the batch dimension. Must not set *average_across_batch* and *sum_over_batch* at the same time.

- **sum_over_classes** (*bool*) – If set, sum the loss across the class dimension. Must not set *average_across_classes* and *sum_over_classes* at the same time. Ignored if `logits` is a 2D Tensor.
- **return_pos_neg_losses** (*bool*) – If set, additionally returns the losses on `pos_logits` and `neg_logits`, respectively.
- **name** (*str, optional*) – A name for the operation.

Returns

By default, a Tensor containing the loss, of rank 0, 1, or 2 depending on the arguments `{average_across}/{sum_over}_{batch}/{classes}`. For example:

- If `sum_over_batch` and `average_across_classes` are *True* (default), the return Tensor is of rank 0.
- If arguments are *False*, the return Tensor is of shape *[batch_size(, num_classes)]*.

If `return_pos_neg_losses` is *True*, returns a tuple *(loss, pos_loss, neg_loss)*, where *loss* is the loss above; *pos_loss* is the loss on *pos_logits* only; and *neg_loss* is the loss on *neg_logits* only. They have *loss* = *pos_loss* + *neg_loss*.

binary_sigmoid_cross_entropy_with_clas

texar.losses.**binary_sigmoid_cross_entropy_with_clas**(*clas_fn*, *pos_inputs=None*, *neg_inputs=None*, *average_across_batch=True*, *average_across_classes=True*, *sum_over_batch=False*, *sum_over_classes=False*, *return_pos_neg_losses=False*, *name=None*)

Computes sigmoid cross entropy of binary classifier.

Parameters

- **clas_fn** – A callable takes data (e.g., `pos_inputs` and `fake_inputs`) and returns the logits of being positive. The signature of *clas_fn* must be: `logits (, ...) = clas_fn(inputs)`. The return value of *clas_fn* can be the logits, or a tuple where the logits are the first element.
- **pos_inputs** – The positive data fed into *clas_fn*.
- **neg_inputs** – The negative data fed into *clas_fn*.
- **average_across_batch** (*bool*) – If set, average the loss across the batch dimension. Must not set *average_across_batch*' and *sum_over_batch* at the same time.
- **average_across_classes** (*bool*) – If set, average the loss across the class dimension (if exists). Must not set *average_across_classes*' and *sum_over_classes* at the same time. Ignored if `logits` is a 1D Tensor.
- **sum_over_batch** (*bool*) – If set, sum the loss across the batch dimension. Must not set *average_across_batch* and *sum_over_batch* at the same time.
- **sum_over_classes** (*bool*) – If set, sum the loss across the class dimension. Must not set *average_across_classes* and *sum_over_classes* at the same time. Ignored if `logits` is a 2D Tensor.

- **return_pos_neg_losses** (*bool*) – If set, additionally returns the losses on `pos_logits` and `neg_logits`, respectively.
- **name** (*str, optional*) – A name for the operation.

Returns

By default, a Tensor containing the loss, of rank 0, 1, or 2 depending on the arguments `{average_across}/{sum_over}_{batch}/{classes}`. For example:

- If `sum_over_batch` and `average_across_classes` are *True* (default), the return Tensor is of rank 0.
- If arguments are *False*, the return Tensor is of shape *[batch_size(, num_classes)]*.

If `return_pos_neg_losses`=`True`, returns a tuple *(loss, pos_loss, neg_loss)*, where *loss* is the loss above; *pos_loss* is the loss on *pos_logits* only; and *neg_loss* is the loss on *neg_logits* only. They have *loss* = *pos_loss* + *neg_loss*.

3.6.2 Policy Gradient Loss

pg_loss_with_logits

texar.losses.**pg_loss_with_logits**(*actions*, *logits*, *advantages*, *rank=None*, *batched=False*, *sequence_length=None*, *average_across_batch=True*, *average_across_timesteps=False*, *average_across_remaining=False*, *sum_over_batch=False*, *sum_over_timesteps=True*, *sum_over_remaining=True*, *time_major=False*)

Policy gradient loss with logits. Used for discrete actions.

*pg_loss = reduce(advantages * -log_prob(actions))*, where *advantages* and *actions* do not back-propagate gradients.

All arguments except `logits` and `actions` are the same with *pg_loss_with_log_probs()*.

Parameters

- **actions** – Tensor of shape *[(batch_size,) max_time, d_3, ..., d_rank]* and of dtype *int32* or *int64*. The rank of the Tensor is specified with `rank`.

  The batch dimension exists only if `batched` is *True*.

  The batch and time dimensions are exchanged, i.e., *[max_time, batch_size, ...]* if `time_major` is *True*.
- **logits** – Unscaled log probabilities of shape *[(batch_size,) max_time, d_3, ..., d_{rank+1}]* and dtype *float32* or *float64*. The batch and time dimensions are exchanged if *time_major* is *True*.
- **advantages** – Tensor of shape *[(batch_size,) max_time, d_3, ..., d_rank]* and dtype *float32* or *float64*. The batch and time dimensions are exchanged if *time_major* is *True*.
- **rank** (*int, optional*) – The rank of `actions`. If *None* (default), rank is automatically inferred from *actions* or *advantages*. If the inference fails, *rank* is set to 1 if `batched` is *False*, and set to 2 if `batched` is *True*.
- **batched** (*bool*) – *True* if the inputs are batched.
- **sequence_length** (*optional*) – A Tensor of shape *[batch_size]*. Time steps beyond the respective sequence lengths will have zero losses. Used if `batched` is *True*.

- **average_across_timesteps** (*bool*) – If set, average the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **average_across_batch** (*bool*) – If set, average the loss across the batch dimension. Must not set *average_across_batch*' and *sum_over_batch* at the same time. Ignored if *batched* is *False*.
- **average_across_remaining** (*bool*) – If set, average the sequence across the remaining dimensions. Must not set *average_across_remaining*' and *sum_over_remaining* at the same time. Ignored if no more dimensions other than the batch and time dimensions.
- **sum_over_timesteps** (*bool*) – If set, sum the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **sum_over_batch** (*bool*) – If set, sum the loss across the batch dimension. Must not set *average_across_batch* and *sum_over_batch* at the same time. Ignored if *batched* is *False*.
- **sum_over_remaining** (*bool*) – If set, sum the loss across the remaining dimension. Must not set *average_across_remaining* and *sum_over_remaining* at the same time. Ignored if no more dimensions other than the batch and time dimensions.
- **time_major** (*bool*) – The shape format of the inputs. If *True*, `logits`, `actions` and `advantages` must have shape *[max_time, batch_size, ...]*. If *False* (default), they must have shape *[batch_size, max_time, ...]*. Ignored if *batched* is *False*.

Returns A Tensor containing the loss to minimize, whose rank depends on the reduce arguments. For example, the batch dimension is reduced if either `average_across_batch` or `sum_over_batch` is *True*, which decreases the rank of output tensor by 1.

pg_loss_with_log_probs

`texar.losses.`**pg_loss_with_log_probs**(*log_probs*, *advantages*, *rank=None*, *batched=False*, *sequence_length=None*, *average_across_batch=True*, *average_across_timesteps=False*, *average_across_remaining=False*, *sum_over_batch=False*, *sum_over_timesteps=True*, *sum_over_remaining=True*, *time_major=False*)

Policy gradient loss with log probs of actions.

pg_loss = *reduce(advantages * -log_probs)*, where *advantages* does not back-propagate gradients.

All arguments except `log_probs` are the same as *pg_loss_with_logits()*.

Parameters

- **log_probs** – Log probabilities of shape *[(batch_size,) max_time, ..., d_rank]* and dtype *float32* or *float64*. The rank of the Tensor is specified with `rank`.

  The batch dimension exists only if `batched` is *True*.

  The batch and time dimensions are exchanged, i.e., *[max_time, batch_size, ...]* if `time_major` is *True*.
- **advantages** – Tensor of shape *[(batch_size,) max_time, d_3, ..., d_rank]* and dtype *float32* or *float64*. The batch dimension exists only if *batched* is *True*. The batch and time dimensions are exchanged if *time_major* is *True*.
- **rank** (*int, optional*) – The rank of `log_probs`. If *None* (default), rank is automatically inferred from *log_probs* or *advantages*. If the inference fails, *rank* is set to 1 if *batched''==False*, and set to 2 if *batched''==True*.

- **batched** (*bool*) – *True* if the inputs are batched.
- **sequence_length** (*optional*) – A Tensor of shape *[batch_size]*. Time steps beyond the respective sequence lengths will have zero losses. Used if `batched` is *True*.
- **average_across_timesteps** (*bool*) – If set, average the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **average_across_batch** (*bool*) – If set, average the loss across the batch dimension. Must not set *average_across_batch*' and *sum_over_batch* at the same time. Ignored if *batched* is *False*.
- **average_across_remaining** (*bool*) – If set, average the sequence across the remaining dimensions. Must not set *average_across_remaining*' and *sum_over_remaining* at the same time. Ignored if no more dimensions other than the batch and time dimensions.
- **sum_over_timesteps** (*bool*) – If set, sum the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **sum_over_batch** (*bool*) – If set, sum the loss across the batch dimension. Must not set *average_across_batch* and *sum_over_batch* at the same time. Ignored if *batched* is *False*.
- **sum_over_remaining** (*bool*) – If set, sum the loss across the remaining dimension. Must not set *average_across_remaining* and *sum_over_remaining* at the same time. Ignored if no more dimensions other than the batch and time dimensions.
- **time_major** (*bool*) – The shape format of the inputs. If *True*, `log_probs` and `advantages` must have shape *[max_time, batch_size, ...]*. If *False* (default), they must have shape *[batch_size, max_time, ...]*. Ignored if `batched` is *False*.

