Neural Monkey Documentation

Release 0.1

Jindřich Libovický, Jindřich Helcl, Tomáš Musil

Dec 19, 2018

Contents

1 Getting Started **3**

Python Module Index **143**

Neural Monkey is an open-source toolkit for sequence learning using Tensorflow.

If you want to dig in the code, you can browse the repository.

CHAPTER 1

Getting Started

1.1 Installation

Before you start, make sure that you already have installed Python 3.5, pip and git.

Create and activate a virtual environment to install the package into:

```
$ python3 -m venv nm
$ source nm/bin/activate
# after this, your prompt should change
```

Then clone Neural Monkey from GitHub and switch to its root directory:

```
(nm)$ git clone https://github.com/ufal/neuralmonkey
(nm)$ cd neuralmonkey
```

Run pip to install all requirements. For the CPU version install dependencies by this command:

```
(nm)$ pip install --upgrade -r requirements.txt
```

For the GPU version install dependencies try this command:

```
(nm)$ pip install --upgrade -r requirements-gpu.txt
```

If you are using the GPU version, make sure that the `LD_LIBRARY_PATH` environment variable points to `lib` and `lib64` directories of your CUDA and CuDNN installations. Similarly, your `PATH` variable should point to the `bin` subdirectory of the CUDA installation directory.

You made it! Neural Monkey is now installed!

1.1.1 Note for Ubuntu 14.04 Users

If you get Segmentation fault errors at the very end of the training process, you can either ignore it, or follow the steps outlined in this document.

1.2 Package Overview

This overview should provide you with the basic insight on how Neural Monkey conceptualizes the problem of sequence-to-sequence learning and how the data flow during training and running models looks like.

1.2.1 Loading and Processing Datasets

We call a *dataset* a collection of named data *series*. By a series we mean a list of data items of the same type representing one type of input or desired output of a model. In the simple case of machine translation, there are two series: a list of source-language sentences and a list of target-language sentences.

The following scheme captures how a dataset is created from input data.

The dataset is created in the following steps:

1. An input file is read using a *reader*. Reader can e.g., load a file containing paths to JPEG images and load them as `numpy` arrays, or read a tokenized text as a list of lists (sentences) of string tokens.
2. Series created by the readers can be preprocessed by some *series-level preprocessors*. An example of such preprocessing is byte-pair encoding which loads a list of merges and segments the text accordingly.
3. The final step before creating a dataset is applying *dataset-level* preprocessors which can take more series and output a new series.

Currently there are two implementations of a dataset. An in-memory dataset which stores all data in the memory and a lazy dataset which gradually reads the input files step by step and only stores the batches necessary for the computation in the memory.

1.2.2 Training and Running a Model

This section describes the training and running workflow. The main concepts and their interconnection can be seen in the following scheme.

The dataset series can be used to create a *vocabulary*. A vocabulary represents an indexed set of tokens and provides functionality for converting lists of tokenized sentences into matrices of token indices and vice versa. Vocabularies are used by encoders and decoders for feeding the provided series into the neural network.

The model itself is defined by *encoders* and *decoders*. Most of the TensorFlow code is in the encoders and decoders. Encoders are parts of the model which take some input and compute a representation of it. Decoders are model parts that produce some outputs. Our definition of encoders and decoders is more general than in the classical sequence-to-sequence learning. An encoder can be for example a convolutional network processing an image. The RNN decoder is for us only a special type of decoder, it can be also a sequence labeler or a simple multilayer-perceptron classifier.

Decoders are executed using so-called *runners*. Different runners represent different ways of running the model. We might want to get a single best estimation, get an `n`-best list or a sample from the model. We might want to use an RNN decoder to get the decoded sequences or we might be interested in the word alignment obtained by its attention model. This is all done by employing different runners over the decoders. The outputs of the runners can be subject of further *post-processing*.

Additionally to runners, each training experiment has to have its *trainer*. A *trainer* is a special case of a runner that actually modifies the parameters of the model. It collects the objective functions and uses them in an optimizer.

Neural Monkey manages TensorFlow sessions using an object called *TensorFlow manager*. Its basic capability is to execute runners on provided datasets.

1.3 Post-Editing Task Tutorial

This tutorial will guide you through designing your first experiment in Neural Monkey.

Before we get started with the tutorial, please check that you have the Neural Monkey package properly *installed and working*.

1.3.1 Part I. - The Task

This section gives an overall description of the task we will try to solve in this tutorial. To make things more interesting than plain machine translation, let's try automatic post-editing task (APE, rhyming well with Neural Monkey).

In short, automatic post-editing is a task, in which we have a source language sentence (let's call it `f`, as grown-ups do), a machine-translated sentence of `f` (I actually don't know what grown-ups call this, so let's call this `e'`), and we are expected to generate another sentence in the same language as `e'` but cleaned of all the errors that the machine translation system have made (let's call this cleaned sentence `e`). Consider this small example:

Source sentence `f`: Bärbel hat eine Katze.

Machine-translated sentence `e'`: Bärbel has a dog.

Corrected translation `e`: Bärbel has a cat.

In the example, the machine translation system wrongly translated the German word "Katze" as the English word "dog". It is up to the post-editing system to fix this error.

In theory (and in practice), we regard the machine translation task as searching for a target sentence `e*` that has the highest probability of being the translation given the source sentence `f`. You can put it to a formula:

```
e* = argmax_e p(e|f)
```

In the post-editing task, the formula is slightly different:

```
e* = argmax_e p(e|f, e')
```

If you think about this a little, there are two ways one can look at this task. One is that we are translating the machine-translated sentence from a kind of *synthetic* language into a proper one, with additional knowledge what the source sentence was. The second view regards this as an ordinary machine translation task, with a little help from another MT system.

In our tutorial, we will assume the MT system used to produce the sentence `e'` was good enough. We thus generally trust it and expect only to make small edits to the translated sentence in order to make it fully correct. This means that we don't need to train a whole new MT system that would translate the source sentences from scratch. Instead, we will build a system that will tell us how to edit the machine translated sentence `e'`.

1.3.2 Part II. - The Edit Operations

How can an automatic system tell us how to edit a sentence? Here's one way to do it: We will design a set of edit operations and train the system to generate a sequence of these operations. If we consider a sequence of edit operations a function `R` (as in *rewrite*), which transforms one sequence to another, we can adapt the formulas above to suit our needs more:

```
R* = argmax_R p(R(e')|f, e')
e* = R*(e')
```

So we are searching for the best edit function `R*` that, once applied to `e'`, will give us the corrected output `e*`. Another question is what the class of all possible edit functions should look like, for now we simply limit them to functions that can be defined as sequences of edit operations.

The edit function `R` processes the input sequence token-by-token in left-to-right direction. It has a pointer to the input sequence, which starts by pointing to the first word of the sequence.

We design three types of edit operations as follows:

1. KEEP - this operation copies the current word to the output and moves the pointer to the next token of the input,
2. DELETE - this operation does not emit anything to the output and moves the pointer to the next token of the input,
3. INSERT - this operation puts a word on the output, leaving the pointer to the input intact.

The edit function applies all its operations to the input sentence. We handle malformed edit sequences simply: if the pointer reaches the end of the input seqence, operations KEEP and DELETE do nothing. If the sequence of edits ends before the end of the input sentence is reached, we apply as many additional KEEP operations as needed to reach the end of the input sequence.

Let's see another example:

```
Bärbel  has   a     dog          .
KEEP    KEEP  KEEP  DELETE  cat  KEEP
```

The word "cat" on the second line is an INSERT operation parameterized by the word "cat". If we apply all the edit operations to the input (i.e. keep the words "Bärbel", "has", "a", and ".", delete the word "dog" and put the word "cat" in its place), we get the corrected target sentence.

1.3.3 Part III. - The Data

We are going to use the data for WMT 16 shared APE task. You can get them at the WMT 16 website or directly at the Lindat repository. There are three files in the repository:

1. TrainDev.zip - contains training and development data set
2. Test.zip - contains source and translated test data
3. test_pe.zip - contains the post-edited test data

Now - before we start, let's create our experiment directory, in which we will place all our work. We shall call it for example `exp-nm-ape` (feel free to choose another weird string).

Extract all the files into the `exp-nm-ape/data` directory. Rename the files and directories so you get this directory structure:

```
exp-nm-ape
|
\== data
    |
    |== train
    |    |
    |    |== train.src
    |    |== train.mt
    |    \== train.pe
    |
    |== dev
    |    |
    |    |== dev.src
```

(continues on next page)

(continued from previous page)

```
    |    |== dev.mt
    |    \== dev.pe
    |
    \== test
         |
         |== test.src
         |== test.mt
         \== test.pe
```

The data is already tokenized so we don't need to run any preprocessing tools. The format of the data is plain text with one sentence per line. There are 12k training triplets of sentences, 1k development triplets and 2k of evaluation triplets.

Preprocessing of the Data

The next phase is to prepare the post editing sequences that we should learn during training. We apply the Levenshtein algorithm to find the shortest edit path from the translated sentence to the post-edited sentence. As a little coding excercise, you can implement your own script that does the job, or you may use our preprocessing script from the Neural Monkey package. For this, in the neuralmonkey root directory, run:

```
scripts/postedit_prepare_data.py \
  --translated-sentences=exp-nm-ape/data/train/train.mt \
  --target-sentences=exp-nm-ape/data/train/train.pe \
      > exp-nm-ape/data/train/train.edits
```

And the same for the development data.

NOTE: You may have to change the path to the exp-nm-ape directory if it is not located inside the repository root directory.

NOTE 2: There is a hidden option of the preparation script (`--target-german=True`) which turns on some steps tailored for better processing of German text. In this tutorial, we are not going to use it.

If you look at the preprocessed files, you will see that the KEEP and DELETE operations are represented with special tokens while the INSERT operations are represented simply with the word they insert.

Congratulations! Now, you should have train.edits, dev.edits and test.edits files all in their respective data directories. We can now move to work with Neural Monkey configurations!

1.3.4 Part IV. - The Model Configuration

In Neural Monkey, all information about a model and its training is stored in configuration files. The syntax of these files is a plain INI syntax (more specifically, the one which gets processed by Python's ConfigParser). The configuration file is structured into a set of sections, each describing a part of the training. In this section, we will go through all of them and write our configuration file needed for the training of the post-editing task.

First of all, create a file called `post-edit.ini` and put it inside the `exp-nm-ape` directory. Put all the snippets that we will describe in the following paragraphs into the file.

1 - Datasets

For training, we prepare two datasets. The first dataset will serve for the training, the second one for validation. In Neural Monkey, each dataset contains a number of so called *data series*. In our case, we will call the data series *source*, *translated*, and *edits*. Each of those series will contain the respective set of sentences.

It is assumed that all series within a given dataset have the same number of elements (i.e. sentences in our case).

The configuration of the datasets looks like this:

```
[train_dataset]
class=dataset.load_dataset_from_files
s_source="exp-nm-ape/data/train/train.src"
s_translated="exp-nm-ape/data/train/train.mt"
s_edits="exp-nm-ape/data/train/train.edits"

[val_dataset]
class=dataset.load_dataset_from_files
s_source="exp-nm-ape/data/dev/dev.src"
s_translated="exp-nm-ape/data/dev/dev.mt"
s_edits="exp-nm-ape/data/dev/dev.edits"
```

Note that series names (*source*, *translated*, and *edits*) are arbitrary and defined by their first mention. The `s_` prefix stands for "series" and is used only here in the dataset sections, not later when the series are referred to.

These two INI sections represent two calls to function `neuralmonkey.config.dataset_from_files`, with the series file paths as keyword arguments. The function serves as a constructor and builds an object for every call. So at the end, we will have two objects representing the two datasets.

2 - Vocabularies

Each encoder and decoder which deals with language data operates with some kind of vocabulary. In our case, the vocabulary is just a list of all unique words in the training data. Note that apart the special `<keep>` and `<delete>` tokens, the vocabularies for the *translated* and *edits* series are from the same language. We can save some memory and perhaps improve quality of the target language embeddings by share vocabularies for these datasets. Therefore, we need to create only two vocabulary objects:

```
[source_vocabulary]
class=vocabulary.from_dataset
datasets=[<train_dataset>]
series_ids=["source"]
max_size=50000

[target_vocabulary]
class=vocabulary.from_dataset
datasets=[<train_dataset>]
series_ids=["edits", "translated"]
max_size=50000
```

The first vocabulary object (called `source_vocabulary`) represents the (English) vocabulary used for this task. The 50,000 is the maximum size of the vocabulary. If the actual vocabulary of the data was bigger, the rare words would be replaced by the `<unk>` token (hardcoded in Neural Monkey, not part of the 50,000 items), which stands for unknown words. In our case, however, the vocabularies of the datasets are much smaller so we won't lose any words.

Both vocabularies are created out of the training dataset, as specified by the line `datasets=[<train_dataset>]` (more datasets could be given in the list). This means that if there are any unseen words in the development or test data, our model will treat them as unknown words.

We know that the languages in the `translated` series and `edits` are the same (except for the KEEP and DELETE tokens in the edits), so we create a unified vocabulary for them. This is achieved by specifying `series_ids=[edits, translated]`. The one-hot encodings (or more precisely, indices to the vocabulary) will be identical for words in `translated` and `edits`.

3 - Encoders

Our network will have two inputs. Therefore, we must design two separate encoders. The first encoder will process source sentences, and the second will process translated sentences, i.e. the candidate translations that we are expected to post-edit. This is the configuration of the encoder for the source sentences:

```
[src_encoder]
class=encoders.recurrent.SentenceEncoder
rnn_size=300
max_input_len=50
embedding_size=300
dropout_keep_prob=0.8
data_id="source"
name="src_encoder"
vocabulary=<source_vocabulary>
```

This configuration initializes a new instance of sentence encoder with the hidden state size set to 300 and the maximum input length set to 50. (Longer sentences are trimmed.) The sentence encoder looks up the words in a word embedding matrix. The size of the embedding vector used for each word from the source vocabulary is set to 300. The source data series is fed to this encoder. 20% of the weights is dropped out during training from the word embeddings and from the attention vectors computed over the hidden states of this encoder. Note the `name` attribute must be set in each encoder and decoder in order to prevent collisions of the names of Tensorflow graph nodes.

The configuration of the second encoder follows:

```
[trans_encoder]
class=encoders.recurrent.SentenceEncoder
rnn_size=300
max_input_len=50
embedding_size=300
dropout_keep_prob=0.8
data_id="translated"
name="trans_encoder"
vocabulary=<target_vocabulary>
```

This config creates a second encoder for the `translated` data series. The setting is the same as for the first encoder, except for the different vocabulary and name.

To be able to use the attention mechanism, we need to define the attention components for each encoder we want to process. In our tutorial, we use attention over both of the encoders:

```
[src_attention]
class=attention.Attention
name="attention_src_encoder"
encoder=<src_encoder>
dropout_keep_prob=0.8
state_size=300

[trans_attention]
class=attention.Attention
name="attention_trans_encoder"
encoder=<trans_encoder>
dropout_keep_prob=0.8
state_size=300
```

4 - Decoder

Now, we configure perhaps the most important object of the training - the decoder. Without further ado, here it goes:

```
[decoder]
class=decoders.decoder.Decoder
name="decoder"
encoders=[<trans_encoder>, <src_encoder>]
attentions=[<trans_attention>, <src_attention>]
rnn_size=300
max_output_len=50
embeddings_source=<trans_encoder.input_sequence>
dropout_keep_prob=0.8
data_id="edits"
vocabulary=<target_vocabulary>
```

As in the case of encoders, the decoder needs its RNN and embedding size settings, maximum output length, dropout parameter, and vocabulary settings.

The outputs of the individual encoders are by default simply concatenated and projected to the decoder hidden state (of `rnn_size`). Internally, the code is ready to support arbitrary mappings by adding one more parameter here: `encoder_projection`. For an additional view on the encoders, we use the two attention mechanism objects defined in the previous section.

Note that you may set `rnn_size` to `None`. Neural Monkey will then directly use the concatenation of encoder states without any mapping. This is particularly useful when you have just one encoder as in MT.

The line `embeddings_encoder=<trans_encoder.input_sequence>` means that the embeddings (including embedding size) are to be shared with the input sequence object of the `trans_encoder` (the input sequence object is an underlying structure that manages the input layer of an encoder and, in case of `SentenceEncoder`, provides access to the word embeddings.

The loss of the decoder is computed against the `edits` data series of whatever dataset the decoder will be applied to.

5 - Runner and Trainer

As their names suggest, runners and trainers are used for running and training models. The `trainer` object provides the optimization operation to the graph. In the case of the cross entropy trainer (used in our tutorial), the default optimizer is Adam and it is run against the decoder's loss, with added L2 regularization (controlled by the `l2_weight` parameter of the trainer). The runner is used to process a dataset by the model and return the decoded sentences, and (if possible) decoder losses.

We define these two objects like this:

```
[trainer]
class=trainers.cross_entropy_trainer.CrossEntropyTrainer
decoders=[<decoder>]
l2_weight=1.0e-8

[runner]
class=runners.runner.GreedyRunner
decoder=<decoder>
output_series="greedy_edits"
```

Note that a runner can only have one decoder, but during training you can train several decoders, all contributing to the loss function.

The purpose of the trainer is to optimize the model, so we are not interested in the actual outputs it produces, only the loss compared to the reference outputs (and the loss is calculated by the given decoder).

The purpose of the runner is to get the actual outputs and for further use, they are collected to a new series called `greedy_edits` (see the line `output_series=`) of whatever dataset the runner will be applied to.

6 - Evaluation Metrics

During validation, the whole validation dataset gets processed by the models and the decoded sentences are evaluated against a reference to provide the user with the state of the training. For this, we need to specify evaluator objects which will be used to score the outputted sentences. In our case, we will use BLEU and TER:

```
[bleu]
class=evaluators.bleu.BLEUEvaluator
name="BLEU-4"
```

7 - TensorFlow Manager

In order to handle global variables such as how many CPU cores TensorFlow should use, you need to specify a "TensorFlow manager":

```
[tf_manager]
class=tf_manager.TensorFlowManager
num_threads=4
num_sessions=1
minimize_metric=True
save_n_best=3
```

8 - Main Configuration Section

Almost there! The last part of the configuration puts all the pieces together. It is called `main` and specifies the rest of the training parameters:

```
[main]
name="post editing"
output="exp-nm-ape/training"
runners=[<runner>]
tf_manager=<tf_manager>
trainer=<trainer>
train_dataset=<train_dataset>
val_dataset=<val_dataset>
evaluation=[("greedy_edits", "edits", <bleu>), ("greedy_edits", "edits", evaluators.
↪ter.TER)]
batch_size=128
runners_batch_size=256
epochs=100
validation_period=1000
logging_period=20
```

The `output` parameter specifies the directory, in which all the files generated by the training (used for replicability of the experiment, logging, and saving best models variables) are stored. It is also worth noting, that if the output directory exists, the training is not run, unless the line `overwrite_output_dir=True` is also included here.