Returns A Tensor containing the loss to minimize, whose rank depends on the reduce arguments. For example, the batch dimension is reduced if either `average_across_batch` or `sum_over_batch` is *True*, which decreases the rank of output tensor by 1.

3.6.3 Reward

discount_reward

`texar.losses.`**discount_reward**(*reward*, *sequence_length=None*, *discount=1.0*, *normalize=False*, *dtype=None*, *tensor_rank=1*)

Computes discounted reward.

`reward` and `sequence_length` can be either Tensors or python arrays. If both are python array (or *None*), the return will be a python array as well. Otherwise tf Tensors are returned.

Parameters

- **reward** – A Tensor or python array. Can be 1D with shape *[batch_size]*, or 2D with shape *[batch_size, max_time]*.
- **sequence_length** (*optional*) – A Tensor or python array of shape *[batch_size]*. Time steps beyond the respective sequence lengths will be masked. Required if `reward` is 1D.
- **discount** (*float*) – A scalar. The discount factor.
- **normalize** (*bool*) – Whether to normalize the discounted reward, by *(discounted_reward - mean) / std*. Here *mean* and *std* are over all time steps and all samples in the batch.
- **dtype** (*dtype*) – Type of `reward`. If *None*, infer from *reward* automatically.

- **tensor_rank** (*int*) – The number of dimensions of `reward`. Default is 1, i.e., `reward` is a 1D Tensor consisting of a batch dimension. Ignored if `reward` and `sequence_length` are python arrays (or *None*).

Returns

A 2D Tensor or python array of the discounted reward.

If `reward` and `sequence_length` are python arrays (or *None*), the returned value is a python array as well.

Example

```
r = [2., 1.]
seq_length = [3, 2]
discounted_r = discount_reward(r, seq_length, discount=0.1)
# discounted_r == [[2. * 0.1^2, 2. * 0.1, 2.],
#                  [1. * 0.1,   1.,       0.]]

r = [[3., 4., 5.], [6., 7., 0.]]
seq_length = [3, 2]
discounted_r = discount_reward(r, seq_length, discount=0.1)
# discounted_r == [[3. + 4.*0.1 + 5.*0.1^2, 4. + 5.*0.1, 5.],
#                  [6. + 7.*0.1,            7.,          0.]]
```

3.6.4 Adversarial Loss

binary_adversarial_losses

`texar.losses.`**binary_adversarial_losses**(*real_data*, *fake_data*, *discriminator_fn*, *mode='max_real'*)

Computes adversarial losses of real/fake binary discrimination game.

Parameters

- **real_data** (*Tensor or array*) – Real data of shape *[num_real_examples, ...]*.
- **fake_data** (*Tensor or array*) – Fake data of shape *[num_fake_examples, ...]*. *num_real_examples* does not necessarily equal *num_fake_examples*.
- **discriminator_fn** – A callable takes data (e.g., `real_data` and `fake_data`) and returns the logits of being real. The signature of *discriminator_fn* must be: `logits, ... = discriminator_fn(data)`. The return value of *discriminator_fn* can be the logits, or a tuple where the logits are the first element.
- **mode** (*str*) – Mode of the generator loss. Either "max_real" or "min_fake".
  - **"max_real"** (default): minimizing the generator loss is to maximize the probability of fake data being classified as real.
  - **"min_fake"**: minimizing the generator loss is to minimize the probability of fake data being classified as fake.

Returns A tuple *(generator_loss, discriminator_loss)* each of which is a scalar Tensor, loss to be minimized.

3.6.5 Entropy

entropy_with_logits

texar.losses.**entropy_with_logits**(*logits*, *rank=None*, *average_across_batch=True*, *average_across_remaining=False*, *sum_over_batch=False*, *sum_over_remaining=True*)

Shannon entropy given logits.

Parameters

- **logits** – Unscaled log probabilities of shape *[batch_size, d_2, ..., d_{rank-1}, distribution_dim]* and of dtype *float32* or *float64*.

  The rank of the tensor is optionally specified by the argument `rank`.

  The tensor is considered as having *[batch_size, .., d_{rank-1}]* elements, each of which has a distribution of length *d_rank* (i.e., *distribution_dim*). So the last dimension is always summed out to compute the entropy.

- **rank** (*int, optional*) – The rank of `logits`. If *None* (default), *rank* is inferred automatically from *logits*. If the inference fails, *rank* is set to 2, i.e., assuming `logits` is of shape *[batch_size, distribution_dim]*

- **average_across_batch** (*bool*) – If set, average the entropy across the batch dimension. Must not set *average_across_batch*' and *sum_over_batch* at the same time.

- **average_across_remaining** (*bool*) – If set, average the entropy across the remaining dimensions. Must not set *average_across_remaining*' and *sum_over_remaining* at the same time. Used only when `logits` has rank >= 3.

- **sum_over_batch** (*bool*) – If set, sum the entropy across the batch dimension. Must not set *average_across_batch* and *sum_over_batch* at the same time.

- **sum_over_remaining** (*bool*) – If set, sum the entropy across the remaining dimension. Must not set *average_across_remaining* and *sum_over_remaining* at the same time. Used only when `logits` has rank >= 3.

Returns A Tensor containing the shannon entropy. The dimensionality of the Tensor depends on the configuration of reduction arguments. For example, if both batch and remaining dimensions are reduced (by either sum or average), the returned Tensor is a scalar Tensor.

sequence_entropy_with_logits

texar.losses.**sequence_entropy_with_logits**(*logits*, *rank=None*, *sequence_length=None*, *average_across_batch=True*, *average_across_timesteps=False*, *average_across_remaining=False*, *sum_over_batch=False*, *sum_over_timesteps=True*, *sum_over_remaining=True*, *time_major=False*)

Shannon entropy given logits.

Parameters

- **logits** – Unscaled log probabilities of shape *[batch_size, max_time, d_3, ..., d_{rank-1}, distribution_dim]* and of dtype *float32* or *float64*.

  The rank of the tensor is optionally specified by the argument `rank`.

The tensor is considered as having *[batch_size, .., d_{rank-1}]* elements, each of which has a distribution of length *d_rank* (i.e., *distribution_dim*). So the last dimension is always summed out to compute the entropy.

The batch and time dimensions are exchanged if `time_major` is *True*.

- **rank** (*int, optional*) – The rank of `logits`. If *None* (default), *rank* is inferred automatically from *logits*. If the inference fails, *rank* is set to 3, i.e., assuming *logits* is of shape *[batch_size, max_time, distribution_dim]*
- **sequence_length** (*optional*) – A Tensor of shape *[batch_size]*. Time steps beyond the respective sequence lengths are counted into the entropy.
- **average_across_timesteps** (*bool*) – If set, average the entropy across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **average_across_batch** (*bool*) – If set, average the entropy across the batch dimension. Must not set *average_across_batch*' and *sum_over_batch* at the same time.
- **average_across_remaining** (*bool*) – If set, average the entropy across the remaining dimensions. Must not set *average_across_remaining*' and *sum_over_remaining* at the same time. Used only when `logits` has rank >= 4.
- **sum_over_timesteps** (*bool*) – If set, sum the entropy across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **sum_over_batch** (*bool*) – If set, sum the entropy across the batch dimension. Must not set *average_across_batch* and *sum_over_batch* at the same time.
- **sum_over_remaining** (*bool*) – If set, sum the entropy across the remaining dimension. Must not set *average_across_remaining* and *sum_over_remaining* at the same time. Used only when `logits` has rank >= 4.
- **time_major** (*bool*) – The shape format of the inputs. If *True*, `logits` must have shape *[max_time, batch_size, ...]*. If *False* (default), it must have shape *[batch_size, max_time, ...]*.

Returns A Tensor containing the shannon entropy. The dimensionality of the Tensor depends on the configuration of reduction arguments. For example, if batch, time, and remaining dimensions are all reduced (by either sum or average), the returned Tensor is a scalar Tensor.

3.6.6 Loss Utils

mask_and_reduce

`texar.losses.`**mask_and_reduce**(*sequence*, *sequence_length*, *rank=2*, *average_across_batch=True*, *average_across_timesteps=False*, *average_across_remaining=False*, *sum_over_batch=False*, *sum_over_timesteps=True*, *sum_over_remaining=True*, *dtype=None*, *time_major=False*)

Masks out sequence entries that are beyond the respective sequence lengths, and reduces (average or sum) away dimensions.

This is a combination of `mask_sequences()` and `reduce_batch_time()`.