The `runners`, `tf_manager`, `trainer`, `train_dataset` and `val_dataset` options are self-explanatory.

The parameter `evaluation` takes list of tuples, where each tuple contains: - the name of output series (as produced by some runner), `greedy_edits` here, - the name of the reference series of the dataset, `edits` here, - the reference to the evaluation algorithm, `<bleu>` and `evaluators.ter.TER` in the two tuples here.

The `batch_size` parameter controls how many sentences will be in one training mini-batch. When the model does not fit into GPU memory, it might be a good idea to start reducing this number before anything else. The larger the batch size, however, the sooner the training should converge to the optimum.

Runners are less memory-demanding, so `runners_batch_size` can be set higher than `batch_size`.

The `epochs` parameter specifies the number of passes through the training data that the training loop should do. There is no early stopping mechanism in Neural Monkey yet, the training can be resumed after the end, however. The training can be safely ctrl+C'ed in any time: Neural Monkey preserves the last `save_n_best` best model variables saved on the disk.

The validation and logging periods specify how often to measure the model's performance on the training batch (`logging_period`) or on validation data (`validation_period`). Note that both logging and validation involve running the runners over the current batch or the validation data, resp. If this happens too often, the time needed to train the model can significantly grow.

At each validation (and logging), the output is scored using the specified evaluation metrics. The last of the evaluation metrics (TER in our case) is used to keep track of the model performance over time. Whenever the score on validation data is better than any of the `save_n_best` (3 in our case) previously saved models, the model is saved, discaring unneccessary lower scoring models.

1.3.5 Part V. - Running an Experiment

Now that we have prepared the data and the experiment INI file, we can run the training. If your Neural Monkey installation is OK, you can just run this command from the root directory of the Neural Monkey repository:

```
bin/neuralmonkey-train exp-nm-ape/post-edit.ini
```

You should see the training program reporting the parsing of the configuration file, initializing the model, and eventually the training process. If everything goes well, the training should run for 100 epochs. You should see a new line with the status of the model's performance on the current batch every few seconds, and there should be a validation report printed every few minutes.

As given in the `main.output` config line, the Neural Monkey creates the directory `experiments/training` with these files:

- `git_commit` - the Git hash of the current Neural Monkey revision.
- `git_diff` - the diff between the clean checkout and the working copy.
- `experiment.ini` - the INI file used for running the training (a simple copy of the file NM was started with).
- `experiment.log` - the output log of the training script.
- `checkpoint` - file created by Tensorflow, keeps track of saved variables.
- `events.out.tfevents.<TIME>.<HOST>` - file created by Tensorflow, keeps the summaries for TensorBoard visualisation
- `variables.data[.<N>]` - a set of files with N best saved models.
- `variables.data.best` - a symbolic link that points to the variable file with the best model.

1.3.6 Part VI. - Evaluation of the Trained Model

If you have reached this point, you have nearly everything this tutorial offers. The last step of this tutorial is to take the trained model and to apply it to a previously unseen dataset. For this you will need two additional configuration files. But fear not - it's not going to be that difficult. The first configuration file is the specification of the model. We have this from Part III and a small optional change is needed. The second configuration file tells the run script which datasets to process.

The optional change of the model INI file prevents the training dataset from loading. This is a flaw in the present design and it is planned to change. The procedure is simple:

1. Copy the file `post-edit.ini` into e.g. `post-edit.test.ini`
2. Open the `post-edit.test.ini` file and remove the `train_dataset` and `val_dataset` sections, as well as the `train_dataset` and `val_dataset` configuration from the `[main]` section.

Now we have to make another file specifying the testing dataset configuration. We will call this file `post-edit_run.ini`:

```
[main]
test_datasets=[<eval_data>]

[eval_data]
class=dataset.load_dataset_from_files
s_source="exp-nm-ape/data/test/test.src"
s_translated="exp-nm-ape/data/test/test.mt"
s_greedy_edits_out="exp-nm-ape/test_output.edits"
```

The dataset specifies the two input series `s_source` and `s_translated` (the candidate MT output output to be post-edited) as in the training. The series `s_edits` (containing reference edits) is **not** present in the evaluation dataset, because we do not want to use the reference edits to compute loss at this point. Usually, we don't even *know* the correct output at runtime.

Instead, we introduce the output series `s_greedy_edits_out` (the prefix `s_` and the suffix `_out` are hardcoded in Neural Monkey and the series name in between has to match the name of the series produced by the runner).

The line `s_greedy_edits_out=` specifies the file where the output should be saved. (You may want to alter the path to the `exp-nm-ape` directory if it is not located inside the Neural Monkey package root dir.)

We have all that we need to run the trained model on the evaluation dataset. From the root directory of the Neural Monkey repository, run:

```
bin/neuralmonkey-run exp-nm-ape/post-edit.test.ini exp-nm-ape/post-edit_run.ini
```

At the end, you should see a new file `exp-nm-ape/test_output.edits`. As you notice, the contents of this file are the sequences of edit operations, which if applied to the machine translated sentences, generate the output that we want. The final step is to call the provided post-processing script. Again, feel free to write your own as a simple exercise:

```
scripts/postedit_reconstruct_data.py \
  --edits=exp-nm-ape/test_output.edits \
  --translated-sentences=exp-nm-ape/data/test/test.mt \
    > test_output.pe
```

Now, you can run the official tools (like mteval or the tercom software available on the WMT 16 website) to measure the score of `test_output.pe` on the `data/test/test.pe` reference evaluation dataset.

1.3.7 Part VII. - Conclusions

This tutorial gave you the basic overview of how to design your experiments using Neural Monkey. The sample experiment was the task of automatic post-editing. We got the data from the WMT 16 APE shared task and pre-processed them to fit our needs. We have written the configuration file and run the training. At the end, we evaluated the model on the test dataset.

If you want to learn more, the next step is perhaps to browse the `examples` directory in Neural Monkey repository and see some further possible setups. If you are planning to just design an experiment using existing modules, you can start by editing one of those examples as well.

If you want to dig in the code, you can browse the repository. Please feel free to fork the repository and to send us pull requests. The API documentation is currently under construction, but it already contains a little information about Neural Monkey objects and their configuraiton options.

Have fun!

1.4 Machine Translation Tutorial

This tutorial will guide you through designing Machnine Translation experiments in Neural Monkey. We assumes that you already read *the post-editing tutorial*.

The goal of the translation task is to translate sentences from one language into another. For this tutorial we use data from the WMT 16 IT-domain translation shared task on English-to-Czech direction.

WMT is an annual machine translation conference where academic groups compete in translating different datasets over various language pairs.

1.4.1 Part I. - The Data

We are going to use the data for the WMT 16 IT-domain translation shared task. You can get them at the WMT IT Translation Shared Task webpage and there download Batch1 and Batch2 answers and Batch3 as a testing set. Or directly here and testset.

Note: In this tutorial we are using only small dataset as an example, which is not big enough for real-life machine translation training.

We find several files for different languages in the downloaded archive. From which we use only the following files as our training, validation and test set:

```
1. ``Batch1a_cs.txt and Batch1a_en.txt`` as our Training set
2. ``Batch2a_cs.txt and Batch2a_en.txt`` as a Validation set
3. ``Batch3a_en.txt`` as a Test set
```

Now - before we start, let's make our experiment directory, in which we place all our work. Let's call it `exp-nm-mt`.

First extract all the downloaded files, then make gzip files from individual files and put arrange them into the following directory structure:

```
exp-nm-mt
|
\== data
     |
     |== train
     |    |
     |    |== Batch1a_en.txt.gz
```

(continues on next page)

(continued from previous page)

```
    |    \== Batch1a_cs.txt.gz
    |
    |== dev
    |    |
    |    |== Batch2a_en.txt.gz
    |    \== Batch2a_cs.txt.gz
    |
    \== test
         |
         \== Batch3a_en.txt.gz
```

The gzipping is not necessary, if you put the dataset there in plaintext, it will work the same way. Neural Monkey recognizes gzipped files by their MIME

type and chooses the correct way to open them.

TODO The dataset is not tokenized and need to be preprocessed.

Byte Pair Encoding

Neural machine translation (NMT) models typically operate with a fixed vocabulary, but translation is an open-vocabulary problem. Byte pair encoding (BPE) enables NMT model translation on open-vocabulary by encoding rare and unknown words as sequences of subword units. This is based on an intuition that various word classes are translatable via smaller units than words. More information in the paper https://arxiv.org/abs/1508.07909 BPE creates a list of merges that are used for splitting out-of-vocabulary words. Example of such splitting:

```
basketball => basket@@ ball
```

Postprocessing can be manually done by:

```
sed "s/@@ //g"
```

but Neural Monkey manages it for you.

BPE Generation

In order to use BPE, you must first generate *merge_file*, over all data. This file is generated on both source and target dataset. You can generate it by running following script:

```
neuralmonkey/lib/subword_nmt/learn_bpe.py -s 50000 < DATA > merge_file.bpe
```

With the data from this tutorial it would be the following command:

```
paste Batch1a_en.txt Batch1a_cs.txt \
| neuralmonkey/lib/subword_nmt/learn_bpe.py -s 8000 \
> exp-nm-mt/data/merge_file.bpe
```

You can change number of merges, this number is equivalent to the size of the vocabulary. Do not forget that as an input is the file containing both source and target sides.

1.4.2 Part II. - The Model Configuration

In this section, we create the configuration file `translation.ini` needed for the machine translation training. We mention only the differences from the main post-editing tutorial.

1 - Datasets

For training, we prepare two datasets. Since we are using BPE, we need to define the preprocessor. The configuration of the datasets looks like this:

```
[train_data]
class=dataset.load_dataset_from_files
s_source="exp-nm-mt/data/train/Batch1a_en.txt.gz"
s_target="exp-nm-mt/data/train/Batch1a_cs.txt.gz"
preprocessors=[("source", "source_bpe", <bpe_preprocess>), ("target", "target_bpe",
↪<bpe_preprocess>)]

[val_data]
class=dataset.load_dataset_from_files
s_source="exp-nm-mt/data/dev/Batch2a_en.txt.gz"
s_target="exp-nm-mt/data/dev/Batch2a_cs.txt.gz"
preprocessors=[("source", "source_bpe", <bpe_preprocess>), ("target", "target_bpe",
↪<bpe_preprocess>)]
```

2 - Preprocessor and Postprocessor

We need to tell the Neural Monkey how it should handle preprocessing and postprocessing due to the BPE:

```
[bpe_preprocess]
class=processors.bpe.BPEPreprocessor
merge_file="exp-nm-mt/data/merge_file.bpe"

[bpe_postprocess]
class=processors.bpe.BPEPostprocessor
```

3 - Vocabularies

For both encoder and decoder we use shared vocabulary created from BPE merges:

```
[shared_vocabulary]
class=vocabulary.from_bpe
path="exp-nm-mt/data/merge_file.bpe"
```

4 - Encoder and Decoder

The encoder and decored are similar to those from *the post-editing tutorial*:

```
[encoder]
class=encoders.recurrent.SentenceEncoder
name="sentence_encoder"
rnn_size=300
max_input_len=50
embedding_size=300
dropout_keep_prob=0.8
data_id="source_bpe"
vocabulary=<shared_vocabulary>

[attention]
```

(continues on next page)

(continued from previous page)

```
class=attentions.Attention
name="att_sent_enc"
encoder=<encoder>
state_size=300
dropout_keep_prob=0.8

[decoder]
class=decoders.decoder.Decoder
name="decoder"
encoders=[<encoder>]
attentions=[<attention>]
rnn_size=256
embedding_size=300
dropout_keep_prob=0.8
data_id="target_bpe"
vocabulary=<shared_vocabulary>
max_output_len=50
```

You can notice that both encoder and decoder uses as input data id the data preprocessed by *<bpe_preprocess>*.

5 - Training Sections

The following sections are described in more detail in *the post-editing tutorial*:

```
[trainer]
class=trainers.cross_entropy_trainer.CrossEntropyTrainer
decoders=[<decoder>]
l2_weight=1.0e-8

[runner]
class=runners.runner.GreedyRunner
decoder=<decoder>
output_series="series_named_greedy"
postprocess=<bpe_postprocess>

[bleu]
class=evaluators.bleu.BLEUEvaluator
name="BLEU-4"

[tf_manager]
class=tf_manager.TensorFlowManager
num_threads=4
num_sessions=1
minimize_metric=False
save_n_best=3
```

As for the main configuration section do not forget to add BPE postprocessing:

```
[main]
name="machine translation"
output="exp-nm-mt/out-example-translation"
runners=[<runner>]
tf_manager=<tf_manager>
trainer=<trainer>
train_dataset=<train_data>
```

(continues on next page)

(continued from previous page)

```
val_dataset=<val_data>
evaluation=[("series_named_greedy", "target", <bleu>), ("series_named_greedy", "target
↪", evaluators.ter.TER)]
batch_size=80
runners_batch_size=256
epochs=10
validation_period=5000
logging_period=80
```

1.4.3 Part III. - Running and Evaluation of the Experiment

1 - Training

The training can be run as simply as:

```
bin/neuralmonkey-train exp-nm-mt/translation.ini
```

2 - Resuming Training

If training stopped and you want to resume it, you can load pre-trained parameters by specifying the `initial_variables` of the model in the [main] section:

```
[main]
initial_variables=/path/to/variables.data
```

Note there is actually no file called `variables.data`, but three files with this common prefix. The `initial_variables` config value should correspond to this prefix.

3 - Evaluation

As for the evaluation, you need to create `translation_run.ini`:

```
[main]
test_datasets=[<eval_data>]
; We saved 3 models (save_n_best=3), so there are
; multiple models we could to translate with.
; We can go with the best model, or select one manually:
;variables=["exp-nm-mt/out-example-translation/variables.data.0"]

[bpe_preprocess]
class=processors.bpe.BPEPreprocessor
merge_file="exp-nm-mt/data/merge_file.bpe"

[eval_data]
class=dataset.load_dataset_from_files
s_source="exp-nm-mt/data/test/Batch3a_en.txt.gz"
s_series_named_greedy_out="exp-nm-mt/out-example-translation/evaluation.txt.out"
preprocessors=[("source", "source_bpe", <bpe_preprocess>)]
```

and run:

```
bin/neuralmonkey-run exp-nm-mt/translation.ini exp-nm-mt/translation_run.ini
```

You are ready to experiment with your own models.

1.5 Configuration

Experiments with NeuralMonkey are configured using configuration files which specifies the architecture of the model, meta-parameters of the learning, the data, the way the data are processed and the way the model is run.

1.5.1 Syntax

The configuration files are based on the syntax of INI files, see e.g., the corresponding Wikipedia page..

Neural Monkey INI files contain *key-value pairs*, delimited by an equal sign (=) with no spaces around. The key-value pairs are grouped into *sections* (Neural Monkey requires all pairs to belong to a section.)

Every section starts with its header which consists of the section name in square brackets. Everything below the header is considered a part of the section.

Comments can appear on their own (otherwise empty) line, prefixed either with a hash sign (`#`) or a semicolon (`;`) and possibly indented.

The configuration introduces several additional constructs for the values. There are both atomic values, and compound values.

Supported atomic values are:

- booleans: literals `True` and `False`
- integers: strings that could be interpreted as integers by Python (e.g., `1`, `002`)
- floats: strings that could be interpreted as floats by Python (e.g., `1.0`, `.123`, `2.`, `2.34e-12`)
- strings: string literals in quotes (e.g., `"walrus"`, `"5"`)
- section references: string literals in angle brackets (e.g., `<encoder>`), sections are later interpreted as Python objects
- Python names: strings without quotes which are neither booleans, integers and floats, nor section references (e.g., `neuralmonkey.encoders.SentenceEncoder`)

On top of that, there are two compound types syntax from Python:

- lists: comma-separated in squared brackets (e.g., `[1, 2, 3]`)
- tuples: comma-separated in round brackets (e.g., `("target", <ter>)`)

1.5.2 Variables

The configuration file can contain a `[vars]` section, defining variables which can be used in the rest of the config file. Their values can have any of the types listed above. To reference a variable, use its name preceded by a dollar sign (e.g. `$variable`). Variable values can also be included inside strings using the `str.format()` notation. For example:

```
[vars]
parent_dir="experiments"
drop_keep_p=0.5
output_dir="{parent_dir}/test_drop{drop_keep_p:.2f}"
prefix="my"

[main]
output=$output_dir

...

[encoder]
name="{prefix}_encoder"
dropout_keep_prob=$drop_keep_p
...
```

1.5.3 Interpretation

Each configuration file contains a `[main]` section which is interpreted as a dictionary having keys specified in the section and values which are results of interpretation of the right hand sides.

Both the atomic and compound types taken from Python (i.e., everything except the section references) are interpreted as their Python counterparts. (So if you write 42, Neural Monkey actually sees 42.)

Section references are interpreted as references to objects constructed when interpreting the referenced section. (So if you write `<session_manager>` in a right-hand side and a section `[session_manager]` later in the file, Neural Monkey will construct a Python object based on the key-value pairs in the section `[session_manager]`.)

Every section except the `[main]` and `[vars]` sections needs to contain the key `class` with a value of Python name which is a callable (e.g., a class constructor or a function). The other keys are used as named arguments of the callable.

1.5.4 Session Manager

This and following sections describes TensorFlow Manager from the users' perspective: what can be configured in Neural Monkey with respect to TensorFlow. The configuration of the TensorFlow manager is specified within the INI file in section with class *`neuralmonkey.tf_manager.TensorFlowManager`*:

```
[session_manager]
class=tf_manager.TensorFlowManager
...
```

The `session_manager` configuration object is then referenced from the main section of the configuration:

```
[main]
tf_manager=<session_manager>
...
```

1.5.5 Training on GPU

You can easily switch between CPU and GPU version by running your experiments in virtual environment containing either CPU or GPU version of TensorFlow without any changes to config files.