Parameters

- **sequence** – A Tensor of sequence values. If *time_major=False* (default), this must be a Tensor of shape *[batch_size, max_time, d_2, ..., d_rank]*, where the rank of the Tensor is specified with `rank`. The batch and time dimensions are exchanged if *time_major* is True.
- **sequence_length** – A Tensor of shape *[batch_size]*. Time steps beyond the respective sequence lengths will be made zero. If *None*, not masking is performed.
- **rank** (*int*) – The rank of `sequence`. Must be >= 2. Default is 2, i.e., *sequence* is a 2D Tensor consisting of batch and time dimensions.
- **average_across_timesteps** (*bool*) – If set, average the sequence across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **average_across_batch** (*bool*) – If set, average the sequence across the batch dimension. Must not set *average_across_batch*' and *sum_over_batch* at the same time.
- **average_across_remaining** (*bool*) – If set, average the sequence across the remaining dimensions. Must not set *average_across_remaining*' and *sum_over_remaining* at the same time.
- **sum_over_timesteps** (*bool*) – If set, sum the loss across the time dimension. Must not set *average_across_timesteps* and *sum_over_timesteps* at the same time.
- **sum_over_batch** (*bool*) – If set, sum the loss across the batch dimension. Must not set *average_across_batch* and *sum_over_batch* at the same time.
- **sum_over_remaining** (*bool*) – If set, sum the loss across the remaining dimension. Must not set *average_across_remaining* and *sum_over_remaining* at the same time.
- **time_major** (*bool*) – The shape format of the inputs. If *True*, `sequence` must have shape *[max_time, batch_size, ...]*. If *False* (default), *sequence* must have shape *[batch_size, max_time, ...]*.
- **dtype** (*dtype*) – Type of `sequence`. If *None*, infer from `sequence` automatically.

Returns A Tensor containing the masked and reduced sequence.

reduce_batch_time

texar.losses.**reduce_batch_time**(*sequence*, *sequence_length*, *average_across_batch=True*, *average_across_timesteps=False*, *sum_over_batch=False*, *sum_over_timesteps=True*)

Average or sum over the respective dimensions of `sequence`, which is of shape *[batch_size, max_time, ...]*.

Assumes `sequence` has been properly masked according to `sequence_length`.

reduce_dimensions

texar.losses.**reduce_dimensions**(*tensor*, *average_axes=None*, *sum_axes=None*, *keep-dims=None*)

Average or sum over dimensions of `tensor`.

`average_axes` and `sum_axes` must be mutually exclusive. That is, elements in *average_axes* must not be contained in *sum_axes*, and vice versa.

Parameters

- **tensor** – A tensor to reduce.

- **average_axes** (*optional*) – A (list of) *int* that indicates the dimensions to reduce by taking average.
- **sum_axes** (*optional*) – A (list of) *int* that indicates the dimensions to reduce by taking sum.
- **keepdims** (*optional*) – If *True*, retains reduced dimensions with length 1.

3.7 Evaluations

3.7.1 BLEU

sentence_bleu

texar.evals.**sentence_bleu**(*references*, *hypothesis*, *max_order=4*, *lowercase=False*, *smooth=False*, *return_all=False*)

Calculates BLEU score of a hypothesis sentence.

Parameters

- **references** – A list of reference for the hypothesis. Each reference can be either a list of string tokens, or a string containing tokenized tokens separated with whitespaces. List can also be numpy array.
- **hypotheses** – A hypothesis sentence. Each hypothesis can be either a list of string tokens, or a string containing tokenized tokens separated with whitespaces. List can also be numpy array.
- **lowercase** (*bool*) – If *True*, pass the "-lc" flag to the multi-bleu script.
- **max_order** (*int*) – Maximum n-gram order to use when computing BLEU score.
- **smooth** (*bool*) – Whether or not to apply (Lin et al. 2004) smoothing.
- **return_all** (*bool*) – If *True*, returns BLEU and all n-gram precisions.

Returns

If `return_all` is *False* (default), returns a float32 BLEU score.

If `return_all` is *True*, returns a list of float32 scores: *[BLEU] + n-gram precisions*, which is of length :attr:‘max_order‘+1.

corpus_bleu

texar.evals.**corpus_bleu**(*list_of_references*, *hypotheses*, *max_order=4*, *lowercase=False*, *smooth=False*, *return_all=True*)

Computes corpus-level BLEU score.

Parameters

- **list_of_references** – A list of lists of references for each hypothesis. Each reference can be either a list of string tokens, or a string containing tokenized tokens separated with whitespaces. List can also be numpy array.
- **hypotheses** – A list of hypothesis sentences. Each hypothesis can be either a list of string tokens, or a string containing tokenized tokens separated with whitespaces. List can also be numpy array.
- **lowercase** (*bool*) – If *True*, lowercase reference and hypothesis tokens.

- **max_order** (*int*) – Maximum n-gram order to use when computing BLEU score.
- **smooth** (*bool*) – Whether or not to apply (Lin et al. 2004) smoothing.
- **return_all** (*bool*) – If *True*, returns BLEU and all n-gram precisions.

Returns

If `return_all` is *False* (default), returns a float32 BLEU score.

If `return_all` is *True*, returns a list of float32 scores: *[BLEU] + n-gram precisions*, which is of length :attr:‘max_order‘+1.

sentence_bleu_moses

`texar.evals.`**`sentence_bleu_moses`**(*references*, *hypothesis*, *lowercase=False*, *return_all=False*)

Calculates BLEU score of a hypothesis sentence using the **MOSES multi-bleu.perl** script.

Parameters

- **references** – A list of reference for the hypothesis. Each reference can be either a string, or a list of string tokens. List can also be numpy array.
- **hypotheses** – A hypothesis sentence. The hypothesis can be either a string, or a list of string tokens. List can also be numpy array.
- **lowercase** (*bool*) – If *True*, pass the “-lc” flag to the multi-bleu script.
- **return_all** (*bool*) – If *True*, returns BLEU and all n-gram precisions.

Returns

If `return_all` is *False* (default), returns a float32 BLEU score.

If `return_all` is *True*, returns a list of 5 float32 scores: *[BLEU, 1-gram precision, . . . , 4-gram precision]*.

corpus_bleu_moses

`texar.evals.`**`corpus_bleu_moses`**(*list_of_references*, *hypotheses*, *lowercase=False*, *return_all=False*)

Calculates corpus-level BLEU score using the **MOSES multi-bleu.perl** script.

Parameters

- **list_of_references** – A list of lists of references for each hypothesis. Each reference can be either a string, or a list of string tokens. List can also be numpy array.
- **hypotheses** – A list of hypothesis sentences. Each hyperthsis can be either a string, or a list of string tokens. List can also be numpy array.
- **lowercase** (*bool*) – If *True*, pass the “-lc” flag to the multi-bleu script.
- **return_all** (*bool*) – If *True*, returns BLEU and all n-gram precisions.

Returns

If `return_all` is *False* (default), returns a float32 BLEU score.

If `return_all` is *True*, returns a list of 5 float32 scores: *[BLEU, 1-gram precision, . . . , 4-gram precision]*.

3.7.2 Accuracy

accuracy

texar.evals.**accuracy**(*labels*, *preds*)
Calculates the accuracy of predictions.

Parameters

- **labels** – The ground truth values. A Tensor of the same shape of `preds`.
- **preds** – A Tensor of any shape containing the predicted values.

Returns A float scalar Tensor containing the accuracy.

binary_clas_accurac

texar.evals.**binary_clas_accuracy**(*pos_preds=None*, *neg_preds=None*)
Calculates the accuracy of binary predictions.

Parameters

- **pos_preds** (*optional*) – A Tensor of any shape containing the predicted values on positive data (i.e., ground truth labels are *1*).
- **neg_preds** (*optional*) – A Tensor of any shape containing the predicted values on negative data (i.e., ground truth labels are *0*).

Returns A float scalar Tensor containing the accuracy.

3.8 Models

3.8.1 ModelBase

class texar.models.**ModelBase**(*hparams=None*)
Base class inherited by all model classes.

A model class implements interfaces that are compatible with TF Estimator. In particular, *_build()* implements the model_fn interface; and *get_input_fn()* is for the `input_fn` interface.

_build(*features*, *labels*, *params*, *mode*, *config=None*)
Used for the model_fn argument when constructing tf.estimator.Estimator.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

get_input_fn(**args*, ***kwargs*)
Returns the `input_fn` function that constructs the input data, used in tf.estimator.Estimator.

hparams
A *HParams* instance. The hyperparameters of the module.

3.8.2 Seq2seqBase

class texar.models.**Seq2seqBase**(*data_hparams*, *hparams=None*)
Base class inherited by all seq2seq model classes.

_build(*features*, *labels*, *params*, *mode*, *config=None*)
Used for the model_fn argument when constructing tf.estimator.Estimator.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

```
{
    "source_embedder": "WordEmbedder",
    "source_embedder_hparams": {},
    "target_embedder": "WordEmbedder",
    "target_embedder_hparams": {},
    "embedder_share": True,
    "embedder_hparams_share": True,
    "encoder": "UnidirectionalRNNEncoder",
    "encoder_hparams": {},
    "decoder": "BasicRNNDecoder",
    "decoder_hparams": {},
    "decoding_strategy_train": "train_greedy",
    "decoding_strategy_infer": "infer_greedy",
    "beam_search_width": 0,
    "connector": "MLPTransformConnector",
    "connector_hparams": {},
    "optimization": {},
    "name": "seq2seq",
}
```

Here:

"source_embedder" [str or class or instance] Word embedder for source text. Can be a class, its name or module path, or a class instance.

"source_embedder_hparams" [dict] Hyperparameters for constructing the source embedder. E.g., See *default_hparams()* for hyperparameters of *WordEmbedder*. Ignored if "source_embedder" is an instance.

"target_embedder", "target_embedder_hparams" : Same as "source_embedder" and "source_embedder_hparams" but for target text embedder.

"embedder_share" [bool] Whether to share the source and target embedder. If *True*, source embedder will be used to embed target text.

"embedder_hparams_share" [bool] Whether to share the embedder configurations. If *True*, target embedder will be created with "source_embedder_hparams". But the two embedders have different set of trainable variables.

"encoder", "encoder_hparams" : Same as "source_embedder" and "source_embedder_hparams" but for encoder.

"decoder", "decoder_hparams" : Same as "source_embedder" and "source_embedder_hparams" but for decoder.

"decoding_strategy_train" [str] The decoding strategy in training mode. See *_build()* for details.

"decoding_strategy_infer" [str] The decoding strategy in eval/inference mode.

"beam_search_width" [int] Beam width. If > 1, beam search is used in eval/inference mode.

"connector", "connector_hparams" : The connector class and hyperparameters. A connector transforms an encoder final state to a decoder initial state.

"optimization" [dict] Hyperparameters of optimizating the model. See *default_optimization_hparams()* for details.

"name" [str] Name of the model.

get_loss(*decoder_results*, *features*, *labels*)
Computes the training loss.

embed_source(*features*, *labels*, *mode*)
Embeds the inputs.

embed_target(*features*, *labels*, *mode*)
Embeds the target inputs. Used in training.

encode(*features*, *labels*, *mode*)
Encodes the inputs.

decode(*encoder_results*, *features*, *labels*, *mode*)
Decodes.

get_input_fn(*mode*, *hparams=None*)
Creates an input function *input_fn* that provides input data for the model in an Estimator. See, e.g., tf.estimator.train_and_evaluate.

Parameters

- **mode** – One of members in tf.estimator.ModeKeys.
- **hparams** – A *dict* or an *HParams* instance containing the hyperparameters of *PairedTextData*. See *default_hparams()* for the the structure and default values of the hyperparameters.

Returns An input function that returns a tuple *(features, labels)* when called. *features* contains data fields that are related to source text, and *labels* contains data fields related to target text. See *PairedTextData* for all data fields.

hparams
A *HParams* instance. The hyperparameters of the module.

3.8.3 BasicSeq2seq

class texar.models.**BasicSeq2seq**(*data_hparams*, *hparams=None*)
The basic seq2seq model (without attention).

Example

```
model = BasicSeq2seq(data_hparams, model_hparams)
exor = tx.run.Executor(
    model=model,
    data_hparams=data_hparams,
    config=run_config)
exor.train_and_evaluate(
    max_train_steps=10000,
    eval_steps=100)
```

_build(*features*, *labels*, *params*, *mode*, *config=None*)
Used for the model_fn argument when constructing tf.estimator.Estimator.

static default_hparams()
Returns a dictionary of hyperparameters with default values.

Same as *default_hparams()* of *Seq2seqBase*.

embed_source (*features*, *labels*, *mode*)
Embeds the inputs.

embed_target (*features*, *labels*, *mode*)
Embeds the target inputs. Used in training.

encode (*features*, *labels*, *mode*)
Encodes the inputs.

decode (*encoder_results*, *features*, *labels*, *mode*)
Decodes.

get_input_fn (*mode*, *hparams=None*)
Creates an input function *input_fn* that provides input data for the model in an Estimator. See, e.g., tf.estimator.train_and_evaluate.

Parameters

- **mode** – One of members in tf.estimator.ModeKeys.
- **hparams** – A *dict* or an *HParams* instance containing the hyperparameters of *PairedTextData*. See *default_hparams()* for the the structure and default values of the hyperparameters.

Returns An input function that returns a tuple *(features, labels)* when called. *features* contains data fields that are related to source text, and *labels* contains data fields related to target text. See *PairedTextData* for all data fields.

get_loss (*decoder_results*, *features*, *labels*)
Computes the training loss.

hparams
A *HParams* instance. The hyperparameters of the module.

3.9 Executor

class texar.run.**Executor** (*model*, *data_hparams*, *config*, *model_hparams=None*, *train_hooks=None*, *eval_hooks=None*, *session_config=None*)
Class that executes training, evaluation, prediction, export, and other actions of Estimator.

Parameters

- **model** – An instance of a subclass of ModelBase.
- **data_hparams** – A *dict* or an instance of HParams containing the hyperparameters of data. It must contain *train* and/or *eval* fields for relevant processes. For example, for *train_and_evaluate()*, both fields are required.
- **config** – An instance of tf.estimator.RunConfig, used as the config argument of Estimator.
- **model_hparams** (*optional*) – A *dict* or an instance of HParams containing the hyperparameters of the model. If *None*, uses model.hparams. Used as the params argument of Estimator.
- **train_hooks** (*optional*) – Iterable of tf.train.SessionRunHook objects to run during training.
- **eval_hooks** (*optional*) – Iterable of tf.train.SessionRunHook objects to run during evaluation.

- **session_config** (*optional*) – An instance of tf.ConfigProto, used as the `config` argument of tf session.

Example

```
model = BasicSeq2seq(data_hparams, model_hparams)
exor = Executor(
    model=model,
    data_hparams=data_hparams,
    config=run_config)
exor.train_and_evaluate(
    max_train_steps=10000,
    eval_steps=100)
```

See *bin/train.py* for the usage in detail.

train(*max_steps=None*)
Trains the model. See tf.estimator.Estimator.train for more details.

Parameters **max_steps** (*int, optional*) – Total number of steps for which to train model. If *None*, train forever or until the train data generates the OutOfRange exception. If OutOfRange occurs in the middle, training stops before `max_steps` steps.

evaluate(*steps=None*, *checkpoint_path=None*)
Evaluates the model. See tf.estimator.Estimator.evaluate for more details.

Parameters

- **steps** (*int, optional*) – Number of steps for which to evaluate model. If *None*, evaluates until the eval data raises an OutOfRange exception.
- **checkpoint_path** (*str, optional*) – Path of a specific checkpoint to evaluate. If *None*, the the latest checkpoint in `config.model_dir` is used. If there are no checkpoints in `model_dir`, evaluation is run with newly initialized variables instead of restored from checkpoint.

train_and_evaluate(*max_train_steps=None*, *eval_steps=None*)
Trains and evaluates the model. See tf.estimator.train_and_evaluate for more details.

Parameters

- **max_train_steps** (*int, optional*) – Total number of steps for which to train model. If *None*, train forever or until the train data generates the OutOfRange exception. If OutOfRange occurs in the middle, training stops before `max_steps` steps.
- **eval_steps** (*int, optional*) – Number of steps for which to evaluate model. If *None*, evaluates until the eval data raises an OutOfRange exception.

3.10 Context

3.10.1 Global Mode

global_mode

texar.**global_mode**()
Returns the Tensor of global mode.

This is a placeholder with default value of tf.estimator.ModeKeys.TRAIN.

Example

```
mode = session.run(global_mode())
# mode == tf.estimator.ModeKeys.TRAIN

mode = session.run(
    global_mode(),
    feed_dict={tf.global_mode(): tf.estimator.ModeKeys.PREDICT})
# mode == tf.estimator.ModeKeys.PREDICT
```

global_mode_train

texar.**global_mode_train**()
Returns a bool Tensor indicating whether the global mode is TRAIN.

Example

```
is_train = session.run(global_mode_train())
# is_train == True

is_train = session.run(
    global_mode_train()
    feed_dict={tf.global_mode(): tf.estimator.ModeKeys.PREDICT})
# is_train == False
```

global_mode_eval

texar.**global_mode_eval**()
Returns a bool Tensor indicating whether the global mode is EVAL.

global_mode_predict

texar.**global_mode_predict**()
Returns a bool Tensor indicating whether the global mode is PREDICT.

valid_modes

texar.**valid_modes**()
Returns a set of possible values of mode.

3.11 Utils

3.11.1 Frequent Use

AverageRecorder

class texar.utils.**AverageRecorder**(*size=None*)

Maintains the moving averages (i.e., the average of the latest N records) of (possibly multiple) fields.

Fields are determined by the first call of `add()`.

Parameters **size** (*int, optional*) – The window size of moving average. If *None*, the average of all added records is maintained.

Example

```
## Use to maintain moving average of training loss
avg_rec = AverageRecorder(size=10) # average over latest 10 records
while training:
    loss_0, loss_1  = ...
    avg_rec.add([loss_0, loss_1])
    # avg_rec.avg() == [0.12343452, 0.567800323]
    # avg_rec.avg(0) == 0.12343452
    # avg_rec.to_str(precision=2, ) == '0.12 0.57'

## Use to maintain average of test metrics on the whole test set
avg_rec = AverageRecorder() # average over ALL records
while test:
    metric_0, metric_1  = ...
    avg_rec.add({'m0': metric_0, 'm1': metric_1}) # dict is allowed
print(avg_rec.to_str(precision=4, delimiter=' , '))
# 'm0: 0.1234 , m1: 0.5678'
#
# avg_rec.avg() == {'m0': 0.12343452, 'm1': 0.567800323}
# avg_rec.avg(0) == 0.12343452
```

add(*record*, *weight=None*)

Appends a new record.

`record` can be a *list*, *dict*, or a single scalar. The record type is determined at the first time `add()` is called. All subsequent calls to `add()` must have the same type of `record`.

`record` in subsequent calls to `add()` can contain only a subset of fields than the first call to `add()`.

Example

```
recorder.add({'1': 0.2, '2': 0.2}) # 1st call to `add`
x = recorder.add({'1': 0.4}) # 2nd call to `add`
# x == {'1': 0.3, '2': 0.2}
```

Parameters

- **record** – A single scalar, a list of scalars, or a dict of scalars.

- **weight** (*optional*) – A scalar, weight of the new record for calculating a weighted average. If *None*, weight is set to *1*. For example, `weight` can be set to batch size and `record` the average value of certain metrics on the batch in order to calculate the average metric values on a whole dataset.

Returns The (moving) average after appending the record, with the same type as `record`.

avg(*id_or_name=None*)
Returns the (moving) average.