Similarly, standard techniques like setting the environment variable `CUDA_VISIBLE_DEVICES` can be used to control which GPUs are accessible for Neural Monkey.

By default, Neural Monkey prefers to allocate GPU memory stepwise only as needed. This can create problems with memory fragmentation. If you know that you can allocate the whole memory at once add the following parameter the `session_manager` section:

```
gpu_allow_growth=False
```

You can also restrict TensorFlow to use only a fixed proportion of GPU memory:

```
per_process_gpu_memory_fraction=0.65
```

This parameter tells TensorFlow to use only 65% of GPU memory.

1.5.6 Training on CPUs

TensorFlow Manager settings also affect training on CPUs.

The line:

```
num_threads=4
```

indicates that 4 CPUs should be used for TensorFlow computations.

1.6 API Documentation

1.6.1 `neuralmonkey` package

The neuralmonkey package is the root package of this project.

Sub-modules

neuralmonkey

neuralmonkey package

Subpackages

neuralmonkey.attention package

Submodules

neuralmonkey.attention.base_attention module

Decoding functions using multiple attentions for RNN decoders.

See http://arxiv.org/abs/1606.07481

The attention mechanisms used in Neural Monkey are inherited from the `BaseAttention` class defined in this module.

The attention function can be viewed as a soft lookup over an associative memory. The *query* vector is used to compute a similarity score of the *keys* of the associative memory and the resulting scores are used as weights in a

weighted sum of the *values* associated with the keys. We call the (unnormalized) similarity scores *energies*, we call *attention distribution* the energies after (softmax) normalization, and we call the resulting weighted sum of states a *context vector*.

Note that it is possible (and true in most cases) that the attention keys are equal to the values. In case of self-attention, even queries are from the same set of vectors.

To abstract over different flavors of attention mechanism, we conceptualize the procedure as follows: Each attention object has the `attention` function which operates on the query tensor. The attention function receives the query tensor (the decoder state) and optionally the previous state of the decoder, and computes the context vector. The function also receives a *loop state*, which is used to store data in an autoregressive loop that generates a sequence.

The attention uses the loop state to store to store attention distributions and context vectors in time. This structure is called `AttentionLoopState`. To be able to initialize the loop state, each attention object that uses this feature defines the `initial_loop_state` function with empty tensors.

Since there can be many *modes* in which the decoder that uses the attention operates, the attention objects have the `finalize_loop` method, which takes the last attention loop state and the name of the mode (a string) and processes this data to be available in the `histories` dictionary. The single and most used example of two *modes* are the *train* and *runtime* modes of the autoregressive decoder.

class neuralmonkey.attention.base_attention.**BaseAttention**(*name: str, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.model.model_part.ModelPart`

The abstract class for the attenion mechanism flavors.

__init__(*name: str, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new `BaseAttention` object.

attention(*query: tensorflow.python.framework.ops.Tensor, decoder_prev_state: tensorflow.python.framework.ops.Tensor, decoder_input: tensorflow.python.framework.ops.Tensor, loop_state: Any*) → Tuple[tensorflow.python.framework.ops.Tensor, Any]

Get context vector for a given query.

context_vector_size

Return the static size of the context vector.

Returns An integer specifying the context vector dimension.

finalize_loop(*key: str, last_loop_state: Any*) → None

Store the attention histories from loop state under a given key.

Parameters

- **key** – The key to the histories dictionary to store the data in.
- **last_loop_state** – The loop state object from the last state of the decoding loop.

histories

Return the attention histories dictionary.

Use this property after it has been populated.

Returns The attention histories dictionary.

initial_loop_state() → Any

Get initial loop state for the attention object.

Returns The newly created initial loop state object.

visualize_attention(*key: str*, *max_outputs: int = 16*) → None

Include the attention histories under a given key into a summary.

Parameters

- **key** – The key to the attention histories dictionary.
- **max_outputs** – Maximum number of images to save.

neuralmonkey.attention.base_attention.**empty_attention_loop_state**(*batch_size: Union[int, tensorflow.python.framework.ops.Tensor], length: Union[int, tensorflow.python.framework.ops.Tensor], dimension: Union[int, tensorflow.python.framework.ops.Tensor]*) → neuralmonkey.attention.namedtuples.Attenti

Create an empty attention loop state.

The attention loop state is a technical object for storing the attention distributions and the context vectors in time. It is used with the `tf.while_loop` dynamic implementation of decoders.

Parameters

- **batch_size** – The size of the batch.
- **length** – The number of encoder states (keys).
- **dimension** – The dimension of the context vector

Returns This function returns an empty attention loop state which means there are two empty Tensors one for attention distributions in time, and one for the attention context vectors in time.

neuralmonkey.attention.base_attention.**get_attention_mask**(*encoder: Union[neuralmonkey.model.stateful.TemporalStat neuralmonkey.model.stateful.SpatialStateful]*) → Union[tensorflow.python.framework.ops.Tensor, NoneType]

Return the temporal or spatial mask of an encoder.

Parameters **encoder** – The encoder to get the mask from.

Returns Either a 2D or a 3D tensor, depending on whether the encoder is temporal (e.g. recurrent encoder) or spatial (e.g. a CNN encoder).

neuralmonkey.attention.base_attention.**get_attention_states**(*encoder: Union[neuralmonkey.model.stateful.TemporalS neuralmonkey.model.stateful.SpatialStateful]*) → tensorflow.python.framework.ops.Tensor

Return the temporal or spatial states of an encoder.

Parameters **encoder** – The encoder with the states to attend.

Returns Either a 3D or a 4D tensor, depending on whether the encoder is temporal (e.g. recurrent encoder) or spatial (e.g. a CNN encoder). The first two dimensions are (batch, time).

neuralmonkey.attention.combination module

Attention combination strategies.

This modules implements attention combination strategies for multi-encoder scenario when we may want to combine the hidden states of the encoders in more complicated fashion.

Currently there are two attention combination strategies flat and hierarchical (see paper Attention Combination Strategies for Multi-Source Sequence-to-Sequence Learning).

The combination strategies may use the sentinel mechanism which allows the decoder not to attend to the, and extract information on its own hidden state (see paper Knowing when to Look: Adaptive Attention via a Visual Sentinel for Image Captioning).

class neuralmonkey.attention.combination.**FlatMultiAttention**(*name: str, encoders: List[Union[neuralmonkey.model.stateful.Temp neuralmonkey.model.stateful.SpatialStateful]], attention_state_size: int, share_attn_projections: bool = False, use_sentinels: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: *neuralmonkey.attention.combination.MultiAttention*

Flat attention combination strategy.

Using this attention combination strategy, hidden states of the encoders are first projected to the same space (different projection for different encoders) and then we compute a joint distribution over all the hidden states. The context vector is then a weighted sum of another / then projection of the encoders hidden states. The sentinel vector can be added as an additional hidden state.

See equations 8 to 10 in the Attention Combination Strategies paper.

__init__(*name: str, encoders: List[Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful]], attention_state_size: int, share_attn_projections: bool = False, use_sentinels: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new `BaseAttention` object.

attention(*query: tensorflow.python.framework.ops.Tensor, decoder_prev_state: tensorflow.python.framework.ops.Tensor, decoder_input: tensorflow.python.framework.ops.Tensor, loop_state: neuralmonkey.attention.namedtuples.AttentionLoopState*) → Tuple[tensorflow.python.framework.ops.Tensor, neuralmonkey.attention.namedtuples.AttentionLoopState]

Get context vector for given decoder state.

context_vector_size

Return the static size of the context vector.

Returns An integer specifying the context vector dimension.

finalize_loop(*key: str, last_loop_state: neuralmonkey.attention.namedtuples.AttentionLoopState*) → None

Store the attention histories from loop state under a given key.

Parameters

- **key** – The key to the histories dictionary to store the data in.
- **last_loop_state** – The loop state object from the last state of the decoding loop.

get_encoder_projections(*scope*)

initial_loop_state() → neuralmonkey.attention.namedtuples.AttentionLoopState

Get initial loop state for the attention object.

Returns The newly created initial loop state object.

class neuralmonkey.attention.combination.**HierarchicalMultiAttention**(*name: str, attentions: List[neuralmonkey.attention.base_attention.BaseAttention], attention_state_size: int, use_sentinels: bool, share_attn_projections: bool, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.attention.combination.MultiAttention`

Hierarchical attention combination.

Hierarchical attention combination strategy first computes the context vector for each encoder separately using whatever attention type the encoders have. After that it computes a second attention over the resulting context vectors and optionally the sentinel vector.

See equations 6 and 7 in the Attention Combination Strategies paper.

__init__(*name: str, attentions: List[neuralmonkey.attention.base_attention.BaseAttention], attention_state_size: int, use_sentinels: bool, share_attn_projections: bool, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new `BaseAttention` object.

attention(*query: tensorflow.python.framework.ops.Tensor, decoder_prev_state: tensorflow.python.framework.ops.Tensor, decoder_input: tensorflow.python.framework.ops.Tensor, loop_state: neuralmonkey.attention.namedtuples.HierarchicalLoopState*) → Tuple[tensorflow.python.framework.ops.Tensor, neuralmonkey.attention.namedtuples.HierarchicalLoopState]

Get context vector for given decoder state.

context_vector_size

Return the static size of the context vector.

Returns An integer specifying the context vector dimension.

finalize_loop(*key: str, last_loop_state: Any*) → None

Store the attention histories from loop state under a given key.

Parameters

- **key** – The key to the histories dictionary to store the data in.
- **last_loop_state** – The loop state object from the last state of the decoding loop.

initial_loop_state() → neuralmonkey.attention.namedtuples.HierarchicalLoopState
Get initial loop state for the attention object.

Returns The newly created initial loop state object.

class neuralmonkey.attention.combination.**MultiAttention**(*name: str, attention_state_size: int, share_attn_projections: bool = False, use_sentinels: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None
Bases: *neuralmonkey.attention.base_attention.BaseAttention*

Base class for attention combination.

__init__(*name: str, attention_state_size: int, share_attn_projections: bool = False, use_sentinels: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None
Create a new BaseAttention object.

attention(*query: tensorflow.python.framework.ops.Tensor, decoder_prev_state: tensorflow.python.framework.ops.Tensor, decoder_input: tensorflow.python.framework.ops.Tensor, loop_state: Any*) → Tuple[tensorflow.python.framework.ops.Tensor, Any]
Get context vector for given decoder state.

attn_size

attn_v

neuralmonkey.attention.coverage module

Coverage attention introduced in Tu et al. (2016).

See arxiv.org/abs/1601.04811

The CoverageAttention class inherites from the basic feed-forward attention introduced by Bahdanau et al. (2015)

class neuralmonkey.attention.coverage.**CoverageAttention**(*name: str, encoder: Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful], dropout_keep_prob: float = 1.0, state_size: int = None, max_fertility: int = 5, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.attention.feed_forward.Attention`

__init__(*name: str, encoder: Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful], dropout_keep_prob: float = 1.0, state_size: int = None, max_fertility: int = 5, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new `BaseAttention` object.

get_energies(*y: tensorflow.python.framework.ops.Tensor, weights_in_time: tensorflow.python.framework.ops.Tensor*)

neuralmonkey.attention.feed_forward module

The feed-forward attention mechanism.

This is the attention mechanism used in Bahdanau et al. (2015)

See arxiv.org/abs/1409.0473

class neuralmonkey.attention.feed_forward.**Attention**(*name: str, encoder: Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful], dropout_keep_prob: float = 1.0, state_size: int = None, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.attention.base_attention.BaseAttention`

__init__(*name: str, encoder: Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful], dropout_keep_prob: float = 1.0, state_size: int = None, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new `BaseAttention` object.

attention (*query: tensorflow.python.framework.ops.Tensor, decoder_prev_state: tensorflow.python.framework.ops.Tensor, decoder_input: tensorflow.python.framework.ops.Tensor, loop_state: neuralmonkey.attention.namedtuples.AttentionLoopState*) → Tuple[tensorflow.python.framework.ops.Tensor, neuralmonkey.attention.namedtuples.AttentionLoopState]

Get context vector for a given query.

attention_mask

attention_states

bias_term

context_vector_size

Return the static size of the context vector.

Returns An integer specifying the context vector dimension.

finalize_loop (*key: str, last_loop_state: neuralmonkey.attention.namedtuples.AttentionLoopState*) → None

Store the attention histories from loop state under a given key.

Parameters

- **key** – The key to the histories dictionary to store the data in.
- **last_loop_state** – The loop state object from the last state of the decoding loop.

get_energies (*y, _*)

hidden_features

initial_loop_state () → neuralmonkey.attention.namedtuples.AttentionLoopState

Get initial loop state for the attention object.

Returns The newly created initial loop state object.

key_projection_matrix

projection_bias_vector

query_projection_matrix

similarity_bias_vector

state_size

neuralmonkey.attention.namedtuples module

class neuralmonkey.attention.namedtuples.**AttentionLoopState**

Bases: *neuralmonkey.attention.namedtuples.AttentionLoopState*

Basic loop state of an attention mechanism.

contexts

A tensor of shape `(query_time, batch, context_dim)` which stores the context vectors for every decoder time step.

weights

A tensor of shape `(query_time, batch, keys_len)` which stores the attention distribution over the keys given the query in each decoder time step.

class neuralmonkey.attention.namedtuples.**HierarchicalLoopState**
Bases: *neuralmonkey.attention.namedtuples.HierarchicalLoopState*

Loop state of the hierarchical attention mechanism.

The input to the hierarchical attetnion is the output of a set of underlying (child) attentions. To record the inner states of the underlying attentions, we use the `HierarchicalLoopState`, which holds information about both the underlying attentions, and the top-level attention itself.

child_loop_states
A list of attention loop states of the underlying attention mechanisms.

loop_state
The attention loop state of the top-level attention.

class neuralmonkey.attention.namedtuples.**MultiHeadLoopState**
Bases: *neuralmonkey.attention.namedtuples.MultiHeadLoopState*

Loop state of a multi-head attention.

contexts
A tensor of shape `(query_time, batch, context_dim)` which stores the context vectors for every decoder time step.

head_weights
A tensor of shape `(query_time, n_heads, batch, keys_len)` which stores the attention distribution over the keys given the query in each decoder time step **for each attention head**.

neuralmonkey.attention.scaled_dot_product module

The scaled dot-product attention mechanism defined in Vaswani et al. (2017).

The attention energies are computed as dot products between the query vector and the key vector. The query vector is scaled down by the square root of its dimensionality. This attention function has no trainable parameters.

See arxiv.org/abs/1706.03762

class neuralmonkey.attention.scaled_dot_product.**MultiHeadAttention**(*name: str, n_heads: int, keys_encoder: Union[neuralmonkey.model.statefu neural-monkey.model.stateful.SpatialState val-ues_encoder: Union[neuralmonkey.model.statefu neural-monkey.model.stateful.SpatialState = None, dropout_keep_prob: float = 1.0, reuse: neural-monkey.model.model_part.ModelPa = None, save_checkpoint: str = None, load_checkpoint: str = None, ini-tializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.attention.base_attention.BaseAttention`

__init__(*name: str, n_heads: int, keys_encoder: Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful], values_encoder: Union[neuralmonkey.model.stateful.TemporalStateful, neural-monkey.model.stateful.SpatialStateful] = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new `BaseAttention` object.

attention(*query: tensorflow.python.framework.ops.Tensor, decoder_prev_state: tensorflow.python.framework.ops.Tensor, decoder_input: ten-sorflow.python.framework.ops.Tensor, loop_state: neural-monkey.attention.namedtuples.MultiHeadLoopState*) → Tu-ple[tensorflow.python.framework.ops.Tensor, neuralmonkey.attention.namedtuples.MultiHeadLoopState]

Run a multi-head attention getting context vector for a given query.

This method is an API-wrapper for the global function 'attention' defined in this module. Transforms a query of shape(batch, query_size) to shape(batch, 1, query_size) and applies the attention function. Output context has shape(batch, 1, value_size) and weights have shape(batch, n_heads, 1, time(k)). The output is then processed to produce output vector of contexts and the following attention loop state.

Parameters

- **query** – Input query for the current decoding step of shape(batch, query_size).
- **decoder_prev_state** – Previous state of the decoder.

- **decoder_input** – Input to the RNN cell of the decoder.
- **loop_state** – Attention loop state.

Returns Vector of contexts and the following attention loop state.

context_vector_size
Return the static size of the context vector.

Returns An integer specifying the context vector dimension.

finalize_loop (*key: str*, *last_loop_state: neuralmonkey.attention.namedtuples.MultiHeadLoopState*) → None
Store the attention histories from loop state under a given key.

Parameters

- **key** – The key to the histories dictionary to store the data in.
- **last_loop_state** – The loop state object from the last state of the decoding loop.

initial_loop_state () → neuralmonkey.attention.namedtuples.MultiHeadLoopState
Get initial loop state for the attention object.

Returns The newly created initial loop state object.

visualize_attention (*key: str*, *max_outputs: int = 16*) → None
Include the attention histories under a given key into a summary.

Parameters

- **key** – The key to the attention histories dictionary.
- **max_outputs** – Maximum number of images to save.

class neuralmonkey.attention.scaled_dot_product.**ScaledDotProdAttention**(*name: str, keys_encoder: Union[neuralmonkey.model.s neuralmonkey.model.stateful.Spatia values_encoder: Union[neuralmonkey.model.s neuralmonkey.model.stateful.Spatia = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.M = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.attention.scaled_dot_product.MultiHeadAttention`

__init__(*name: str, keys_encoder: Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful], values_encoder: Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful] = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new `BaseAttention` object.

neuralmonkey.attention.scaled_dot_product.**attention**(*queries: tensorflow.python.framework.ops.Tensor, keys: tensorflow.python.framework.ops.Tensor, values: tensorflow.python.framework.ops.Tensor, keys_mask: tensorflow.python.framework.ops.Tensor, num_heads: int, dropout_callback: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], masked: bool = False, use_bias: bool = False*) → tensorflow.python.framework.ops.Tensor

Run multi-head scaled dot-product attention.