Parameters **id_or_name** (*optional*) – A list of or a single element. Each element is the index (if the record type is *list*) or name (if the record type is *dict*) of the field for which the average is calculated. If not given, the average of all fields are returned.

Returns The average value(s). If `id_or_name` is a single element (not a list), then returns the average value of the corresponding field. Otherwise, if `id_or_name` is a list of element(s), then returns average value(s) in the same type as `record` of *add()*.

reset(*id_or_name=None*)
Resets the record.

Parameters **id_or_name** (*optional*) – A list or a single element. Each element is the index (if the record type is *list*) or name (if the record type is *dict*) of the field to reset. If *None*, all fields are reset.

to_str(*precision=None*, *delimiter=' '*)
Returns a string of the average values of the records.

Parameters

- **precision** (*int, optional*) – The number of decimal places to keep in the returned string. E.g., for an average value of *0.1234*, `precision = 2` leads to *'0.12'*.
- **delimiter** (*str*) – The delimiter string that separates between fields.

Returns

A string of the average values.

If record is of type *dict*, the string is a concatenation of 'field_name: average_value', delimited with `delimiter`. E.g., *'field_name_1: 0.1234 field_name_2: 0.5678 ... '*.

Otherwise, the string is of a concatenation of 'average_value'. E.g., *'0.1234 0.5678 ... '*

collect_trainable_variables

texar.utils.**collect_trainable_variables**(*modules*)
Collects all trainable variables of modules.

Trainable variables included in multiple modules occur only once in the returned list.

Parameters **modules** – A (list of) instance of the subclasses of `ModuleBase`.

Returns A list of trainable variables in the modules.

compat_as_text

texar.utils.**compat_as_text**(*str_*)
Converts strings into *unicode* (Python 2) or *str* (Python 3).

Parameters **str_** – A string or other data types convertible to string, or an *n*-D numpy array or (possibly nested) list of such elements.

Returns The converted strings of the same structure/shape as `str_`.

map_ids_to_strs

`texar.utils.`**map_ids_to_strs**(*ids*, *vocab*, *join=True*, *strip_pad='<PAD>'*, *strip_bos='<BOS>'*, *strip_eos='<EOS>'*, *compat=True*)

Transforms *int* indexes to strings by mapping ids to tokens, concatenating tokens into sentences, and stripping special tokens, etc.

Parameters

- **ids** – An n-D numpy array or (possibly nested) list of *int* indexes.
- **vocab** – An instance of *Vocab*.
- **join** (*bool*) – Whether to concat along the last dimension of the the tokens into a string separated with a space character.
- **strip_pad** (*str*) – The PAD token to strip from the strings (i.e., remove the leading and trailing PAD tokens of the strings). Default is '<PAD>' as defined in *SpecialTokens*.PAD. Set to *None* or *False* to disable the stripping.
- **strip_bos** (*str*) – The BOS token to strip from the strings (i.e., remove the leading BOS tokens of the strings). Default is '<BOS>' as defined in *SpecialTokens*.BOS. Set to *None* or *False* to disable the stripping.
- **strip_eos** (*str*) – The EOS token to strip from the strings (i.e., remove the EOS tokens and all subsequent tokens of the strings). Default is '<EOS>' as defined in *SpecialTokens*.EOS. Set to *None* or *False* to disable the stripping.

Returns If `join` is True, returns a *(n-1)*-D numpy array (or list) of concatenated strings. If `join` is False, returns an *n*-D numpy array (or list) of str tokens.

Example

```
text_ids = [[1, 9, 6, 2, 0, 0], [1, 28, 7, 8, 2, 0]]

text = map_ids_to_strs(text_ids, data.vocab)
# text == ['a sentence', 'parsed from ids']

text = map_ids_to_strs(
    text_ids, data.vocab, join=False,
    strip_pad=None, strip_bos=None, strip_eos=None)
# text == [['<BOS>', 'a', 'sentence', '<EOS>', '<PAD>', '<PAD>'],
#          ['<BOS>', 'parsed', 'from', 'ids', '<EOS>', '<PAD>']]
```

write_paired_text

`texar.utils.`**write_paired_text**(*src*, *tgt*, *fname*, *append=False*, *mode='h'*, *sep='\t'*)

Writes paired text to a file.

Parameters

- **src** – A list (or array) of *str* source text.

- **tgt** – A list (or array) of *str* target text.
- **fname** (*str*) – The output filename.
- **append** (*bool*) – Whether append content to the end of the file if exists.
- **mode** (*str*) – The mode of writing, with the following options:
  - **'h'**: The "horizontal" mode. Each source target pair is written in one line, intervened with `sep`, e.g.:

```
source_1 target_1
source_2 target_2
```

  - **'v'**: The "vertical" mode. Each source target pair is written in two consecutive lines, e.g:

```
source_1
target_1
source_2
target_2
```

  - **'s'**: The "separate" mode. Each source target pair is written in corresponding lines of two files named as *"{fname}.src"* and *"fname.tgt"*, respectively.
- **sep** (*str*) – The string intervening between source and target. Used when `mode` is set to 'h'.

Returns The fileanme(s). If *mode* == 'h' or 'v', returns `fname`. If *mode* == 's', returns a list of filenames *["{fname}.src", "{fname}.tgt"]*.

straight_through

texar.utils.**straight_through**(*fw_tensor*, *bw_tensor*)

Use a tensor in forward pass while backpropagating gradient to another.

Parameters

- **fw_tensor** – A tensor to be used in the forward pass.
- **bw_tensor** – A tensor to which gradient is backpropagated. Must have the same shape and type with `fw_tensor`.

Returns A tensor of the same shape and value with `fw_tensor` but will direct gradient to bw_tensor.

3.11.2 Variables

collect_trainable_variables

texar.utils.**collect_trainable_variables**(*modules*)

Collects all trainable variables of modules.

Trainable variables included in multiple modules occur only once in the returned list.

Parameters **modules** – A (list of) instance of the subclasses of `ModuleBase`.

Returns A list of trainable variables in the modules.

get_unique_named_variable_scope

texar.utils.**get_unique_named_variable_scope**(*base_name*)
Returns a variable scope with a unique name.

Parameters **base_name** (*str*) – The base name to uniquified.

Returns An instance of variable_scope.

Example

```
vs = get_unique_named_variable_scope('base_name')
with tf.variable_scope(vs):
    ....
```

add_variable

texar.utils.**add_variable**(*variable*, *var_list*)
Adds variable to a given list.

Parameters

- **variable** – A (list of) variable(s).
- **var_list** (*list*) – The list where the variable are added to.

3.11.3 IO

write_paired_text

texar.utils.**write_paired_text**(*src*, *tgt*, *fname*, *append=False*, *mode='h'*, *sep='\t'*)
Writes paired text to a file.

Parameters

- **src** – A list (or array) of *str* source text.
- **tgt** – A list (or array) of *str* target text.
- **fname** (*str*) – The output filename.
- **append** (*bool*) – Whether append content to the end of the file if exists.
- **mode** (*str*) – The mode of writing, with the following options:
  - **'h'**: The "horizontal" mode. Each source target pair is written in one line, intervened with sep, e.g.:

```
source_1 target_1
source_2 target_2
```

  - **'v'**: The "vertical" mode. Each source target pair is written in two consecutive lines, e.g:

```
source_1
target_1
source_2
target_2
```

- **'s'**: The "separate" mode. Each source target pair is written in corresponding lines of two files named as *"{fname}.src"* and *"fname.tgt"*, respectively.

- **sep** (*str*) – The string intervening between source and target. Used when `mode` is set to 'h'.

Returns The fileanme(s). If *mode* == 'h' or 'v', returns `fname`. If *mode* == 's', returns a list of filenames *["{fname}.src", "{fname}.tgt"]*.

load_config

`texar.utils.`**`load_config`**(*config_path*, *config=None*)

Loads configs from (possibly multiple) file(s).

A config file can be either a Python file (with suffix '.py') or a YAML file. If the filename is not suffixed with '.py', the file is parsed as YAML.

Parameters

- **config_path** – Paths to configuration files. This can be a *list* of config file names, or a path to a directory in which all files are loaded, or a string of multiple file names separated by commas.

- **config** (*dict, optional*) – A config dict to which new configurations are added. If *None*, a new config dict is created.

Returns A *dict* of configurations.

maybe_create_dir

`texar.utils.`**`maybe_create_dir`**(*dirname*)

Creates directory if doesn't exist

get_files

`texar.utils.`**`get_files`**(*file_paths*)

Gets a list of file paths given possibly a pattern `file_paths`.

Adapted from *tf.contrib.slim.data.parallel_reader.get_data_files*.

Parameters **file_paths** – A (list of) path to the files. The path can be a pattern, e.g., /path/to/train*, /path/to/train[12]

Returns A list of file paths.

Raises **ValueError** – If no files are not found

3.11.4 DType

compat_as_text

`texar.utils.`**`compat_as_text`**(*str_*)

Converts strings into *unicode* (Python 2) or *str* (Python 3).

Parameters **str_** – A string or other data types convertible to string, or an *n*-D numpy array or (possibly nested) list of such elements.

Returns The converted strings of the same structure/shape as `str_`.

get_tf_dtype

`texar.utils.`**`get_tf_dtype`**(*dtype*)
Returns equivalent tf dtype.

Parameters **`dtype`** – A str, python numeric or string type, numpy data type, or tf dtype.