See arxiv.org/abs/1706.03762

When performing multi-head attention, the queries, keys and values vectors are first split to sets of smaller vectors, one for each attention head. Next, they are transformed using a linear layer and a separate attention (from a corresponding head) is applied on each set of the transformed triple of query, key and value. The resulting contexts from each head are then concatenated and a linear layer is applied on this concatenated output. The following can be summed by following equations:

```
MultiHead(Q, K, V) = Concat(head_1, ..., head_h) * W_o
head_i = Attention(Q * W_Q_i, K * W_K_i, V * W_V_i)
```

The scaled dot-product attention is a simple dot-product between the query and a transposed key vector. The result is then scaled using square root of the vector dimensions and a softmax layer is applied. Finally, the output of the softmax layer is multiplied by the value vector. See the following equation:

```
Attention(Q, K, V) = softmax(Q * K^T / (d_k)) * V
```

Parameters

- **queries** – Input queries of shape `(batch, time(q), k_channels)`.
- **keys** – Input keys of shape `(batch, time(k), k_channels)`.
- **values** – Input values of shape `(batch, time(k), v_channels)`.
- **keys_mask** – A float Tensor for masking sequences in keys.
- **num_heads** – Number of attention heads.
- **dropout_callback** – Callable function implementing dropout.
- **masked** – Boolean indicating whether we want to mask future energies.
- **use_bias** – If True, enable bias in the attention head projections (for all queries, keys and values).

Returns Contexts of shape `(batch, time(q), v_channels)` and weights of shape `(batch, time(q), time(k))`.

neuralmonkey.attention.scaled_dot_product.**empty_multi_head_loop_state**(*batch_size: Union[int, tensorflow.python.framework.ops.Ter num_heads: Union[int, tensorflow.python.framework.ops.Ter length: Union[int, tensorflow.python.framework.ops.Ter dimension: Union[int, tensorflow.python.framework.ops.Ter* → neuralmonkey.attention.namedtuples

neuralmonkey.attention.scaled_dot_product.**mask_energies**(*energies_4d: tensorflow.python.framework.ops.Tensor*, *mask: tensorflow.python.framework.ops.Tensor*, *mask_value=-1000000000.0*) → tensorflow.python.framework.ops.Tensor

Apply mask to the attention energies before passing to softmax.

Parameters

- **energies_4d** – Energies of shape `(batch, n_heads, time(q), time(k))`.
- **mask** – Float Tensor of zeros and ones of shape `(batch, time(k))`, specifies valid positions in the energies tensor.
- **mask_value** – Value used to mask energies. Default taken value from tensor2tensor.

Returns Energies (logits) of valid positions. Same shape as `energies_4d`.

Note: We do not use `mask_value=-np.inf` to avoid potential underflow.

neuralmonkey.attention.scaled_dot_product.**mask_future**(*energies: tensorflow.python.framework.ops.Tensor*, *mask_value=-1000000000.0*) → tensorflow.python.framework.ops.Tensor

Mask energies of keys using lower triangular matrix.

Mask simulates autoregressive decoding, such that it prevents the attention to look at what has not yet been decoded. Mask is not necessary during training when true output values are used instead of the decoded ones.

Parameters

- **energies** – A tensor to mask.
- **mask_value** – Value used to mask energies.

Returns Masked energies tensor.

neuralmonkey.attention.scaled_dot_product.**split_for_heads**(*x: tensorflow.python.framework.ops.Tensor*, *n_heads: int*, *head_dim: int*) → tensorflow.python.framework.ops.Tensor

Split a tensor for multi-head attention.

Split last dimension of 3D vector of shape `(batch, time, dim)` and return a 4D vector with shape `(batch, n_heads, time, dim/n_heads)`.

Parameters

- **x** – input Tensor of shape `(batch, time, dim)`.
- **n_heads** – Number of attention heads.
- **head_dim** – Dimension of the attention heads.

Returns A 4D Tensor of shape `(batch, n_heads, time, head_dim/n_heads)`

neuralmonkey.attention.stateful_context module

class neuralmonkey.attention.stateful_context.**StatefulContext**(*name: str*, *encoder: neuralmonkey.model.stateful.Stateful*, *reuse: neuralmonkey.model.model_part.ModelPart = None*, *save_checkpoint: str = None*, *load_checkpoint: str = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Bases: *neuralmonkey.attention.base_attention.BaseAttention*

Provides a *Stateful* encoder's output as context to a decoder.

This is not really an attention mechanism, but rather a hack which (mis)uses the attention interface to provide a "static" context vector to the decoder cell. In other words, the context vector is the same for all positions in the sequence and doesn't depend on the query vector.

To use this, simply pass an instance of this class to the decoder using the *attentions* parameter.

`__init__`(*name: str, encoder: neuralmonkey.model.stateful.Stateful, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None
Create a new `BaseAttention` object.

`attention`(*query: tensorflow.python.framework.ops.Tensor, decoder_prev_state: tensorflow.python.framework.ops.Tensor, decoder_input: tensorflow.python.framework.ops.Tensor, loop_state: neuralmonkey.attention.namedtuples.AttentionLoopState*) → Tuple[tensorflow.python.framework.ops.Tensor, neuralmonkey.attention.namedtuples.AttentionLoopState]
Get context vector for a given query.

`attention_mask`

`attention_states`

`context_vector_size`
Return the static size of the context vector.

Returns An integer specifying the context vector dimension.

`finalize_loop`(*key: str, last_loop_state: neuralmonkey.attention.namedtuples.AttentionLoopState*) → None
Store the attention histories from loop state under a given key.

Parameters

- **`key`** – The key to the histories dictionary to store the data in.
- **`last_loop_state`** – The loop state object from the last state of the decoding loop.

`initial_loop_state`() → neuralmonkey.attention.namedtuples.AttentionLoopState
Get initial loop state for the attention object.

Returns The newly created initial loop state object.

`state_size`

`visualize_attention`(*key: str, max_outputs: int = 16*) → None
Include the attention histories under a given key into a summary.

Parameters

- **`key`** – The key to the attention histories dictionary.
- **`max_outputs`** – Maximum number of images to save.

neuralmonkey.attention.transformer_cross_layer module

Input combination strategies for multi-source Transformer decoder.

`neuralmonkey.attention.transformer_cross_layer.`**`flat`**(*queries: tensorflow.python.framework.ops.Tensor, encoder_states: List[tensorflow.python.framework.ops.Tensor], encoder_masks: List[tensorflow.python.framework.ops.Tensor], heads: int, attention_dropout_callback: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], dropout_callback: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor]*) → tensorflow.python.framework.ops.Tensor

Run attention with flat input combination.

The procedure is as follows: 1. concatenate the states and mask along the time axis 2. run attention over the concatenation

Parameters

- **`queries`** – The input for the attention.
- **`encoder_states`** – The states of each encoder.
- **`encoder_masks`** – The temporal mask of each encoder.
- **`heads`** – Number of attention heads to use for each encoder.
- **`attention_dropout_callbacks`** – Dropout functions to apply in attention over each encoder.
- **`dropout_callback`** – The dropout function to apply on the output of the attention.

Returns A Tensor that contains the context vector.

`neuralmonkey.attention.transformer_cross_layer.`**`hierarchical`**(*queries: tensorflow.python.framework.ops.Tensor, encoder_states: List[tensorflow.python.framework.ops.Tensor encoder_masks: List[tensorflow.python.framework.ops.Tensor heads: List[int], heads_hier: int, attention_dropout_callbacks: List[Callable[[tensorflow.python.framework.o tensorflow.python.framework.ops.Tensor]], dropout_callback: Callable[[tensorflow.python.framework.ops.T tensorflow.python.framework.ops.Tensor]*) → tensorflow.python.framework.ops.Tensor

Run attention with hierarchical input combination.

The procedure is as follows: 1. normalize queries 2. attend to every encoder 3. attend to the resulting context vectors (reuse normalized queries) 4. apply dropout, add residual connection and return

Parameters

- **queries** – The input for the attention.
- **encoder_states** – The states of each encoder.
- **encoder_masks** – The temporal mask of each encoder.
- **heads** – Number of attention heads to use for each encoder.
- **heads_hier** – Number of attention heads to use in the second attention.
- **attention_dropout_callbacks** – Dropout functions to apply in attention over each encoder.
- **dropout_callback** – The dropout function to apply in the second attention and over the outputs of each sub-attention.

Returns A Tensor that contains the context vector.

neuralmonkey.attention.transformer_cross_layer.**parallel**(*queries: tensorflow.python.framework.ops.Tensor, encoder_states: List[tensorflow.python.framework.ops.Tensor], encoder_masks: List[tensorflow.python.framework.ops.Tensor], heads: List[int], attention_dropout_callbacks: List[Callable[[tensorflow.python.framework.ops.Te* *tensorflow.python.framework.ops.Tensor]], dropout_callback: Callable[[tensorflow.python.framework.ops.Tensor* *tensorflow.python.framework.ops.Tensor]*) → tensorflow.python.framework.ops.Tensor

Run attention with parallel input combination.

The procedure is as follows: 1. normalize queries, 2. attend and dropout independently for every encoder, 3. sum up the results 4. add residual and return

Parameters

- **queries** – The input for the attention.
- **encoder_states** – The states of each encoder.
- **encoder_masks** – The temporal mask of each encoder.
- **heads** – Number of attention heads to use for each encoder.
- **attention_dropout_callbacks** – Dropout functions to apply in attention over each encoder.
- **dropout_callback** – The dropout function to apply on the outputs of each sub-attention.

Returns A Tensor that contains the context vector.

`neuralmonkey.attention.transformer_cross_layer.`**`serial`**(*queries: tensorflow.python.framework.ops.Tensor, encoder_states: List[tensorflow.python.framework.ops.Tensor], encoder_masks: List[tensorflow.python.framework.ops.Tensor], heads: List[int], attention_dropout_callbacks: List[Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor]], dropout_callback: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor]*) → tensorflow.python.framework.ops.Tensor

Run attention with serial input combination.

The procedure is as follows: 1. repeat for every encoder:

- lnorm + attend + dropout + add residual

2. update queries between layers

Parameters

- **`queries`** – The input for the attention.
- **`encoder_states`** – The states of each encoder.
- **`encoder_masks`** – The temporal mask of each encoder.
- **`heads`** – Number of attention heads to use for each encoder.
- **`attention_dropout_callbacks`** – Dropout functions to apply in attention over each encoder.
- **`dropout_callback`** – The dropout function to apply on the outputs of each sub-attention.

Returns A Tensor that contains the context vector.

neuralmonkey.attention.transformer_cross_layer.**single**(*queries: tensorflow.python.framework.ops.Tensor, states: tensorflow.python.framework.ops.Tensor, mask: tensorflow.python.framework.ops.Tensor, n_heads: int, attention_dropout_callback: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], dropout_callback: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], normalize: bool = True, use_dropout: bool = True, residual: bool = True, use_att_transform_bias: bool = False*)

Run attention on a single encoder.

Parameters

- **queries** – The input for the attention.
- **states** – The encoder states (keys & values).
- **mask** – The temporal mask of the encoder.
- **n_heads** – Number of attention heads to use.
- **attention_dropout_callback** – Dropout function to apply in attention.
- **dropout_callback** – Dropout function to apply on the attention output.
- **normalize** – If True, run layer normalization on the queries.
- **use_dropout** – If True, perform dropout on the attention output.
- **residual** – If True, sum the context vector with the input queries.
- **use_att_transform_bias** – If True, enable bias in the attention head projections (for all queries, keys and values).

Returns A Tensor that contains the context vector.

Module contents

neuralmonkey.decoders package

Submodules

neuralmonkey.decoders.autoregressive module

Abstract class for autoregressive decoding.

Either for the recurrent decoder, or for the transformer decoder.

The autoregressive decoder uses the while loop to get the outputs. Descendants should only specify the initial state and the while loop body.

class neuralmonkey.decoders.autoregressive.**AutoregressiveDecoder**(*name: str, vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, max_output_len: int, dropout_keep_prob: float = 1.0, embedding_size: int = None, embeddings_source: neuralmonkey.model.sequence.EmbeddedSeq = None, tie_embeddings: bool = False, label_smoothing: float = None, supress_unk: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart

__init__(*name: str, vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, max_output_len: int, dropout_keep_prob: float = 1.0, embedding_size: int = None, embeddings_source: neuralmonkey.model.sequence.EmbeddedSequence = None, tie_embeddings: bool = False, label_smoothing: float = None, supress_unk: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Initialize parameters common for all autoregressive decoders.

Parameters

- **name** – Name of the decoder. Should be unique accross all Neural Monkey objects.
- **vocabulary** – Target vocabulary.
- **data_id** – Target data series.

- **max_output_len** – Maximum length of an output sequence.
- **reuse** – Reuse the variables from the model part.
- **dropout_keep_prob** – Probability of keeping a value during dropout.
- **embedding_size** – Size of embedding vectors for target words.
- **embeddings_source** – Embedded sequence to take embeddings from.
- **tie_embeddings** – Use decoder.embedding_matrix also in place of the output decoding matrix.
- **label_smoothing** – Label smoothing parameter.
- **supress_unk** – If true, decoder will not produce symbols for unknown tokens.

cost

decoded

decoding_b

decoding_loop (*train_mode: bool*, *sample: bool = False*, *temperature: float = 1*) → Tuple[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor]

Run the decoding while loop.

Calls get_initial_loop_state and constructs tf.while_loop with the continuation criterion returned from loop_continue_criterion, and body function returned from get_body.

After finishing the tf.while_loop, it calls finalize_loop to further postprocess the final decoder loop state (usually by stacking Tensors containing decoding histories).

Parameters

- **train_mode** – Boolean flag, telling whether this is a training run.
- **sample** – Boolean flag, telling whether we should sample the output symbols from the output distribution instead of using argmax or gold data.
- **temperature** – float value specifying the softmax temperature

decoding_w

embedding_matrix

Variables and operations for embedding of input words.

If we are reusing word embeddings, this function takes the embedding matrix from the first encoder

embedding_size

feed_dict (*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]

Populate the feed dictionary for the decoder object.

Parameters

- **dataset** – The dataset to use for the decoder.
- **train** – Boolean flag, telling whether this is a training run.

finalize_loop (*final_loop_state: neuralmonkey.decoders.autoregressive.LoopState*, *train_mode: bool*) → None

Execute post-while loop operations.

Parameters

- **final_loop_state** – Decoder loop state at the end of the decoding loop.
- **train_mode** – Boolean flag, telling whether this is a training run.

get_body (*train_mode: bool*, *sample: bool = False*, *temperature: float = 1*) → Callable
Return the while loop body function.

get_initial_loop_state () → neuralmonkey.decoders.autoregressive.LoopState

get_logits (*state: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor
Project the decoder's output layer to logits over the vocabulary.

go_symbols

loop_continue_criterion (**args*) → tensorflow.python.framework.ops.Tensor
Decide whether to break out of the while loop.

Parameters **loop_state** – `LoopState` instance (see the docs for this module). Represents current decoder loop state.

output_dimension

runtime_logits

runtime_logprobs

runtime_loop_result

runtime_loss

runtime_mask

runtime_output_states

runtime_xents

train_inputs

train_logits

train_logprobs

train_loop_result

train_loss

train_mask

train_output_states

train_xents

class neuralmonkey.decoders.autoregressive.**DecoderConstants**
Bases: *neuralmonkey.decoders.autoregressive.DecoderConstants*

The constants used by an autoregressive decoder.

train_inputs
During training, this is populated by the target token ids.

class neuralmonkey.decoders.autoregressive.**DecoderFeedables**
Bases: *neuralmonkey.decoders.autoregressive.DecoderFeedables*

The input of a single step of an autoregressive decoder.

step
A scalar int tensor, stores the number of the current time step.

finished
A boolean tensor of shape `(batch)`, which says whether the decoding of a sentence in the batch is finished or not. (E.g. whether the end token has already been generated.)

input_symbol
A boolean `batch`-sized tensor with the inputs to the decoder. During inference, this contains the previously generated tokens. During training, this contains the reference tokens.

prev_logits
A tensor of shape `(batch, vocabulary)`. Contains the logits from the previous decoding step.

class neuralmonkey.decoders.autoregressive.**DecoderHistories**
Bases: *neuralmonkey.decoders.autoregressive.DecoderHistories*

The values collected during the run of an autoregressive decoder.

logits
A tensor of shape `(time, batch, vocabulary)` which contains the unnormalized output scores of words in a vocabulary.

decoder_outputs
A tensor of shape `(time, batch, state_size)`. The states of the decoder before the final output (logit) projection.

outputs
An int tensor of shape `(time, batch)`. Stores the generated symbols. (Either an argmax-ed value from the logits, or a target token, during training.)

mask
A float tensor of zeros and ones of shape `(time, batch)`. Keeps track of valid positions in the decoded data.

class neuralmonkey.decoders.autoregressive.**LoopState**
Bases: *neuralmonkey.decoders.autoregressive.LoopState*

The loop state object.

The LoopState is a structure that works with the tf.while_loop function the decoder loop state stores all the information that is not invariant for the decoder run.

histories
A set of tensors that grow in time as the decoder proceeds.

constants
A set of independent tensors that do not change during the entire decoder run.

feedables
A set of tensors used as the input of a single decoder step.

neuralmonkey.decoders.beam_search_decoder module

Beam search decoder.

This module implements the beam search algorithm for autoregressive decoders.

As any autoregressive decoder, this decoder works dynamically, which means it uses the `tf.while_loop` function conditioned on both maximum output length and list of finished hypotheses.

The beam search decoder uses four data strcutures during the decoding process. `SearchState`, `SearchResults`, `BeamSearchLoopState`, and `BeamSearchOutput`. The purpose of these is described in their own docstring.

These structures help the decoder to keep track of the decoding, enabling it to be called e.g. during ensembling, when the content of the structures can be changed and then fed back to the model.

The implementation mimics the API of the `AutoregressiveDecoder` class. There are functions that prepare and return values that are supplied to the `tf.while_loop` function.

class neuralmonkey.decoders.beam_search_decoder.**BeamSearchDecoder**(*name: str*, *parent_decoder: neuralmonkey.decoders.autoregressive.Auto*
beam_size: int, *max_steps: int*, *length_normalization: float*) → None

Bases: `neuralmonkey.model.model_part.ModelPart`

In-graph beam search decoder.

The hypothesis scoring algorithm is taken from https://arxiv.org/pdf/1609.08144.pdf. Length normalization is parameter alpha from equation 14.

__init__(*name: str*, *parent_decoder: neuralmonkey.decoders.autoregressive.AutoregressiveDecoder*, *beam_size: int*, *max_steps: int*, *length_normalization: float*) → None

Construct the beam search decoder graph.

Parameters

- **name** – The name for the model part.
- **parent_decoder** – An autoregressive decoder from which to sample.
- **beam_size** – The number of hypotheses in the beam.
- **max_steps** – The maximum number of time steps to perform.
- **length_normalization** – The alpha parameter from Eq. 14 in the paper.

decoder_state

decoding_loop() → neuralmonkey.decoders.beam_search_decoder.BeamSearchOutput

Create the decoding loop.

This function mimics the behavior of the `decoding_loop` method of the `AutoregressiveDecoder`, except the initial loop state is created outside this method because it is accessed and fed during ensembling.

TODO: The `finalize_loop` method and the handling of attention loop states might be implemented in the future.

Returns This method returns a populated `BeamSearchOutput` object.

expand_to_beam(*val: tensorflow.python.framework.ops.Tensor*, *dim: int = 0*) → tensorflow.python.framework.ops.Tensor

Copy a tensor along a new beam dimension.

Parameters

- **val** – The `Tensor` to expand.
- **dim** – The dimension along which to expand. Usually, the batch axis.

Returns The expanded tensor.

get_body() → Callable[[Any], neuralmonkey.decoders.beam_search_decoder.BeamSearchLoopState]
Return a body function for `tf.while_loop`.

Returns A function that performs a single decoding step.

get_initial_loop_state() → neuralmonkey.decoders.beam_search_decoder.BeamSearchLoopState
Construct the initial loop state for the beam search decoder.

During the construction, the body function of the underlying decoder is called once to retrieve the initial log probabilities of the first token.

The values are initialized as follows:

- **search_state**
  - `logprob_sum` - For each sentence in batch, logprob sum of the first hypothesis in the beam is set to zero while the others are set to negative infinity.
  - `prev_logprobs` - This is the softmax over the logits from the initial decoder step.
  - `lengths` - All zeros.
  - `finshed` - All false.
- **search_results**
  - `scores` - A (batch, beam)-sized tensor of zeros.
  - `token_ids` - A (1, batch, beam)-sized tensor filled with indices of decoder-specific initial input symbols (usually start symbol IDs).
- **decoder_loop_state - The loop state of the underlying** autoregressive decoder, as returned from the initial call to the body function.

Returns A populated `BeamSearchLoopState` structure.

loop_continue_criterion(**args*) → tensorflow.python.framework.ops.Tensor
Decide whether to break out of the while loop.

The criterion for stopping the loop is that either all hypotheses are finished or a maximum number of steps has been reached. Here the number of steps is the number of steps of the underlying decoder minus one, because this function is evaluated after the decoder step has been called and its step has been incremented. This is caused by the fact that we call the decoder body function at the end of the beam body function. (And that, in turn, is to support ensembling.)

Parameters **args** – A `BeamSearchLoopState` instance.

Returns A scalar boolean `Tensor`.

search_results

search_state

vocabulary

class neuralmonkey.decoders.beam_search_decoder.**BeamSearchLoopState**
Bases: *neuralmonkey.decoders.beam_search_decoder.BeamSearchLoopState*

The loop state of the beam search decoder.

A loop state object that is used for transferring data between cycles through the symbolic while loop. It groups together the `SearchState` and `SearchResults` structures and also keeps track of the underlying decoder loop state.

search_state
A `SearchState` object representing the current search state.

search_results
The growing `SearchResults` object which accummulates the outputs of the decoding process.

decoder_loop_state
The current loop state of the underlying autoregressive decoder.

class neuralmonkey.decoders.beam_search_decoder.**BeamSearchOutput**
Bases: *neuralmonkey.decoders.beam_search_decoder.BeamSearchOutput*

The final structure that is returned from the while loop.

last_search_step_output
A populated `SearchResults` object.

last_dec_loop_state
Final loop state of the underlying decoder.

last_search_state
Final loop state of the beam search decoder.

attention_loop_states
The final loop states of the attention objects.

class neuralmonkey.decoders.beam_search_decoder.**SearchResults**
Bases: *neuralmonkey.decoders.beam_search_decoder.SearchResults*

The intermediate results of the beam search decoding.

A cummulative structure that holds the actual decoded tokens and hypotheses scores (after applying a length penalty term).

scores
A `(time, batch, beam)`-shaped tensor with the scores for each hypothesis. The score is computed from the `logprob_sum` of a hypothesis and accounting for the hypothesis length.

token_ids
A `(time, batch, beam)`-shaped tensor with the vocabulary indices of the tokens in each hypothesis.

class neuralmonkey.decoders.beam_search_decoder.**SearchState**
Bases: *neuralmonkey.decoders.beam_search_decoder.SearchState*

Search state of a beam search decoder.

This structure keeps track of a current state of the beam search algorithm. The search state contains tensors that represent hypotheses in the beam, namely their log probability, length, and distribution over the vocabulary when decoding the last word, as well as if the hypothesis is finished or not.

logprob_sum
A `(batch, beam)`-shaped tensor with the sums of token log-probabilities of each hypothesis.

prev_logprobs
A `(batch, beam, vocabulary)`-sized tensor. Stores the log-distribution over the vocabulary from the previous decoding step for each hypothesis.

lengths
A `(batch, beam)`-shaped tensor with the lengths of the hypotheses.

finished
A boolean tensor with shape `(batch, beam)`. Marks finished and unfinished hypotheses.

neuralmonkey.decoders.classifier module

class neuralmonkey.decoders.classifier.**Classifier**(*name: str, encoders: List[neuralmonkey.model.stateful.Stateful], vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, layers: List[int], activation_fn: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor] = <function relu>, dropout_keep_prob: float = 0.5, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart

A simple MLP classifier over encoders.

The API pretends it is an RNN decoder which always generates a sequence of length exactly one.

__init__(*name: str, encoders: List[neuralmonkey.model.stateful.Stateful], vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, layers: List[int], activation_fn: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor] = <function relu>, dropout_keep_prob: float = 0.5, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Construct a new instance of the sequence classifier.

Parameters

- **name** – Name of the decoder. Should be unique accross all Neural Monkey objects
- **encoders** – Input encoders of the decoder
- **vocabulary** – Target vocabulary
- **data_id** – Target data series
- **layers** – List defining structure of the NN. Ini example: layers=[100,20,5] ;creates classifier with hidden layers of

  size 100, 20, 5 and one output layer depending on the size of vocabulary
- **activation_fn** – activation function used on the output of each hidden layer.
- **dropout_keep_prob** – Probability of keeping a value during dropout

cost

decoded

decoded_logits

decoded_seq

feed_dict (*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]

Return a feed dictionary for the given feedable object.

Parameters

- **dataset** – A dataset instance from which to get the data.
- **train** – Boolean indicating whether the model runs in training mode.

Returns A *FeedDict* dictionary object.

loss_with_decoded_ins

loss_with_gt_ins

runtime_logprobs

runtime_loss

train_loss

neuralmonkey.decoders.ctc_decoder module

class neuralmonkey.decoders.ctc_decoder.**CTCDecoder** (*name: str*, *encoder: neuralmonkey.model.stateful.TemporalStateful*, *vocabulary: neuralmonkey.vocabulary.Vocabulary*, *data_id: str*, *max_length: int = None*, *merge_repeated_targets: bool = False*, *merge_repeated_outputs: bool = True*, *beam_width: int = 1*, *reuse: neuralmonkey.model.model_part.ModelPart = None*, *save_checkpoint: str = None*, *load_checkpoint: str = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart

Connectionist Temporal Classification.

See *tf.nn.ctc_loss*, *tf.nn.ctc_greedy_decoder* etc.

__init__ (*name: str*, *encoder: neuralmonkey.model.stateful.TemporalStateful*, *vocabulary: neuralmonkey.vocabulary.Vocabulary*, *data_id: str*, *max_length: int = None*, *merge_repeated_targets: bool = False*, *merge_repeated_outputs: bool = True*, *beam_width: int = 1*, *reuse: neuralmonkey.model.model_part.ModelPart = None*, *save_checkpoint: str = None*, *load_checkpoint: str = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Construct a new parameterized object.

Parameters

- **name** – The name for the model part. Will be used in the variable and name scopes.
- **reuse** – Optional parameterized part with which to share parameters.
- **save_checkpoint** – Optional path to a checkpoint file which will store the parameters of this object.

- **load_checkpoint** – Optional path to a checkpoint file from which to load initial variables for this object.
- **initializers** – An *InitializerSpecs* instance with specification of the initializers.

cost

decoded

feed_dict (*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]
Return a feed dictionary for the given feedable object.

Parameters

- **dataset** – A dataset instance from which to get the data.
- **train** – Boolean indicating whether the model runs in training mode.

Returns A *FeedDict* dictionary object.

logits

runtime_loss

train_loss

train_targets

neuralmonkey.decoders.decoder module

class neuralmonkey.decoders.decoder.**Decoder**(*encoders: List[neuralmonkey.model.stateful.Stateful], vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, name: str, max_output_len: int, dropout_keep_prob: float = 1.0, embedding_size: int = None, embeddings_source: neuralmonkey.model.sequence.EmbeddedSequence = None, tie_embeddings: bool = False, label_smoothing: float = None, rnn_size: int = None, output_projection: Union[Tuple[Callable[[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, List[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], int], Callable[[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, List[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor]] = None, encoder_projection: Callable[[tensorflow.python.framework.ops.Tensor, int, List[neuralmonkey.model.stateful.Stateful]], tensorflow.python.framework.ops.Tensor] = None, attentions: List[neuralmonkey.attention.base_attention.BaseAttention] = None, attention_on_input: bool = False, rnn_cell: str = 'GRU', conditional_gru: bool = False, supress_unk: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.decoders.autoregressive.AutoregressiveDecoder`

A class managing parts of the computation graph used during decoding.

__init__(*encoders: List[neuralmonkey.model.stateful.Stateful], vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, name: str, max_output_len: int, dropout_keep_prob: float = 1.0, embedding_size: int = None, embeddings_source: neuralmonkey.model.sequence.EmbeddedSequence = None, tie_embeddings: bool = False, label_smoothing: float = None, rnn_size: int = None, output_projection: Union[Tuple[Callable[[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, List[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], int], Callable[[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, List[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor]] = None, encoder_projection: Callable[[tensorflow.python.framework.ops.Tensor, int, List[neuralmonkey.model.stateful.Stateful]], tensorflow.python.framework.ops.Tensor] = None, attentions: List[neuralmonkey.attention.base_attention.BaseAttention] = None, attention_on_input: bool = False, rnn_cell: str = 'GRU', conditional_gru: bool = False, supress_unk: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a refactored version of monster decoder.

Parameters

- **encoders** – Input encoders of the decoder.
- **vocabulary** – Target vocabulary.
- **data_id** – Target data series.
- **name** – Name of the decoder. Should be unique accross all Neural Monkey objects.
- **max_output_len** – Maximum length of an output sequence.
- **dropout_keep_prob** – Probability of keeping a value during dropout.
- **embedding_size** – Size of embedding vectors for target words.
- **embeddings_source** – Embedded sequence to take embeddings from.
- **tie_embeddings** – Use decoder.embedding_matrix also in place of the output decoding matrix.
- **rnn_size** – Size of the decoder hidden state, if None set according to encoders.
- **output_projection** – How to generate distribution over vocabulary from decoder_outputs.
- **encoder_projection** – How to construct initial state from encoders.
- **attention** – The attention object to use. Optional.
- **rnn_cell** – RNN Cell used by the decoder (GRU or LSTM).
- **conditional_gru** – Flag whether to use the Conditional GRU architecture.
- **attention_on_input** – Flag whether attention from previous decoding step should be combined with the input in the next step.
- **supress_unk** – If true, decoder will not produce symbols for unknown tokens.
- **reuse** – Reuse the model variables from the given model part.

embed_input_symbol(**args*) → tensorflow.python.framework.ops.Tensor

encoder_projection

finalize_loop(*final_loop_state: neuralmonkey.decoders.autoregressive.LoopState*, *train_mode: bool*) → None
Execute post-while loop operations.

Parameters

- **final_loop_state** – Decoder loop state at the end of the decoding loop.
- **train_mode** – Boolean flag, telling whether this is a training run.

get_body(*train_mode: bool*, *sample: bool = False*, *temperature: float = 1*) → Callable
Return the while loop body function.

get_initial_loop_state() → neuralmonkey.decoders.autoregressive.LoopState

initial_state
Compute initial decoder state.

The part of the computation graph that computes the initial state of the decoder.

input_plus_attention(**args*) → tensorflow.python.framework.ops.Tensor
Merge input and previous attentions.

Input and previous attentions are merged into a single vector of the size fo embedding.

output_dimension

output_projection

output_projection_spec

rnn_size

class neuralmonkey.decoders.decoder.**RNNFeedables**
Bases: *neuralmonkey.decoders.decoder.RNNFeedables*

The feedables used by an RNN-based decoder.

Shares attributes with the `DecoderFeedables` class. The special attributes are listed below.

prev_rnn_state
The recurrent state from the previous step. A tensor of shape `(batch, rnn_size)`

prev_rnn_output
The output of the recurrent network from the previous step. A tensor of shape `(batch, output_size)`

prev_contexts
A list of context vectors returned from attention mechanisms. Tensors of shape `(batch, encoder_state_size)` for each attended encoder.

class neuralmonkey.decoders.decoder.**RNNHistories**
Bases: *neuralmonkey.decoders.decoder.RNNHistories*

The loop state histories for RNN-based decoders.

Shares attributes with the `DecoderHistories` class. The special attributes are listed below.

attention_histories
A list of `AttentionLoopState` objects (or similar) populated by values from the attention mechanisms used in the decoder.

neuralmonkey.decoders.encoder_projection module

Encoder Projection Module.

This module contains different variants of projection of encoders into the initial state of the decoder.

Encoder projections are specified in the configuration file. Each encoder projection function has a unified type `EncoderProjection`, which is a callable that takes three arguments:

1. `train_mode` – boolean tensor specifying whether the train mode is on
2. `rnn_size` – the size of the resulting initial state
3. `encoders` – a list of `Stateful` objects used as the encoders.

To enable further parameterization of encoder projection functions, one can use higher-order functions.

`neuralmonkey.decoders.encoder_projection.`**`concat_encoder_projection`**(*train_mode: tensorflow.python.framework.ops.Tensor, rnn_size: int = None, encoders: List[neuralmonkey.model.stateful.S = None*) → tensorflow.python.framework.ops.Tensor

Concatenate the encoded values of the encoders.

`neuralmonkey.decoders.encoder_projection.`**`empty_initial_state`**(*train_mode: tensorflow.python.framework.ops.Tensor, rnn_size: int, encoders: List[neuralmonkey.model.stateful.Stateful] = None*) → tensorflow.python.framework.ops.Tensor

Return an empty vector.

`neuralmonkey.decoders.encoder_projection.`**`linear_encoder_projection`**(*dropout_keep_prob: float*) → Callable[[tensorflow.python.framew int, List[neuralmonkey.model.stateful.S tensorflow.python.framework.ops.Tensor

Return a linear encoder projection.

Return a projection function which applies dropout on concatenated encoder final states and returns a linear projection to a rnn_size-sized tensor.

Parameters **`dropout_keep_prob`** – The dropout keep probability

`neuralmonkey.decoders.encoder_projection.`**`nematus_projection`**(*dropout_keep_prob: float = 1.0*) → Callable[[tensorflow.python.framework.ops.T int, List[neuralmonkey.model.stateful.Stateful]], tensorflow.python.framework.ops.Tensor]

Return encoder projection used in Nematus.

The initial state is a dense projection with tanh activation computed on the averaged states of the encoders. Dropout is applied to the means (before the projection).

Parameters **`dropout_keep_prob`** – The dropout keep probability.

neuralmonkey.decoders.output_projection module

Output Projection Module.

This module contains different variants of projection functions of decoder outputs into the logit function inputs.

Output projections are specified in the configuration file. Each output projection function has a unified type `OutputProjection`, which is a callable that takes four arguments and returns a tensor:

1. `prev_state` – the hidden state of the decoder.
2. `prev_output` – embedding of the previously decoded word (or train input)
3. `ctx_tensots` – a list of context vectors (for each attention object)

To enable further parameterization of output projection functions, one can use higher-order functions.

`neuralmonkey.decoders.output_projection.`**`maxout_output`**(*maxout_size: int*) → Tuple[Callable[[tensorflow.python.framework.ops.Tensor tensorflow.python.framework.ops.Tensor, List[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], int]

Apply maxout.

Compute RNN output out of the previous state and output, and the context tensors returned from attention mechanisms, as described in the article

This function corresponds to the equations for computation the t_tilde in the Bahdanau et al. (2015) paper, on page 14, with the maxout projection, before the last linear projection.

Parameters **`maxout_size`** – The size of the hidden maxout layer in the deep output

Returns Returns the maxout projection of the concatenated inputs

neuralmonkey.decoders.output_projection.**mlp_output**(*layer_sizes: List[int], activation: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor] = <function tanh>, dropout_keep_prob: float = 1.0*) → Tuple[Callable[[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, List[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], int]

Apply a multilayer perceptron.

Compute RNN deep output using the multilayer perceptron with a specified activation function. (Pascanu et al., 2013 [https://arxiv.org/pdf/1312.6026v5.pdf])

Parameters

- **layer_sizes** – A list of sizes of the hiddel layers of the MLP
- **dropout_keep_prob** – the dropout keep probability
- **activation** – The activation function to use in each layer.

neuralmonkey.decoders.output_projection.**nematus_output**(*output_size: int, activation_fn: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor] = <function tanh>, dropout_keep_prob: float = 1.0*) → Tuple[Callable[[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, List[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], int]

Apply nonlinear one-hidden-layer deep output.

Implementation consistent with Nematus. Can be used instead of (and is in theory equivalent to) nonlinear_output.