Returns The corresponding tf dtype.

is_callable

`texar.utils.`**`is_callable`**(*x*)
Return *True* if `x` is callable.

is_str

`texar.utils.`**`is_str`**(*x*)
Returns *True* if `x` is either a str or unicode. Returns *False* otherwise.

is_placeholder

`texar.utils.`**`is_placeholder`**(*x*)
Returns *True* if `x` is a tf.placeholder or tf.placeholder_with_default.

maybe_hparams_to_dict

`texar.utils.`**`maybe_hparams_to_dict`**(*hparams*)
If `hparams` is an instance of *HParams*, converts it to a *dict* and returns. If `hparams` is a *dict*, returns as is.

3.11.5 Shape

mask_sequences

`texar.utils.`**`mask_sequences`**(*sequence*, *sequence_length*, *dtype=None*, *time_major=False*, *tensor_rank=2*)
Masks out sequence entries that are beyond the respective sequence lengths. Masks along the time dimension.

`sequence` and `sequence_length` can either be python arrays or Tensors, respectively. If both are python arrays (or None), the return will be a python array as well.

Parameters

- **`sequence`** – A Tensor or python array of sequence values. If *time_major==False* (default), this must be a Tensor of shape *[batch_size, max_time, ...]*. The batch and time dimension is exchanged if *time_major==True*.
- **`sequence_length`** – A Tensor or python array of shape *[batch_size]*. Time steps beyond the respective sequence lengths will be made zero.
- **`dtype`** (*dtype*) – Type of `sequence`. If *None*, infer from `sequence` automatically.

- **time_major** (*bool*) – The shape format of the inputs. If *True*, `sequence` must have shape *[max_time, batch_size, ...]*. If *False* (default), `sequence` must have shape *[batch_size, max_time, ...]*.
- **tensor_rank** (*int*) – The number of dimensions of `sequence`. Default is 2, i.e., `sequence` is a 2D Tensor consisting of batch and time dimensions. Ignored if both `sequence` and `sequence_length` are python arrays.

Returns

The masked sequence, i.e., a Tensor or python array of the same shape as `sequence` but with masked-out entries (set to zero).

If both `sequence` and `sequence_length` are python arrays, the returned value is a python array as well.

transpose_batch_time

`texar.utils.`**`transpose_batch_time`**(*inputs*)

Transposes inputs between time-major and batch-major.

Parameters **inputs** – A Tensor of shape *[batch_size, max_time, ...]* (batch-major) or *[max_time, batch_size, ...]* (time-major), or a (possibly nested) tuple of such elements.

Returns A (possibly nested tuple of) Tensor with transposed batch and time dimensions of inputs.

get_batch_size

`texar.utils.`**`get_batch_size`**(*tensor*)

Returns a unit *Tensor* representing the batch size, i.e., the size of the 1st dimension of `tensor`.

get_rank

`texar.utils.`**`get_rank`**(*tensor*)

Returns the tensor rank as a python *int*. The input tensor can also be a python array.

Parameters **tensor** – A Tensor or python array.

Returns A python *int* representing the rank of `tensor`. Returns *None* if the rank cannot be determined.

shape_list

`texar.utils.`**`shape_list`**(*x*)

Returns **static** shape of the input Tensor whenever possible.

Parameters **x** – A Tensor.

Returns *tf.shape(x)* - Otherwise, returns a list of dims, each of which is either an *int* whenever it can be statically determined, or a scalar Tensor otherwise.

Return type

- If the rank of `x` is unknown, returns the dynamic shape

pad_and_concat

texar.utils.**pad_and_concat**(*values*, *axis*, *rank=None*, *pad_axis=None*, *pad_constant_values=0*)

Concats tensors along one dimension. Pads each of other dimensions of the tensors to the corresponding maximum size if necessary.

Parameters

- **values** – A list of Tensors of the same rank.
- **axis** (*int*) – A Python int. Dimension along which to concatenate.
- **rank** (*int, optional*) – Rank of the tensors. If *None*, inferred automatically from `values`.
- **pad_axis** (*int or list, optional*) – A Python int or a list of int. Dimensions to pad. Paddings are only added to the end of corresponding dimensions. If *None*, all dimensions except the `axis` dimension are padded.
- **pad_constant_values** – The scalar pad value to use. Must be same type as the tensors.

Returns A *Tensor* resulting from padding and concatenation of the input tensors.

Raises **ValueError** – If `rank` is *None* and cannot be inferred from `values`.

Example

```
a = tf.ones([1, 2])
b = tf.ones([2, 3])

c = pad_and_concat([a,b], 0)
# c.shape == [3, 3]
# c == [[1, 1, 0],
#       [1, 1, 1],
#       [1, 1, 1]]

d = pad_and_concat([a,b], 1)
# d.shape == [2, 5]
# d == [[1, 1, 1, 1, 1]
#       [0, 0, 1, 1, 1]]
```

flatten

texar.utils.**flatten**(*tensor*, *preserve_dims*, *flattened_dim=None*)

Flattens a tensor whiling keeping several leading dimensions.

`preserve_dims` must < tensor's rank

Parameters

- **tensor** – A Tensor to flatten.
- **preserve_dims** (*int*) – The number of leading dimensions to preserve.
- **flatterned_dim** (*int, optional*) – The size of the resulting flattened dimension. If not given, infer automatically, which can cause a statically unknown dimension size.

Returns A Tensor with rank :attr:‘perserve_dims‘+1.

Example

```
x = tf.ones(shape=[d_1, d_2, d_3, d_4])
y = flatten(x, 2) # y.shape == [d_1, d_2, d_3 * d_4]
```

3.11.6 Dictionary

dict_patch

texar.utils.**dict_patch**(*tgt_dict*, *src_dict*)

Recursively patch `tgt_dict` by adding items from `src_dict` that do not exist in `tgt_dict`.

If respective items in `src_dict` and `tgt_dict` are both *dict*, the `tgt_dict` item is patched recursively.

Parameters

- **tgt_dict** (*dict*) – Target dictionary to patch.
- **src_dict** (*dict*) – Source dictionary.

Returns The new `tgt_dict` that is patched.

Return type dict

dict_lookup

texar.utils.**dict_lookup**(*dict_*, *keys*, *default=None*)

Looks up `keys` in the dict, returns the corresponding values.

The `default` is used for keys not present in the dict.

Parameters

- **dict** (*dict*) – A dictionary for lookup.
- **keys** – A numpy array or a (possibly nested) list of keys.
- **default** (*optional*) – Value to be returned when a key is not in `dict_`. Error is raised if `default` is not given and key is not in the dict.

Returns A numpy array of values with the same structure as `keys`.

Raises **TypeError** – If key is not in `dict_` and `default` is *None*.

dict_fetch

texar.utils.**dict_fetch**(*src_dict*, *tgt_dict_or_keys*)

Fetches a sub dict of `src_dict` with the keys in `tgt_dict_or_keys`.

Parameters

- **src_dict** – A dict or instance of *HParams*. The source dict to fetch values from.
- **tgt_dict_or_keys** – A dict, instance of *HParams*, or a list (or a dict_keys) of keys to be included in the output dict.

Returns A new dict that is a subdict of `src_dict`.

dict_pop

texar.utils.**dict_pop**(*dict_*, *pop_keys*, *default=None*)

Removes keys from a dict and returns their values.

Parameters

- **dict** (*dict*) – A dictionary from which items are removed.
- **pop_keys** – A key or a list of keys to remove and return respective values or `default`.
- **default** (*optional*) – Value to be returned when a key is not in `dict_`. The default value is *None*.

Returns A *dict* of the items removed from `dict_`.

flatten_dict

texar.utils.**flatten_dict**(*dict_*, *parent_key=''*, *sep='.'*)

Flattens a nested dictionary. Namedtuples within the dictionary are converted to dicts.

Adapted from: https://github.com/google/seq2seq/blob/master/seq2seq/models/model_base.py

Parameters

- **dict** (*dict*) – The dictionary to flatten.
- **parent_key** (*str*) – A prefix to prepend to each key.
- **sep** (*str*) – Separator that intervenes between parent and child keys. E.g., if *sep* == '.', then *{ "a": { "b": 3 } }* is converted into *{ "a.b": 3 }*.

Returns A new flattened *dict*.

3.11.7 String

map_ids_to_strs

texar.utils.**map_ids_to_strs**(*ids*, *vocab*, *join=True*, *strip_pad='<PAD>'*, *strip_bos='<BOS>'*, *strip_eos='<EOS>'*, *compat=True*)

Transforms *int* indexes to strings by mapping ids to tokens, concatenating tokens into sentences, and stripping special tokens, etc.

Parameters

- **ids** – An n-D numpy array or (possibly nested) list of *int* indexes.
- **vocab** – An instance of *Vocab*.
- **join** (*bool*) – Whether to concat along the last dimension of the the tokens into a string separated with a space character.
- **strip_pad** (*str*) – The PAD token to strip from the strings (i.e., remove the leading and trailing PAD tokens of the strings). Default is '<PAD>' as defined in *SpecialTokens*.PAD. Set to *None* or *False* to disable the stripping.
- **strip_bos** (*str*) – The BOS token to strip from the strings (i.e., remove the leading BOS tokens of the strings). Default is '<BOS>' as defined in *SpecialTokens*.BOS. Set to *None* or *False* to disable the stripping.

- **strip_eos** (*str*) – The EOS token to strip from the strings (i.e., remove the EOS tokens and all subsequent tokens of the strings). Default is '<EOS>' as defined in *SpecialTokens*.EOS. Set to *None* or *False* to disable the stripping.