Projects the RNN state, embedding of the previously outputted word, and concatenation of all context vectors into a shared vector space, sums them up and apply a hyperbolic tangent activation function.

neuralmonkey.decoders.output_projection.**nonlinear_output**(*output_size: int, activation_fn: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor] = <function tanh>*) → Tuple[Callable[[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, List[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], int]

neuralmonkey.decoders.sequence_labeler module

class neuralmonkey.decoders.sequence_labeler.**SequenceLabeler**(*name: str, encoder: Union[neuralmonkey.encoders.recurrent.RecurrentEncoder, neuralmonkey.encoders.facebook_conv.SentenceEncoder], vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.model.model_part.ModelPart`

Classifier assing a label to each encoder's state.

__init__(*name: str, encoder: Union[neuralmonkey.encoders.recurrent.RecurrentEncoder, neuralmonkey.encoders.facebook_conv.SentenceEncoder], vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Construct a new parameterized object.

Parameters

- **name** – The name for the model part. Will be used in the variable and name scopes.
- **reuse** – Optional parameterized part with which to share parameters.

- **save_checkpoint** – Optional path to a checkpoint file which will store the parameters of this object.
- **load_checkpoint** – Optional path to a checkpoint file from which to load initial variables for this object.
- **initializers** – An *InitializerSpecs* instance with specification of the initializers.

cost

decoded

decoding_b

decoding_residual_w

decoding_w

feed_dict (*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]

Return a feed dictionary for the given feedable object.

Parameters

- **dataset** – A dataset instance from which to get the data.
- **train** – Boolean indicating whether the model runs in training mode.

Returns A *FeedDict* dictionary object.

logits

logprobs

runtime_loss

train_loss

neuralmonkey.decoders.sequence_regressor module

class neuralmonkey.decoders.sequence_regressor.**SequenceRegressor**(*name: str, encoders: List[neuralmonkey.model.stateful.Stat data_id: str, layers: List[int] = None, activation_fn: Callable[[tensorflow.python.framewor tensorflow.python.framework.ops.Tensor] = <function relu>, dropout_keep_prob: float = 1.0, dimension: int = 1, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart

A simple MLP regression over encoders.

The API pretends it is an RNN decoder which always generates a sequence of length exactly one.

__init__(*name: str, encoders: List[neuralmonkey.model.stateful.Stateful], data_id: str, layers: List[int] = None, activation_fn: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor] = <function relu>, dropout_keep_prob: float = 1.0, dimension: int = 1, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Construct a new parameterized object.

Parameters

- **name** – The name for the model part. Will be used in the variable and name scopes.
- **reuse** – Optional parameterized part with which to share parameters.
- **save_checkpoint** – Optional path to a checkpoint file which will store the parameters of this object.
- **load_checkpoint** – Optional path to a checkpoint file from which to load initial variables for this object.
- **initializers** – An *InitializerSpecs* instance with specification of the initializers.

cost

decoded

feed_dict (*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]

Return a feed dictionary for the given feedable object.

Parameters

- **dataset** – A dataset instance from which to get the data.
- **train** – Boolean indicating whether the model runs in training mode.

Returns A *FeedDict* dictionary object.

predictions

runtime_loss

train_loss

neuralmonkey.decoders.transformer module

Implementation of the decoder of the Transformer model.

Described in Vaswani et al. (2017), arxiv.org/abs/1706.03762

class neuralmonkey.decoders.transformer.**TransformerDecoder**(*name: str, encoders: List[Union[neuralmonkey.model.stateful.Tempo neuralmonkey.model.stateful.SpatialStateful]], vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, ff_hidden_size: int, n_heads_self: int, n_heads_enc: Union[List[int], int], depth: int, max_output_len: int, attention_combination_strategy: str = 'serial', n_heads_hier: int = None, dropout_keep_prob: float = 1.0, embedding_size: int = None, embeddings_source: neuralmonkey.model.sequence.EmbeddedSequence = None, tie_embeddings: bool = True, label_smoothing: float = None, self_attention_dropout_keep_prob: float = 1.0, attention_dropout_keep_prob: Union[float, List[float]] = 1.0, use_att_transform_bias: bool = False, supress_unk: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.decoders.autoregressive.AutoregressiveDecoder

__init__(*name: str, encoders: List[Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful]], vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, ff_hidden_size: int, n_heads_self: int, n_heads_enc: Union[List[int], int], depth: int, max_output_len: int, attention_combination_strategy: str = 'serial', n_heads_hier: int = None, dropout_keep_prob: float = 1.0, embedding_size: int = None, embeddings_source: neuralmonkey.model.sequence.EmbeddedSequence = None, tie_embeddings: bool = True, label_smoothing: float = None, self_attention_dropout_keep_prob: float = 1.0, attention_dropout_keep_prob: Union[float, List[float]] = 1.0, use_att_transform_bias: bool = False, supress_unk: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a decoder of the Transformer model.

Described in Vaswani et al. (2017), arxiv.org/abs/1706.03762

Parameters

- **encoders** – Input encoders for the decoder.
- **vocabulary** – Target vocabulary.
- **data_id** – Target data series.
- **name** – Name of the decoder. Should be unique accross all Neural Monkey objects.
- **max_output_len** – Maximum length of an output sequence.
- **dropout_keep_prob** – Probability of keeping a value during dropout.
- **embedding_size** – Size of embedding vectors for target words.
- **embeddings_source** – Embedded sequence to take embeddings from.
- **tie_embeddings** – Use decoder.embedding_matrix also in place of the output decoding matrix.
- **ff_hidden_size** – Size of the feedforward sublayers.
- **n_heads_self** – Number of the self-attention heads.
- **n_heads_enc** – Number of the attention heads over each encoder. Either a list which size must be equal to `encoders`, or a single integer. In the latter case, the number of heads is equal for all encoders.
- **attention_comnbination_strategy** – One of `serial`, `parallel`, `flat`, `hierarchical`. Controls the attention combination strategy for enc-dec attention.
- **n_heads_hier** – Number of the attention heads for the second attention in the `hierarchical` attention combination.
- **depth** – Number of sublayers.
- **label_smoothing** – A label smoothing parameter for cross entropy loss computation.
- **attention_dropout_keep_prob** – Probability of keeping a value during dropout on the attention output.
- **supress_unk** – If true, decoder will not produce symbols for unknown tokens.
- **reuse** – Reuse the variables from the given model part.

dimension

embed_inputs(*inputs: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor

embedded_train_inputs

encoder_attention_sublayer(*queries: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor
Create the encoder-decoder attention sublayer.

feedforward_sublayer(*layer_input: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor
Create the feed-forward network sublayer.

get_body(*train_mode: bool*, *sample: bool = False*, *temperature: float = 1.0*) → Callable
Return the while loop body function.

get_initial_loop_state() → neuralmonkey.decoders.autoregressive.LoopState

layer(*level: int*, *inputs: tensorflow.python.framework.ops.Tensor*, *mask: tensorflow.python.framework.ops.Tensor*) → neuralmonkey.encoders.transformer.TransformerLayer

output_dimension

self_attention_sublayer(*prev_layer: neuralmonkey.encoders.transformer.TransformerLayer*) → tensorflow.python.framework.ops.Tensor
Create the decoder self-attention sublayer with output mask.

train_logits

class neuralmonkey.decoders.transformer.**TransformerHistories**
Bases: *neuralmonkey.decoders.transformer.TransformerHistories*

The loop state histories for the transformer decoder.

Shares attributes with the `DecoderHistories` class. The special attributes are listed below.

decoded_symbols
A tensor which stores the decoded symbols.

input_mask
A float tensor with zeros and ones which marks the valid positions on the input.

neuralmonkey.decoders.word_alignment_decoder module

class neuralmonkey.decoders.word_alignment_decoder.**WordAlignmentDecoder**(*encoder: neuralmonkey.encoders.recurrent.*, *decoder: neuralmonkey.decoders.decoder.D*, *data_id: str*, *name: str*, *reuse: neuralmonkey.model.model_part.I = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart

A decoder that computes soft alignment from an attentive encoder.

Loss is computed as cross-entropy against a reference alignment.

__init__(*encoder: neuralmonkey.encoders.recurrent.RecurrentEncoder*, *decoder: neuralmonkey.decoders.decoder.Decoder*, *data_id: str*, *name: str*, *reuse: neuralmonkey.model.model_part.ModelPart = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Construct a new parameterized object.

Parameters

- **name** – The name for the model part. Will be used in the variable and name scopes.
- **reuse** – Optional parameterized part with which to share parameters.
- **save_checkpoint** – Optional path to a checkpoint file which will store the parameters of this object.
- **load_checkpoint** – Optional path to a checkpoint file from which to load initial variables for this object.
- **initializers** – An *InitializerSpecs* instance with specification of the initializers.

alignment_target

cost

decoded

feed_dict (*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]

Return a feed dictionary for the given feedable object.

Parameters

- **dataset** – A dataset instance from which to get the data.
- **train** – Boolean indicating whether the model runs in training mode.

Returns A *FeedDict* dictionary object.

ref_alignment

runtime_loss

runtime_outputs

train_loss

Module contents

neuralmonkey.encoders package

Submodules

neuralmonkey.encoders.attentive module

class neuralmonkey.encoders.attentive.**AttentiveEncoder** (*name: str, input_sequence: Union[neuralmonkey.model.stateful.TemporalStatefu neuralmonkey.model.stateful.SpatialStateful], hidden_size: int, num_heads: int, output_size: int = None, state_proj_size: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.model.model_part.ModelPart`, `neuralmonkey.model.stateful.TemporalStatefulWithOutput`

An encoder with attention over the input and a fixed-dimension output.

Based on "A Structured Self-attentive Sentence Embedding", https://arxiv.org/abs/1703.03130.

The encoder combines a sequence of vectors into a fixed-size matrix where each row of the matrix is computed using a different attention head. This matrix is exposed as the `temporal_states` property (the time dimension corresponds to the different attention heads). The `output` property provides a flattened and, optionally, projected representation of this matrix.

__init__(*name: str, input_sequence: Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful], hidden_size: int, num_heads: int, output_size: int = None, state_proj_size: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Initialize an instance of the encoder.

attention_weights

output

Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

temporal_mask

Return mask for the temporal_states.

A 2D *Tensor* of shape (batch, time) of type float32 which masks the temporal states so each sequence can have a different length. It should only contain ones or zeros.

temporal_states

Return object states in time.

A 3D *Tensor* of shape (batch, time, state_size) which contains the states of the object in time (e.g. hidden states of a recurrent encoder.

neuralmonkey.encoders.cnn_encoder module

CNN for image processing.

class neuralmonkey.encoders.cnn_encoder.**CNNEncoder**(*name: str, data_id: str, convolutions: List[Union[Tuple[str, int, int, str, int], Tuple[str, int, int], Tuple[str, int, int, str]]], image_height: int, image_width: int, pixel_dim: int, fully_connected: List[int] = None, batch_normalize: bool = False, dropout_keep_prob: float = 0.5, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart, neuralmonkey.model.stateful.SpatialStatefulWithOutput

An image encoder.

It projects the input image through a serie of convolutioal operations. The projected image is vertically cut and fed to stacked RNN layers which encode the image into a single vector.

__init__(*name: str, data_id: str, convolutions: List[Union[Tuple[str, int, int, str, int], Tuple[str, int, int], Tuple[str, int, int, str]]], image_height: int, image_width: int, pixel_dim: int, fully_connected: List[int] = None, batch_normalize: bool = False, dropout_keep_prob: float = 0.5, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Initialize a convolutional network for image processing.

The convolutional network can consist of plain convolutions, max-pooling layers and residual block. In the configuration, they are specified using the following tuples.

- convolution: (“C”, kernel_size, stride, padding, out_channel);
- max / average pooling: (“M”/”A”, kernel_size, stride, padding);
- residual block: (“R”, kernel_size, out_channels).

Padding must be either “valid” or “same”.

Parameters

- **`convolutions`** – Configuration of convolutional layers.
- **`data_id`** – Identifier of the data series in the dataset.
- **`image_height`** – Height of the input image in pixels.
- **`image_width`** – Width of the image.
- **`pixel_dim`** – Number of color channels in the input images.
- **`dropout_keep_prob`** – Probability of keeping neurons active in dropout. Dropout is done between all convolutional layers and fully connected layer.

`batch_norm_callback`(*layer_output: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor

`feed_dict`(*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]

Return a feed dictionary for the given feedable object.

Parameters

- **`dataset`** – A dataset instance from which to get the data.
- **`train`** – Boolean indicating whether the model runs in training mode.

Returns A *FeedDict* dictionary object.

`image_input`

`image_mask`

`image_processing_layers`

Do all convolutions and return the last conditional map.

No dropout is applied between the convolutional layers. By default, the activation function is ReLU.

`output`

Output vector of the CNN.

If there are specified some fully connected layers, there are applied on top of the last convolutional map. Dropout is applied between all layers, default activation function is ReLU. There are only projection layers, no softmax is applied.

If there is fully_connected layer specified, average-pooled last convolutional map is used as a vector output.

`spatial_mask`

Return mask for the spatial_states.

A 3D *Tensor* of shape (batch, width, height) of type float32 which masks the spatial states that they can be of different shapes. The mask should only contain ones or zeros.

spatial_states
Return object states in space.

A 4D *Tensor* of shape (batch, width, height, state_size) which contains the states of the object in space (e.g. final layer of a convolution network processing an image.

class neuralmonkey.encoders.cnn_encoder.**CNNTemporalView**(*name: str*, *cnn: neuralmonkey.encoders.cnn_encoder.CNNEncoder*) → None

Bases: neuralmonkey.model.model_part.ModelPart, neuralmonkey.model.stateful.TemporalStatefulWithOutput

Slice the convolutional maps left to right.

__init__(*name: str*, *cnn: neuralmonkey.encoders.cnn_encoder.CNNEncoder*) → None
Construct a new parameterized object.

Parameters

- **name** – The name for the model part. Will be used in the variable and name scopes.
- **reuse** – Optional parameterized part with which to share parameters.
- **save_checkpoint** – Optional path to a checkpoint file which will store the parameters of this object.
- **load_checkpoint** – Optional path to a checkpoint file from which to load initial variables for this object.
- **initializers** – An *InitializerSpecs* instance with specification of the initializers.

dependencies
Return a list of attribute names regarded as dependents.

output
Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

temporal_mask
Return mask for the temporal_states.

A 2D *Tensor* of shape (batch, time) of type float32 which masks the temporal states so each sequence can have a different length. It should only contain ones or zeros.

temporal_states
Return object states in time.

A 3D *Tensor* of shape (batch, time, state_size) which contains the states of the object in time (e.g. hidden states of a recurrent encoder.

neuralmonkey.encoders.cnn_encoder.**plain_convolution**(*prev_layer: tensorflow.python.framework.ops.Tensor, prev_mask: tensorflow.python.framework.ops.Tensor, specification: Tuple[str, int, int, str, int], batch_norm_callback: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], layer_num: int*) → Tuple[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, int]

neuralmonkey.encoders.cnn_encoder.**pooling**(*prev_layer: tensorflow.python.framework.ops.Tensor, prev_mask: tensorflow.python.framework.ops.Tensor, specification: Tuple[str, int, int, str], layer_num: int*) → Tuple[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor]

neuralmonkey.encoders.cnn_encoder.**residual_block**(*prev_layer: tensorflow.python.framework.ops.Tensor, prev_mask: tensorflow.python.framework.ops.Tensor, prev_channels: int, specification: Tuple[str, int, int], batch_norm_callback: Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor], layer_num: int*) → Tuple[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, int]

neuralmonkey.encoders.facebook_conv module

From the paper Convolutional Sequence to Sequence Learning.

http://arxiv.org/abs/1705.03122

class neuralmonkey.encoders.facebook_conv.**SentenceEncoder**(*name: str*, *input_sequence: neuralmonkey.model.sequence.EmbeddedSequence*, *conv_features: int*, *encoder_layers: int*, *kernel_width: int = 5*, *dropout_keep_prob: float = 1.0*, *reuse: neuralmonkey.model.model_part.ModelPart = None*, *save_checkpoint: str = None*, *load_checkpoint: str = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart, neuralmonkey.model.stateful.TemporalStatefulWithOutput

__init__(*name: str*, *input_sequence: neuralmonkey.model.sequence.EmbeddedSequence*, *conv_features: int*, *encoder_layers: int*, *kernel_width: int = 5*, *dropout_keep_prob: float = 1.0*, *reuse: neuralmonkey.model.model_part.ModelPart = None*, *save_checkpoint: str = None*, *load_checkpoint: str = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Construct a new parameterized object.

Parameters

- **name** – The name for the model part. Will be used in the variable and name scopes.
- **reuse** – Optional parameterized part with which to share parameters.
- **save_checkpoint** – Optional path to a checkpoint file which will store the parameters of this object.
- **load_checkpoint** – Optional path to a checkpoint file from which to load initial variables for this object.
- **initializers** – An *InitializerSpecs* instance with specification of the initializers.

order_embeddings

ordered_embedded_inputs

output

Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

temporal_mask

Return mask for the temporal_states.

A 2D *Tensor* of shape (batch, time) of type float32 which masks the temporal states so each sequence can have a different length. It should only contain ones or zeros.

temporal_states

Return object states in time.

A 3D *Tensor* of shape (batch, time, state_size) which contains the states of the object in time (e.g. hidden states of a recurrent encoder.

neuralmonkey.encoders.imagenet_encoder module

Pre-trained ImageNet networks.

class neuralmonkey.encoders.imagenet_encoder.**ImageNet**(*name: str*, *data_id: str*, *network_type: str*, *slim_models_path: str*, *load_checkpoint: str = None*, *spatial_layer: str = None*, *encoded_layer: str = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.model.model_part.ModelPart`, `neuralmonkey.model.stateful.SpatialStatefulWithOutput`

Pre-trained ImageNet network.

We use the ImageNet networks as they are in the tesnorflow/models repository (https://github.com/tensorflow/models). In order use them, you need to clone the repository and configure the ImageNet object such that it has a full path to "research/slim" in the repository. Visit https://github.com/tensorflow/models/tree/master/research/slim for information about checkpoints of the pre-trained models.

__init__(*name: str*, *data_id: str*, *network_type: str*, *slim_models_path: str*, *load_checkpoint: str = None*, *spatial_layer: str = None*, *encoded_layer: str = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Initialize pre-trained ImageNet network.