Returns If `join` is True, returns a *(n-1)*-D numpy array (or list) of concatenated strings. If `join` is False, returns an *n*-D numpy array (or list) of str tokens.

Example

```
text_ids = [[1, 9, 6, 2, 0, 0], [1, 28, 7, 8, 2, 0]]

text = map_ids_to_strs(text_ids, data.vocab)
# text == ['a sentence', 'parsed from ids']

text = map_ids_to_strs(
    text_ids, data.vocab, join=False,
    strip_pad=None, strip_bos=None, strip_eos=None)
# text == [['<BOS>', 'a', 'sentence', '<EOS>', '<PAD>', '<PAD>'],
#          ['<BOS>', 'parsed', 'from', 'ids', '<EOS>', '<PAD>']]
```

strip_token

texar.utils.**strip_token**(*str_*, *token*, *is_token_list=False*, *compat=True*)

Returns a copy of strings with leading and trailing tokens removed.

Note that besides `token`, all leading and trailing whitespace characters are also removed.

If `is_token_list` is False, then the function assumes tokens in `str_` are separated with whitespace character.

Parameters

- **str_** – A *str*, or an *n*-D numpy array or (possibly nested) list of *str*.
- **token** (*str*) – The token to strip, e.g., the '<PAD>' token defined in *SpecialTokens*.PAD
- **is_token_list** (*bool*) – Whether each sentence in `str_` is a list of tokens. If False, each sentence in `str_` is assumed to contain tokens separated with space character.
- **compat** (*bool*) – Whether to convert tokens into *unicode* (Python 2) or *str* (Python 3).

Returns The stripped strings of the same structure/shape as `str_`.

Example

```
str_ = '<PAD> a sentence <PAD> <PAD>  '
str_stripped = strip_token(str_, '<PAD>')
# str_stripped == 'a sentence'

str_ = ['<PAD>', 'a', 'sentence', '<PAD>', '<PAD>', '', '']
str_stripped = strip_token(str_, '<PAD>', is_token_list=True)
# str_stripped == 'a sentence'
```

strip_eos

texar.utils.**strip_eos**(*str_*, *eos_token='<EOS>'*, *is_token_list=False*, *compat=True*)

Remove the EOS token and all subsequent tokens.

If `is_token_list` is False, then the function assumes tokens in `str_` are separated with whitespace character.

Parameters

- **str_** – A *str*, or an *n*-D numpy array or (possibly nested) list of *str*.
- **eos_token** (*str*) – The EOS token. Default is '<EOS>' as defined in *SpecialTokens*.EOS
- **is_token_list** (*bool*) – Whether each sentence in `str_` is a list of tokens. If False, each sentence in `str_` is assumed to contain tokens separated with space character.
- **compat** (*bool*) – Whether to convert tokens into *unicode* (Python 2) or *str* (Python 3).

Returns Strings of the same structure/shape as `str_`.

strip_special_tokens

texar.utils.**strip_special_tokens**(*str_*, *strip_pad='<PAD>'*, *strip_bos='<BOS>'*, *strip_eos='<EOS>'*, *is_token_list=False*, *compat=True*)

Removes special tokens in strings, including:

- Removes EOS and all subsequent tokens
- Removes leading and and trailing PAD tokens
- Removes leading BOS tokens

Note that besides the special tokens, all leading and trailing whitespace characters are also removed.

This is a joint function of *strip_eos()*, `strip_pad()`, and `strip_bos()`

Parameters

- **str_** – A *str*, or an *n*-D numpy array or (possibly nested) list of *str*.
- **strip_pad** (*str*) – The PAD token to strip from the strings (i.e., remove the leading and trailing PAD tokens of the strings). Default is '<PAD>' as defined in *SpecialTokens*.PAD. Set to *None* or *False* to disable the stripping.
- **strip_bos** (*str*) – The BOS token to strip from the strings (i.e., remove the leading BOS tokens of the strings). Default is '<BOS>' as defined in *SpecialTokens*.BOS. Set to *None* or *False* to disable the stripping.
- **strip_eos** (*str*) – The EOS token to strip from the strings (i.e., remove the EOS tokens and all subsequent tokens of the strings). Default is '<EOS>' as defined in *SpecialTokens*.EOS. Set to *None* or *False* to disable the stripping.
- **is_token_list** (*bool*) – Whether each sentence in `str_` is a list of tokens. If False, each sentence in `str_` is assumed to contain tokens separated with space character.
- **compat** (*bool*) – Whether to convert tokens into *unicode* (Python 2) or *str* (Python 3).

Returns Strings of the same shape of `str_` with special tokens stripped.

str_join

texar.utils.**str_join**(*tokens*, *sep=' '*, *compat=True*)
Concats `tokens` along the last dimension with intervening occurrences of `sep`.

Parameters

- **tokens** – An *n*-D numpy array or (possibly nested) list of *str*.
- **sep** (*str*) – The string intervening between the tokens.
- **compat** (*bool*) – Whether to convert tokens into *unicode* (Python 2) or *str* (Python 3).

Returns An *(n-1)*-D numpy array (or list) of *str*.

default_str

texar.utils.**default_str**(*str_*, *default_str*)
Returns `str_` if it is not *None* or empty, otherwise returns *default_str*.

Parameters

- **str** – A string.
- **default_str** – A string.

Returns Either `str_` or *default_str*.

uniquify_str

texar.utils.**uniquify_str**(*str_*, *str_set*)
Uniquifies `str_` if `str_` is included in `str_set`.

This is done by appending a number to `str_`. Returns `str_` directly if it is not included in `str_set`.

Parameters

- **str** (*string*) – A string to uniquify.
- **str_set** (*set, dict, or list*) – A collection of strings. The returned string is guaranteed to be different from the elements in the collection.

Returns The uniquified string. Returns `str_` directly if it is already unique.

Example

```
print(uniquify_str('name', ['name', 'name_1']))
# 'name_2'
```

3.11.8 Meta

check_or_get_class

texar.utils.**check_or_get_class**(*class_or_name*, *module_path=None*, *superclass=None*)
Returns the class and checks if the class inherits `superclass`.

Parameters

- **class_or_name** – Name or full path to the class, or the class itself.
- **module_paths** (*list, optional*) – Paths to candidate modules to search for the class. This is used if `class_or_name` is a string and the class cannot be located solely based on `class_or_name`. The first module in the list that contains the class is used.
- **superclass** (*optional*) – A (list of) classes that the target class must inherit.

Returns The target class.

Raises

- **ValueError** – If class is not found based on `class_or_name` and `module_paths`.
- **TypeError** – If class does not inherits `superclass`.

get_class

`texar.utils.`**get_class**(*class_name*, *module_paths=None*)

Returns the class based on class name.

Parameters

- **class_name** (*str*) – Name or full path to the class.
- **module_paths** (*list*) – Paths to candidate modules to search for the class. This is used if the class cannot be located solely based on *class_name*. The first module in the list that contains the class is used.

Returns The target class.

Raises **ValueError** – If class is not found based on `class_name` and `module_paths`.

check_or_get_instance

`texar.utils.`**check_or_get_instance**(*ins_or_class_or_name*, *kwargs*, *module_paths=None*, *classtype=None*)

Returns a class instance and checks types.

Parameters

- **ins_or_class_or_name** – Can be of 3 types:
  - A class to instantiate.
  - A string of the name or full path to a class to instantiate.
  - The class instance to check types.
- **kwargs** (*dict*) – Keyword arguments for the class constructor. Ignored if *ins_or_class_or_name* is a class instance.
- **module_paths** (*list, optional*) – Paths to candidate modules to search for the class. This is used if the class cannot be located solely based on `class_name`. The first module in the list that contains the class is used.
- **classtype** (*optional*) – A (list of) class of which the instance must be an instantiation.

Raises

- **ValueError** – If class is not found based on `class_name` and `module_paths`.
- **ValueError** – If `kwargs` contains arguments that are invalid for the class construction.

- **TypeError** – If the instance is not an instantiation of `classtype`.

get_instance

texar.utils.**get_instance**(*class_or_name*, *kwargs*, *module_paths=None*)

Creates a class instance.

Parameters

- **class_or_name** – A class, or its name or full path to a class to instantiate.
- **kwargs** (*dict*) – Keyword arguments for the class constructor.
- **module_paths** (*list, optional*) – Paths to candidate modules to search for the class. This is used if the class cannot be located solely based on `class_name`. The first module in the list that contains the class is used.

Returns A class instance.

Raises

- **ValueError** – If class is not found based on `class_or_name` and `module_paths`.
- **ValueError** – If `kwargs` contains arguments that are invalid for the class construction.

check_or_get_instance_with_redundant_kwargs

texar.utils.**check_or_get_instance_with_redundant_kwargs**(*ins_or_class_or_name*, *kwargs*, *module_paths=None*, *classtype=None*)

Returns a class instance and checks types.

Only those keyword arguments in `kwargs` that are included in the class construction method are used.

Parameters

- **ins_or_class_or_name** – Can be of 3 types:
  - A class to instantiate.
  - A string of the name or module path to a class to instantiate.
  - The class instance to check types.
- **kwargs** (*dict*) – Keyword arguments for the class constructor.
- **module_paths** (*list, optional*) – Paths to candidate modules to search for the class. This is used if the class cannot be located solely based on `class_name`. The first module in the list that contains the class is used.
- **classtype** (*optional*) – A (list of) classes of which the instance must be an instantiation.

Raises

- **ValueError** – If class is not found based on `class_name` and `module_paths`.
- **ValueError** – If `kwargs` contains arguments that are invalid for the class construction.
- **TypeError** – If the instance is not an instantiation of `classtype`.

get_instance_with_redundant_kwargs

texar.utils.**get_instance_with_redundant_kwargs**(*class_name*, *kwargs*, *module_paths=None*)

Creates a class instance.

Only those keyword arguments in `kwargs` that are included in the class construction method are used.

Parameters

- **class_name** (*str*) – A class or its name or module path.
- **kwargs** (*dict*) – A dictionary of arguments for the class constructor. It may include invalid arguments which will be ignored.
- **module_paths** (*list of str*) – A list of paths to candidate modules to search for the class. This is used if the class cannot be located solely based on `class_name`. The first module in the list that contains the class is used.

Returns A class instance.

Raises **ValueError** – If class is not found based on `class_name` and `module_paths`.

get_function

texar.utils.**get_function**(*fn_or_name*, *module_paths=None*)

Returns the function of specified name and module.

Parameters

- **fn_or_name** (*str or callable*) – Name or full path to a function, or the function itself.
- **module_paths** (*list, optional*) – A list of paths to candidate modules to search for the function. This is used only when the function cannot be located solely based on `fn_or_name`. The first module in the list that contains the function is used.

Returns A function.

call_function_with_redundant_kwargs

texar.utils.**call_function_with_redundant_kwargs**(*fn*, *kwargs*)

Calls a function and returns the results.

Only those keyword arguments in `kwargs` that are included in the function's argument list are used to call the function.

Parameters

- **fn** (*function*) – A callable. If `fn` is not a python function, `fn.__call__` is called.
- **kwargs** (*dict*) – A *dict* of arguments for the callable. It may include invalid arguments which will be ignored.

Returns The returned results by calling `fn`.

get_args

texar.utils.**get_args**(*fn*)
Gets the arguments of a function.

Parameters **fn** (*callable*) – The function to inspect.

Returns A list of argument names (str) of the function.

Return type list

get_default_arg_values

texar.utils.**get_default_arg_values**(*fn*)
Gets the arguments and respective default values of a function.

Only arguments with default values are included in the output dictionary.

Parameters **fn** (*callable*) – The function to inspect.

Returns A dictionary that maps argument names (str) to their default values. The dictionary is empty if no arguments have default values.

Return type dict

get_instance_kwargs

texar.utils.**get_instance_kwargs**(*kwargs*, *hparams*)
Makes a dict of keyword arguments with the following structure:

*kwargs_ = {'hparams': dict(hparams), **kwargs}*.

This is typically used for constructing a module which takes a set of arguments as well as a argument named *hparams*.

Parameters

- **kwargs** (*dict*) – A dict of keyword arguments. Can be *None*.
- **hparams** – A dict or an instance of *HParams* Can be *None*.

Returns A *dict* that contains the keyword arguments in `kwargs`, and an additional keyword argument named *hparams*.

3.11.9 Mode

switch_dropout

texar.utils.**switch_dropout**(*dropout_keep_prob*, *mode=None*)
Turns off dropout when not in training mode.

Parameters

- **dropout_keep_prob** – Dropout keep probability in training mode
- **mode** (*optional*) – A Tensor taking values of tf.estimator.ModeKeys. Dropout is activated if `mode` is *TRAIN*. If *None*, the mode is inferred from *texar.global_mode()*.

Returns A unit Tensor that equals the dropout keep probability in *TRAIN* mode, and *1.0* in other modes.

maybe_global_mode

texar.utils.**maybe_global_mode**(*mode*)
Returns *texar.global_mode()* if mode is *None*, otherwise returns mode as-is.

is_train_mode

texar.utils.**is_train_mode**(*mode*)
Returns a bool Tensor indicating whether the global mode is TRAIN. If mode is *None*, the mode is determined by *texar.global_mode()*.

is_eval_mode

texar.utils.**is_eval_mode**(*mode*)
Returns a bool Tensor indicating whether the global mode is EVAL. If mode is *None*, the mode is determined by *texar.global_mode()*.

is_predict_mode

texar.utils.**is_predict_mode**(*mode*)
Returns a bool Tensor indicating whether the global mode is PREDICT. If mode is *None*, the mode is determined by *texar.global_mode()*.

is_train_mode_py

texar.utils.**is_train_mode_py**(*mode*, *default=True*)
Returns a python boolean indicating whether the mode is TRAIN.

Parameters

- **mode** – A string taking value in tf.estimator.ModeKeys. Can be *None*.
- **default** (*bool*) – The return value when mode is *None*. Default is *True*.

Returns A python boolean.

is_eval_mode_py

texar.utils.**is_eval_mode_py**(*mode*, *default=False*)
Returns a python boolean indicating whether the mode is EVAL.

Parameters

- **mode** – A string taking value in tf.estimator.ModeKeys. Can be *None*.
- **default** (*bool*) – The return value when mode is *None*. Default is *False*.

Returns A python boolean.

is_predict_mode_py

texar.utils.**is_predict_mode_py**(*mode*, *default=False*)
Returns a python boolean indicating whether the mode is PREDICT.

Parameters

- **mode** – A string taking value in tf.estimator.ModeKeys. Can be *None*.
- **default** (*bool*) – The return value when `mode` is *None*. Default is *False*.

Returns A python boolean.

3.11.10 Misc

ceildiv

texar.utils.**ceildiv**(*a*, *b*)
Divides with ceil.

E.g., *5 / 2 = 2.5*, *ceildiv(5, 2) = 3*.

Parameters

- **a** (*int*) – Dividend integer.
- **b** (*int*) – Divisor integer.

Returns Ceil quotient.

Return type int

straight_through

texar.utils.**straight_through**(*fw_tensor*, *bw_tensor*)
Use a tensor in forward pass while backpropagating gradient to another.

Parameters

- **fw_tensor** – A tensor to be used in the forward pass.
- **bw_tensor** – A tensor to which gradient is backpropagated. Must have the same shape and type with `fw_tensor`.

Returns A tensor of the same shape and value with `fw_tensor` but will direct gradient to bw_tensor.

3.11.11 AverageRecorder

class texar.utils.**AverageRecorder**(*size=None*)
Maintains the moving averages (i.e., the average of the latest N records) of (possibly multiple) fields.

Fields are determined by the first call of *add()*.

Parameters **size** (*int, optional*) – The window size of moving average. If *None*, the average of all added records is maintained.

Example

```
## Use to maintain moving average of training loss
avg_rec = AverageRecorder(size=10) # average over latest 10 records
while training:
    loss_0, loss_1  = ...
    avg_rec.add([loss_0, loss_1])
    # avg_rec.avg() == [0.12343452, 0.567800323]
    # avg_rec.avg(0) == 0.12343452
    # avg_rec.to_str(precision=2, ) == '0.12 0.57'

## Use to maintain average of test metrics on the whole test set
avg_rec = AverageRecorder() # average over ALL records
while test:
    metric_0, metric_1  = ...
    avg_rec.add({'m0': metric_0, 'm1': metric_1}) # dict is allowed
print(avg_rec.to_str(precision=4, delimiter=' , '))
# 'm0: 0.1234 , m1: 0.5678'
#
# avg_rec.avg() == {'m0': 0.12343452, 'm1': 0.567800323}
# avg_rec.avg(0) == 0.12343452
```

add(*record*, *weight=None*)

Appends a new record.

`record` can be a *list*, *dict*, or a single scalar. The record type is determined at the first time `add()` is called. All subsequent calls to `add()` must have the same type of `record`.

`record` in subsequent calls to `add()` can contain only a subset of fields than the first call to `add()`.

Example

```
recorder.add({'1': 0.2, '2': 0.2}) # 1st call to `add`
x = recorder.add({'1': 0.4}) # 2nd call to `add`
# x == {'1': 0.3, '2': 0.2}
```

Parameters

- **record** – A single scalar, a list of scalars, or a dict of scalars.
- **weight** (*optional*) – A scalar, weight of the new record for calculating a weighted average. If *None*, weight is set to *1*. For example, `weight` can be set to batch size and `record` the average value of certain metrics on the batch in order to calculate the average metric values on a whole dataset.

Returns The (moving) average after appending the record, with the same type as `record`.

avg(*id_or_name=None*)

Returns the (moving) average.

Parameters **id_or_name** (*optional*) – A list of or a single element. Each element is the index (if the record type is *list*) or name (if the record type is *dict*) of the field for which the average is calculated. If not given, the average of all fields are returned.

Returns The average value(s). If `id_or_name` is a single element (not a list), then returns the average value of the corresponding field. Otherwise, if `id_or_name` is a list of element(s), then returns average value(s) in the same type as `record` of `add()`.

reset(*id_or_name=None*)
Resets the record.

Parameters **id_or_name** (*optional*) – A list or a single element. Each element is the index (if the record type is *list*) or name (if the record type is *dict*) of the field to reset. If *None*, all fields are reset.

to_str(*precision=None*, *delimiter=' '*)
Returns a string of the average values of the records.

Parameters

- **precision** (*int, optional*) – The number of decimal places to keep in the returned string. E.g., for an average value of *0.1234*, `precision = 2` leads to *'0.12'*.
- **delimiter** (*str*) – The delimiter string that separates between fields.

Returns

A string of the average values.

If record is of type *dict*, the string is a concatenation of 'field_name: average_value', delimited with `delimiter`. E.g., *'field_name_1: 0.1234 field_name_2: 0.5678 . . . '*.

Otherwise, the string is of a concatenation of 'average_value'. E.g., *'0.1234 0.5678 . . . '*

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z