Parameters

- **name** – Name of the model part (the ImageNet network, will be in its scope, independently on *name*).
- **data_id** – Id of series with images (list of 3D numpy arrays)
- **network_type** – Identifier of ImageNet network from TFSlim.
- **spatial_layer** – String identifier of the convolutional map (model's endpoint). Check TFSlim documentation for end point specifications.
- **encoded_layer** – String id of the network layer that will be used as input of a decoder. *None* means averaging the convolutional maps.
- **path_to_models** – Path to Slim models in tensorflow/models repository.
- **load_checkpoint** – Checkpoint file from which the pre-trained network is loaded.

feed_dict(*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]

Return a feed dictionary for the given feedable object.

Parameters

- **dataset** – A dataset instance from which to get the data.
- **train** – Boolean indicating whether the model runs in training mode.

Returns A *FeedDict* dictionary object.

input_image

output

Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

spatial_mask
Return mask for the spatial_states.

A 3D *Tensor* of shape (batch, width, height) of type float32 which masks the spatial states that they can be of different shapes. The mask should only contain ones or zeros.

spatial_states
Return object states in space.

A 4D *Tensor* of shape (batch, width, height, state_size) which contains the states of the object in space (e.g. final layer of a convolution network processing an image.

class neuralmonkey.encoders.imagenet_encoder.**ImageNetSpec**
Bases: *neuralmonkey.encoders.imagenet_encoder.ImageNetSpec*

Specification of the Imagenet encoder.

Do not use this object directly, instead, use one of the ‘‘get_*‘‘functions in this module.

scope
The variable scope of the network to use.

image_size
A tuple of two integers giving the image width and height in pixels.

apply_net
The function that receives an image and applies the network.

neuralmonkey.encoders.imagenet_encoder.**get_alexnet**() → neuralmonkey.encoders.imagenet_encoder.ImageNetSpec

neuralmonkey.encoders.imagenet_encoder.**get_resnet_by_type**(*resnet_type: str*) → Callable[[], neuralmonkey.encoders.imagenet_encoder.ImageNetSp

neuralmonkey.encoders.imagenet_encoder.**get_vgg_by_type**(*vgg_type: str*) → Callable[[], neuralmonkey.encoders.imagenet_encoder.ImageNetSpec]

neuralmonkey.encoders.numpy_stateful_filler module

class neuralmonkey.encoders.numpy_stateful_filler.**SpatialFiller**(*name: str, input_shape: List[int], data_id: str, projection_dim: int = None, ff_hidden_dim: int = None, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart, neuralmonkey.model.stateful.SpatialStatefulWithOutput

Placeholder class for 3D numerical input.

This model part is used to feed 3D tensors (e.g., pre-trained convolutional maps image captioning). Optionally, the states are projected to given size.

__init__(*name: str, input_shape: List[int], data_id: str, projection_dim: int = None, ff_hidden_dim: int = None, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Instantiate SpatialFiller.

Parameters

- **name** – Name of the model part.
- **input_shape** – Dimensionality of the input.
- **data_id** – Name of the data series with numpy objects.
- **projection_dim** – Optional, dimension of the states projection.

feed_dict(*dataset: neuralmonkey.dataset.Dataset, train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]

Return a feed dictionary for the given feedable object.

Parameters

- **dataset** – A dataset instance from which to get the data.
- **train** – Boolean indicating whether the model runs in training mode.

Returns A *FeedDict* dictionary object.

output

Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

spatial_mask
Return mask for the spatial_states.

A 3D *Tensor* of shape (batch, width, height) of type float32 which masks the spatial states that they can be of different shapes. The mask should only contain ones or zeros.

spatial_states
Return object states in space.

A 4D *Tensor* of shape (batch, width, height, state_size) which contains the states of the object in space (e.g. final layer of a convolution network processing an image.

class neuralmonkey.encoders.numpy_stateful_filler.**StatefulFiller**(*name: str*, *dimension: int*, *data_id: str*, *output_shape: int = None*, *reuse: neuralmonkey.model.model_part.ModelPart = None*, *save_checkpoint: str = None*, *load_checkpoint: str = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart, neuralmonkey.model.stateful.Stateful

Placeholder class for stateful input.

This model part is used to feed 1D tensors to the model. Optionally, it projects the states to given dimension.

__init__(*name: str*, *dimension: int*, *data_id: str*, *output_shape: int = None*, *reuse: neuralmonkey.model.model_part.ModelPart = None*, *save_checkpoint: str = None*, *load_checkpoint: str = None*, *initializers: List[Tuple[str, Callable]] = None*) → None
Instantiate StatefulFiller.

Parameters

- **name** – Name of the model part.
- **dimension** – Dimensionality of the input.
- **data_id** – Series containing the numpy objects.
- **output_shape** – Dimension of optional state projection.

feed_dict(*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]
Return a feed dictionary for the given feedable object.

Parameters

- **dataset** – A dataset instance from which to get the data.

- **train** – Boolean indicating whether the model runs in training mode.

Returns A *FeedDict* dictionary object.

output

Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

neuralmonkey.encoders.pooling module

class neuralmonkey.encoders.pooling.**SequenceAveragePooling**(*name: str, input_sequence: neuralmonkey.model.stateful.TemporalStateful, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.encoders.pooling.SequencePooling`

An average pooling layer over a sequence.

Averages a sequence over time to produce a single state.

output

Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

class neuralmonkey.encoders.pooling.**SequenceMaxPooling**(*name: str, input_sequence: neuralmonkey.model.stateful.TemporalStateful, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.encoders.pooling.SequencePooling`

A max pooling layer over a sequence.

Takes the maximum of a sequence over time to produce a single state.

output

Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

class neuralmonkey.encoders.pooling.**SequencePooling**(*name: str, input_sequence: neuralmonkey.model.stateful.TemporalStateful, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart, neuralmonkey.model.stateful.Stateful

An abstract pooling layer over a sequence.

__init__(*name: str, input_sequence: neuralmonkey.model.stateful.TemporalStateful, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Initialize an instance of the pooling layer.

neuralmonkey.encoders.raw_rnn_encoder module

class neuralmonkey.encoders.raw_rnn_encoder.**RawRNNEncoder**(*name: str, data_id: str, input_size: int, rnn_layers: List[Union[Tuple[int], Tuple[int, str], Tuple[int, str, str]]], max_input_len: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart, neuralmonkey.model.stateful.TemporalStatefulWithOutput

A raw RNN encoder that gets input as a tensor.

__init__(*name: str, data_id: str, input_size: int, rnn_layers: List[Union[Tuple[int], Tuple[int, str], Tuple[int, str, str]]], max_input_len: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new instance of the encoder.

Parameters

- **data_id** – Identifier of the data series fed to this encoder
- **name** – An unique identifier for this encoder

- **rnn_layers** – A list of tuples specifying the size and, optionally, the direction ('forward', 'backward' or 'bidirectional') and cell type ('GRU' or 'LSTM') of each RNN layer.
- **dropout_keep_prob** – The dropout keep probability (default 1.0)

feed_dict(*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]

Populate the feed dictionary with the encoder inputs.

Parameters

- **dataset** – The dataset to use
- **train** – Boolean flag telling whether it is training time

output

Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

temporal_mask

Return mask for the temporal_states.

A 2D *Tensor* of shape (batch, time) of type float32 which masks the temporal states so each sequence can have a different length. It should only contain ones or zeros.

temporal_states

Return object states in time.

A 3D *Tensor* of shape (batch, time, state_size) which contains the states of the object in time (e.g. hidden states of a recurrent encoder.

neuralmonkey.encoders.recurrent module

class neuralmonkey.encoders.recurrent.**DeepSentenceEncoder**(*name: str, vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, embedding_size: int, rnn_sizes: List[int], rnn_directions: List[str], rnn_cell: str = 'GRU', add_residual: bool = False, max_input_len: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None, embedding_initializer: Callable = None*) → None

Bases: *neuralmonkey.encoders.recurrent.SentenceEncoder*

__init__(*name: str, vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, embedding_size: int, rnn_sizes: List[int], rnn_directions: List[str], rnn_cell: str = 'GRU', add_residual: bool = False, max_input_len: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None, embedding_initializer: Callable = None*) → None

Create a new instance of the deep sentence encoder.

Parameters

- **name** – ModelPart name.
- **vocabulary** – The input vocabulary.
- **data_id** – The input sequence data ID.
- **embedding_size** – The dimension of the embedding vectors in the input sequence.
- **max_input_len** – Maximum length of the input sequence (disregard tokens after this position).
- **rnn_sizes** – The list of dimensions of the RNN hidden state vectors in respective layers.
- **rnn_cell** – One of "GRU", "NematusGRU", "LSTM". Which kind of memory cell to use.
- **rnn_directions** – The list of rnn directions in the respective layers. Should be equally long as *rnn_sizes*. Each item must be one of "forward", "backward", "bidirectional". Determines in what order to process the input sequence. Note that choosing "bidirectional"

will double the resulting vector dimension as well as the number of the parameters in the given layer.

- **add_residual** – Add residual connections to each RNN layer output.
- **dropout_keep_prob** – 1 - dropout probability.
- **save_checkpoint** – ModelPart save checkpoint file.
- **load_checkpoint** – ModelPart load checkpoint file.

rnn

Run stacked RNN given sizes and directions.

Inputs of the first RNN are the RNN inputs to the encoder. Outputs from each layer are used as inputs to the next one. As a final state of the stacked RNN, the final state of the final layer is used.

class neuralmonkey.encoders.recurrent.**FactoredEncoder**(*name: str, vocabularies: List[neuralmonkey.vocabulary.Vocabulary], data_ids: List[str], embedding_sizes: List[int], rnn_size: int, rnn_cell: str = 'GRU', rnn_direction: str = 'bidirectional', add_residual: bool = False, max_input_len: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None, input_initializers: List[Tuple[str, Callable]] = None*) → None

Bases: *neuralmonkey.encoders.recurrent.RecurrentEncoder*

__init__(*name: str, vocabularies: List[neuralmonkey.vocabulary.Vocabulary], data_ids: List[str], embedding_sizes: List[int], rnn_size: int, rnn_cell: str = 'GRU', rnn_direction: str = 'bidirectional', add_residual: bool = False, max_input_len: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None, input_initializers: List[Tuple[str, Callable]] = None*) → None

Create a new instance of the factored encoder.

Parameters

- **name** – ModelPart name.
- **vocabularies** – The vocabularies for each factor.
- **data_ids** – The input sequence data ID for each factor.
- **embedding_sizes** – The dimension of the embedding vectors in the input sequence for each factor.
- **max_input_len** – Maximum length of the input sequence (disregard tokens after this position).
- **rnn_size** – The dimension of the RNN hidden state vector.

- **rnn_cell** – One of "GRU", "NematusGRU", "LSTM". Which kind of memory cell to use.
- **rnn_direction** – One of "forward", "backward", "bidirectional". In what order to process the input sequence. Note that choosing "bidirectional" will double the resulting vector dimension as well as the number of encoder parameters.
- **add_residual** – Add residual connections to the RNN layer output.
- **dropout_keep_prob** – 1 - dropout probability.
- **save_checkpoint** – ModelPart save checkpoint file.
- **load_checkpoint** – ModelPart load checkpoint file.

class `neuralmonkey.encoders.recurrent.`**RNNSpec**

Bases: *neuralmonkey.encoders.recurrent.RNNSpec*

Recurrent neural network specifications.

size

The state size.

direction

The RNN processing direction. One of `forward`, `backward`, and `bidirectional`.

cell_type

The recurrent cell type to use. Refer to `encoders.recurrent.RNN_CELL_TYPES` for possible values.

class `neuralmonkey.encoders.recurrent.`**RecurrentEncoder**(*name: str, input_sequence: neuralmonkey.model.stateful.TemporalStateful, rnn_size: int, rnn_cell: str = 'GRU', rnn_direction: str = 'bidirectional', add_residual: bool = False, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: `neuralmonkey.model.model_part.ModelPart`, `neuralmonkey.model.stateful.TemporalStatefulWithOutput`

__init__(*name: str, input_sequence: neuralmonkey.model.stateful.TemporalStateful, rnn_size: int, rnn_cell: str = 'GRU', rnn_direction: str = 'bidirectional', add_residual: bool = False, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new instance of a recurrent encoder.

Parameters

- **name** – ModelPart name.
- **input_seqeunce** – The input sequence for the encoder.

- **rnn_size** – The dimension of the RNN hidden state vector.
- **rnn_cell** – One of "GRU", "NematusGRU", "LSTM". Which kind of memory cell to use.
- **rnn_direction** – One of "forward", "backward", "bidirectional". In what order to process the input sequence. Note that choosing "bidirectional" will double the resulting vector dimension as well as the number of encoder parameters.
- **add_residual** – Add residual connections to the RNN layer output.
- **dropout_keep_prob** – 1 - dropout probability.
- **save_checkpoint** – ModelPart save checkpoint file.
- **load_checkpoint** – ModelPart load checkpoint file.

output

Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

rnn

rnn_input

temporal_mask

Return mask for the temporal_states.

A 2D *Tensor* of shape (batch, time) of type float32 which masks the temporal states so each sequence can have a different length. It should only contain ones or zeros.

temporal_states

Return object states in time.

A 3D *Tensor* of shape (batch, time, state_size) which contains the states of the object in time (e.g. hidden states of a recurrent encoder.

class neuralmonkey.encoders.recurrent.**SentenceEncoder**(*name: str, vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, embedding_size: int, rnn_size: int, rnn_cell: str = 'GRU', rnn_direction: str = 'bidirectional', add_residual: bool = False, max_input_len: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None, embedding_initializer: Callable = None*) → None

Bases: *neuralmonkey.encoders.recurrent.RecurrentEncoder*

__init__(*name: str, vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, embedding_size: int, rnn_size: int, rnn_cell: str = 'GRU', rnn_direction: str = 'bidirectional', add_residual: bool = False, max_input_len: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None, embedding_initializer: Callable = None*) → None

Create a new instance of the sentence encoder.

Parameters

- **name** – ModelPart name.
- **vocabulary** – The input vocabulary.
- **data_id** – The input sequence data ID.
- **embedding_size** – The dimension of the embedding vectors in the input sequence.
- **max_input_len** – Maximum length of the input sequence (disregard tokens after this position).
- **rnn_size** – The dimension of the RNN hidden state vector.
- **rnn_cell** – One of "GRU", "NematusGRU", "LSTM". Which kind of memory cell to use.
- **rnn_direction** – One of "forward", "backward", "bidirectional". In what order to process the input sequence. Note that choosing "bidirectional" will double the resulting vector dimension as well as the number of encoder parameters.
- **add_residual** – Add residual connections to the RNN layer output.
- **dropout_keep_prob** – 1 - dropout probability.
- **save_checkpoint** – ModelPart save checkpoint file.
- **load_checkpoint** – ModelPart load checkpoint file.

neuralmonkey.encoders.recurrent.**rnn_layer**(*rnn_input: tensorflow.python.framework.ops.Tensor, lengths: tensorflow.python.framework.ops.Tensor, rnn_spec: neuralmonkey.encoders.recurrent.RNNSpec, add_residual: bool*) → Tuple[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor]

Construct a RNN layer given its inputs and specs.

Parameters

- **rnn_inputs** – The input sequence to the RNN.
- **lengths** – Lengths of input sequences.
- **rnn_spec** – A valid RNNSpec tuple specifying the network architecture.
- **add_residual** – Add residual connections to the layer output.

neuralmonkey.encoders.sentence_cnn_encoder module

Encoder for sentences withou explicit segmentation.

class neuralmonkey.encoders.sentence_cnn_encoder.**SentenceCNNEncoder**(*name: str, input_sequence: neuralmonkey.model.sequence.Sequence, segment_size: int, highway_depth: int, rnn_size: int, filters: List[Tuple[int, int]], dropout_keep_prob: float = 1.0, use_noisy_activations: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart, neuralmonkey.model.stateful.TemporalStatefulWithOutput

Recurrent over Convolutional Encoder.

Encoder processing a sentence using a CNN then running a bidirectional RNN on the result.

Based on: Jason Lee, Kyunghyun Cho, Thomas Hofmann: Fully Character-Level Neural Machine Translation without Explicit Segmentation.

See https://arxiv.org/pdf/1610.03017.pdf

__init__(*name: str, input_sequence: neuralmonkey.model.sequence.Sequence, segment_size: int, highway_depth: int, rnn_size: int, filters: List[Tuple[int, int]], dropout_keep_prob: float = 1.0, use_noisy_activations: bool = False, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new instance of the sentence encoder.

Parameters

- **name** – An unique identifier for this encoder
- **segment_size** – The size of the segments over which we apply max-pooling.
- **highway_depth** – Depth of the highway layer.
- **rnn_size** – The size of the encoder's hidden state. Note that the actual encoder output state size will be twice as long because it is the result of concatenation of forward and backward hidden states.
- **filters** – Specification of CNN filters. It is a list of tuples specifying the filter size and number of channels.

Keyword Arguments **dropout_keep_prob** – The dropout keep probability (default 1.0)

bidirectional_rnn

cnn_encoded
1D convolution with max-pool that processing characters.

highway_layer
Highway net projection following the CNN.

output
Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

rnn_cells() → Tuple[tensorflow.python.ops.rnn_cell_impl.RNNCell, tensorflow.python.ops.rnn_cell_impl.RNNCell]
Return the graph template to for creating RNN memory cells.

temporal_mask
Return mask for the temporal_states.

A 2D *Tensor* of shape (batch, time) of type float32 which masks the temporal states so each sequence can have a different length. It should only contain ones or zeros.

temporal_states
Return object states in time.

A 3D *Tensor* of shape (batch, time, state_size) which contains the states of the object in time (e.g. hidden states of a recurrent encoder.

neuralmonkey.encoders.sequence_cnn_encoder module

Encoder for sentence classification with 1D convolutions and max-pooling.

class neuralmonkey.encoders.sequence_cnn_encoder.**SequenceCNNEncoder**(*name: str, vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, embedding_size: int, filters: List[Tuple[int, int]], max_input_len: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart, neuralmonkey.model.stateful.Stateful

Encoder processing a sequence using a CNN.

__init__(*name: str, vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, embedding_size: int, filters: List[Tuple[int, int]], max_input_len: int = None, dropout_keep_prob: float = 1.0, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create a new instance of the CNN sequence encoder.

Based on: Yoon Kim: Convolutional Neural Networks for Sentence Classification (http://emnlp2014.org/papers/pdf/EMNLP2014181.pdf)

Parameters

- **vocabulary** – Input vocabulary
- **data_id** – Identifier of the data series fed to this encoder
- **name** – An unique identifier for this encoder

- **max_input_len** – Maximum length of an encoded sequence
- **embedding_size** – The size of the embedding vector assigned to each word
- **filters** – Specification of CNN filters. It is a list of tuples specifying the filter size and number of channels.
- **dropout_keep_prob** – The dropout keep probability (default 1.0)

embedded_inputs

feed_dict (*dataset: neuralmonkey.dataset.Dataset*, *train: bool = False*) → Dict[tensorflow.python.framework.ops.Tensor, Any]

Populate the feed dictionary with the encoder inputs.

Parameters

- **dataset** – The dataset to use
- **train** – Boolean flag telling whether it is training time

output

Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

neuralmonkey.encoders.transformer module

Implementation of the encoder of the Transformer model.

Described in Vaswani et al. (2017), arxiv.org/abs/1706.03762

class neuralmonkey.encoders.transformer.**TransformerEncoder**(*name: str, input_sequence: neuralmonkey.model.stateful.TemporalStateful, ff_hidden_size: int, depth: int, n_heads: int, dropout_keep_prob: float = 1.0, attention_dropout_keep_prob: float = 1.0, target_space_id: int = None, use_att_transform_bias: bool = False, use_positional_encoding: bool = True, input_for_cross_attention: Union[neuralmonkey.model.stateful.TemporalS neuralmonkey.model.stateful.SpatialStateful] = None, n_cross_att_heads: int = None, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart, neuralmonkey.model.stateful.TemporalStatefulWithOutput

__init__(*name: str, input_sequence: neuralmonkey.model.stateful.TemporalStateful, ff_hidden_size: int, depth: int, n_heads: int, dropout_keep_prob: float = 1.0, attention_dropout_keep_prob: float = 1.0, target_space_id: int = None, use_att_transform_bias: bool = False, use_positional_encoding: bool = True, input_for_cross_attention: Union[neuralmonkey.model.stateful.TemporalStateful, neuralmonkey.model.stateful.SpatialStateful] = None, n_cross_att_heads: int = None, reuse: neuralmonkey.model.model_part.ModelPart = None, save_checkpoint: str = None, load_checkpoint: str = None, initializers: List[Tuple[str, Callable]] = None*) → None

Create an encoder of the Transformer model.

Described in Vaswani et al. (2017), arxiv.org/abs/1706.03762

Parameters

- **input_sequence** – Embedded input sequence.
- **name** – Name of the decoder. Should be unique accross all Neural Monkey objects.
- **reuse** – Reuse the model variables.
- **dropout_keep_prob** – Probability of keeping a value during dropout.

- **target_space_id** – Specifies the modality of the target space.
- **use_att_transform_bias** – Add bias when transforming qkv vectors for attention.
- **use_positional_encoding** – If True, position encoding signal is added to the input.

Keyword Arguments

- **ff_hidden_size** – Size of the feedforward sublayers.
- **n_heads** – Number of the self-attention heads.
- **depth** – Number of sublayers.
- **attention_dropout_keep_prob** – Probability of keeping a value during dropout on the attention output.
- **input_for_cross_attention** – An attendable model part that is attended using cross-attention on every layer of the decoder, analogically to how encoder is attended in the decoder.
- **n_cross_att_heads** – Number of heads used in the cross-attention.

cross_attention_sublayer(*queries: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor

dependencies
Return a list of attribute names regarded as dependents.

encoder_inputs

feedforward_sublayer(*layer_input: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor
Create the feed-forward network sublayer.

layer(*level: int*) → neuralmonkey.encoders.transformer.TransformerLayer

modality_matrix
Create an embedding matrix for varyining target modalities.

Used to embed different target space modalities in the tensor2tensor models (e.g. during the zero-shot translation).

output
Return the object output.

A 2D *Tensor* of shape (batch, state_size) which contains the resulting state of the object.

self_attention_sublayer(*prev_layer: neuralmonkey.encoders.transformer.TransformerLayer*) → tensorflow.python.framework.ops.Tensor
Create the encoder self-attention sublayer.

target_modality_embedding
Gather correct embedding of the target space modality.

See TransformerEncoder.modality_matrix for more information.

temporal_mask
Return mask for the temporal_states.

A 2D *Tensor* of shape (batch, time) of type float32 which masks the temporal states so each sequence can have a different length. It should only contain ones or zeros.

temporal_states
Return object states in time.

A 3D *Tensor* of shape (batch, time, state_size) which contains the states of the object in time (e.g. hidden states of a recurrent encoder.

class neuralmonkey.encoders.transformer.**TransformerLayer**(*states: tensorflow.python.framework.ops.Tensor*, *mask: tensorflow.python.framework.ops.Tensor*) → None

Bases: neuralmonkey.model.stateful.TemporalStateful

__init__(*states: tensorflow.python.framework.ops.Tensor*, *mask: tensorflow.python.framework.ops.Tensor*) → None

Initialize self. See help(type(self)) for accurate signature.

temporal_mask

Return mask for the temporal_states.

A 2D *Tensor* of shape (batch, time) of type float32 which masks the temporal states so each sequence can have a different length. It should only contain ones or zeros.

temporal_states

Return object states in time.

A 3D *Tensor* of shape (batch, time, state_size) which contains the states of the object in time (e.g. hidden states of a recurrent encoder.

neuralmonkey.encoders.transformer.**position_signal**(*dimension: int*, *length: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor

Module contents

neuralmonkey.evaluators package

Submodules

neuralmonkey.evaluators.accuracy module

class neuralmonkey.evaluators.accuracy.**AccuracyEvaluator**(*name: str = None*) → None

Bases: *neuralmonkey.evaluators.evaluator.SequenceEvaluator*

Accuracy Evaluator.

This class uses the default *SequenceEvaluator* implementation, i.e. works on sequences of equal lengths (but can be used to others as well) and use == as the token scorer.

class neuralmonkey.evaluators.accuracy.**AccuracySeqLevelEvaluator**(*name: str = None*) → None

Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

Sequence-level accuracy evaluator.

This class uses the default evaluator implementation. It gives 1.0 to equal sequences and 0.0 to others, averaging the scores over the batch.

neuralmonkey.evaluators.average module

class neuralmonkey.evaluators.average.**AverageEvaluator**(*name: str = None*) → None

Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

Just average the numeric output of a runner.

score_instance(*hypothesis: float, reference: float*) → float

Score a single hyp/ref pair.

The default implementation of this method returns 1.0 when the hypothesis and the reference are equal and 0.0 otherwise.

Parameters

- **hypothesis** – The model prediction.
- **reference** – The golden output.

Returns A float.

neuralmonkey.evaluators.beer module

class neuralmonkey.evaluators.beer.**BeerWrapper**(*wrapper: str, name: str = 'BEER', encoding: str = 'utf-8'*) → None

Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

Wrapper for BEER scorer.

Paper: http://aclweb.org/anthology/D14-1025 Code: https://github.com/stanojevic/beer

__init__(*wrapper: str, name: str = 'BEER', encoding: str = 'utf-8'*) → None

Initialize the BEER wrapper.

Parameters

- **name** – Name of the evaluator.
- **wrapper** – Path to the BEER's executable.
- **encoding** – Data encoding.

score_batch(*hypotheses: List[List[str]], references: List[List[str]]*) → float

Score a batch of hyp/ref pairs.

The default implementation of this method calls *score_instance* for each instance in the batch and returns the average score.

Parameters

- **hypotheses** – List of model predictions.
- **references** – List of golden outputs.

Returns A float.

serialize_to_bytes(*sentences: List[List[str]]*) → bytes

neuralmonkey.evaluators.bleu module

class neuralmonkey.evaluators.bleu.**BLEUEvaluator**(*n: int = 4*, *deduplicate: bool = False*, *name: str = None*, *multiple_references_separator: str = None*) → None

Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

__init__(*n: int = 4*, *deduplicate: bool = False*, *name: str = None*, *multiple_references_separator: str = None*) → None

Instantiate BLEU evaluator.

Parameters

- **n** – Longest n-grams considered.
- **deduplicate** – Flag whether repated tokes should be treated as one.
- **name** – Name displayed in the logs and TensorBoard.
- **multiple_references_separator** – Token that separates multiple reference sentences. If `None`, it assumes the reference is one sentence only.

static bleu(*references: List[List[List[str]]], ngrams: int = 4, case_sensitive: bool = True*)

Compute BLEU on a corpus with multiple references.

The n-grams are uniformly weighted.

Default is to use smoothing as in reference implementation on: https://github.com/ufal/qtleap/blob/master/cuni_train/bin/mteval-v13a.pl#L831-L873

Parameters

- **hypotheses** – List of hypotheses
- **references** – LIst of references. There can be more than one reference.
- **ngrams** – Maximum order of n-grams. Default 4.
- **case_sensitive** – Perform case-sensitive computation. Default True.

static deduplicate_sentences() → List[List[str]]

static effective_reference_length(*references_list: List[List[List[str]]]*) → int

Compute the effective reference corpus length.

The effective reference corpus length is based on best match length.

Parameters

- **hypotheses** – List of output sentences as lists of words
- **references_list** – List of lists of references (as lists of words)

static merge_max_counters() → collections.Counter

Merge counters using maximum values.

static minimum_reference_length(*references_list: List[List[str]]*) → int

Compute the minimum reference corpus length.

The minimum reference corpus length is based on the shortest reference sentence length.

Parameters

- **hypotheses** – List of output sentences as lists of words
- **references_list** – List of lists of references (as lists of words)

static modified_ngram_precision(*references_list: List[List[List[str]]], n: int, case_sensitive: bool*) → Tuple[float, int]

Compute the modified n-gram precision on a list of sentences.

Parameters

- **hypotheses** – List of output sentences as lists of words
- **references_list** – List of lists of reference sentences (as lists of words)
- **n** – n-gram order
- **case_sensitive** – Whether to perform case-sensitive computation

static ngram_counts(*n: int*, *lowercase: bool*, *delimiter: str = ' '*) → collections.Counter

Get n-grams from a sentence.

Parameters

- **sentence** – Sentence as a list of words
- **n** – n-gram order
- **lowercase** – Convert ngrams to lowercase
- **delimiter** – delimiter to use to create counter entries

score_batch(*hypotheses: List[List[str]], references: List[List[str]]*) → float

Score a batch of hyp/ref pairs.

The default implementation of this method calls *score_instance* for each instance in the batch and returns the average score.

Parameters

- **hypotheses** – List of model predictions.
- **references** – List of golden outputs.

Returns A float.

neuralmonkey.evaluators.bleu_ref module

neuralmonkey.evaluators.chrf module

class neuralmonkey.evaluators.chrf.**ChrFEvaluator**(*n: int = 6*, *beta: float = 1.0*, *ignored_symbols: List[str] = None*, *name: str = None*) → None

Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

Compute ChrF score.

See http://www.statmt.org/wmt15/pdf/WMT49.pdf

__init__(*n: int = 6*, *beta: float = 1.0*, *ignored_symbols: List[str] = None*, *name: str = None*) → None

Initialize self. See help(type(self)) for accurate signature.

chr_p(*hyp_ngrams: List[Dict[str, int]], ref_ngrams: List[Dict[str, int]]*) → float

chr_r(*hyp_ngrams: List[Dict[str, int]], ref_ngrams: List[Dict[str, int]]*) → float

score_instance(*hypothesis: List[str], reference: List[str]*) → float

Score a single hyp/ref pair.

The default implementation of this method returns 1.0 when the hypothesis and the reference are equal and 0.0 otherwise.

Parameters

- **hypothesis** – The model prediction.
- **reference** – The golden output.

Returns A float.

neuralmonkey.evaluators.edit_distance module

class neuralmonkey.evaluators.edit_distance.**EditDistanceEvaluator**(*name: str = None*) → None

Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

static compare_scores(*score2: float*) → int

Compare scores using this evaluator.

The default implementation regards the bigger score as better.

Parameters

- **score1** – The first score.
- **score2** – The second score.

Returns An int. When *score1* is better, returns 1. When *score2* is better, returns -1. When the scores are equal, returns 0.

score_batch(*hypotheses: List[List[str]], references: List[List[str]]*) → float

Score a batch of hyp/ref pairs.

The default implementation of this method calls *score_instance* for each instance in the batch and returns the average score.

Parameters

- **hypotheses** – List of model predictions.
- **references** – List of golden outputs.

Returns A float.

score_instance(*hypothesis: List[str], reference: List[str]*) → float

Score a single hyp/ref pair.

The default implementation of this method returns 1.0 when the hypothesis and the reference are equal and 0.0 otherwise.

Parameters

- **hypothesis** – The model prediction.
- **reference** – The golden output.

Returns A float.

neuralmonkey.evaluators.evaluator module

class neuralmonkey.evaluators.evaluator.**Evaluator**(*name: str = None*) → None
Bases: typing.Generic

Base class for evaluators in Neural Monkey.

Each evaluator has a *__call__* method which returns a score for a batch of model predictions given a the references. This class provides default implementations of *score_batch* and *score_instance* functions.

__init__(*name: str = None*) → None
Initialize self. See help(type(self)) for accurate signature.

static compare_scores(*score2: float*) → int
Compare scores using this evaluator.

The default implementation regards the bigger score as better.

Parameters

- **score1** – The first score.
- **score2** – The second score.

Returns An int. When *score1* is better, returns 1. When *score2* is better, returns -1. When the scores are equal, returns 0.

name

score_batch(*hypotheses: List[EvalType], references: List[EvalType]*) → float
Score a batch of hyp/ref pairs.

The default implementation of this method calls *score_instance* for each instance in the batch and returns the average score.

Parameters

- **hypotheses** – List of model predictions.
- **references** – List of golden outputs.

Returns A float.

score_instance(*hypothesis: EvalType, reference: EvalType*) → float
Score a single hyp/ref pair.

The default implementation of this method returns 1.0 when the hypothesis and the reference are equal and 0.0 otherwise.

Parameters

- **hypothesis** – The model prediction.
- **reference** – The golden output.

Returns A float.

class neuralmonkey.evaluators.evaluator.**SequenceEvaluator**(*name: str = None*) → None
Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

Base class for token-level evaluators that work with sequences.

score_batch(*hypotheses: List[Sequence[EvalType]], references: List[Sequence[EvalType]]*) → float
Score batch of sequences.

The default implementation assumes equal sequence lengths and operates on the token level (i.e. token-level scores from the whole batch are averaged (in contrast to averaging each sequence first)).

Parameters

- **hypotheses** – List of model predictions.
- **references** – List of golden outputs.

Returns A float.

score_token(*hyp_token: EvalType, ref_token: EvalType*) → float
Score a single hyp/ref pair of tokens.

The default implementation returns 1.0 if the tokens are equal, 0.0 otherwise.

Parameters

- **hyp_token** – A prediction token.
- **ref_token** – A golden token.

Returns A score for the token hyp/ref pair.

neuralmonkey.evaluators.evaluator.**check_lengths**(*scorer*)

neuralmonkey.evaluators.f1_bio module

class neuralmonkey.evaluators.f1_bio.**F1Evaluator**(*name: str = None*) → None
Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

F1 evaluator for BIO tagging, e.g. NP chunking.

The entities are annotated as beginning of the entity (B), continuation of the entity (I), the rest is outside the entity (O).

static chunk2set() → Set[str]

score_instance(*hypothesis: List[str], reference: List[str]*) → float
Score a single hyp/ref pair.

The default implementation of this method returns 1.0 when the hypothesis and the reference are equal and 0.0 otherwise.

Parameters

- **hypothesis** – The model prediction.
- **reference** – The golden output.

Returns A float.

neuralmonkey.evaluators.gleu module

class neuralmonkey.evaluators.gleu.**GLEUEvaluator**(*n: int = 4, deduplicate: bool = False, name: str = None*) → None
Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

Sentence-level evaluation metric correlating with BLEU on corpus-level.

From "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation" by Wu et al. (https://arxiv.org/pdf/1609.08144v2.pdf)

GLEU is the minimum of recall and precision of all n-grams up to n in references and hypotheses.

Ngram counts are based on the bleu methods.

__init__(*n: int = 4*, *deduplicate: bool = False*, *name: str = None*) → None
Initialize self. See help(type(self)) for accurate signature.

static gleu(*references: List[List[List[str]]], ngrams: int = 4, case_sensitive: bool = True*) → float
Compute GLEU on a corpus with multiple references (no smoothing).

Parameters

- **hypotheses** – List of hypotheses
- **references** – LIst of references. There can be more than one reference.
- **ngrams** – Maximum order of n-grams. Default 4.
- **case_sensitive** – Perform case-sensitive computation. Default True.

score_batch(*hypotheses: List[List[str]], references: List[List[str]]*) → float
Score a batch of hyp/ref pairs.

The default implementation of this method calls *score_instance* for each instance in the batch and returns the average score.

Parameters

- **hypotheses** – List of model predictions.
- **references** – List of golden outputs.

Returns A float.

static total_precision_recall(*references_list: List[List[List[str]]], ngrams: int, case_sensitive: bool*) → Tuple[float, float]
Compute a modified n-gram precision and recall on a sentence list.

Parameters

- **hypotheses** – List of output sentences as lists of words
- **references_list** – List of lists of reference sentences (as lists of words)
- **ngrams** – n-gram order
- **case_sensitive** – Whether to perform case-sensitive computation

neuralmonkey.evaluators.mse module

class neuralmonkey.evaluators.mse.**MeanSquaredErrorEvaluator**(*name: str = None*) → None
Bases: *neuralmonkey.evaluators.evaluator.SequenceEvaluator*

Mean squared error evaluator.

Assumes equal vector length across the batch (see *SequenceEvaluator.score_batch*)

static compare_scores(*score2: float*) → int
Compare scores using this evaluator.

The default implementation regards the bigger score as better.

Parameters

- **score1** – The first score.
- **score2** – The second score.

Returns An int. When *score1* is better, returns 1. When *score2* is better, returns -1. When the scores are equal, returns 0.

score_token (*hyp_elem: float*, *ref_elem: float*) → float
Score a single hyp/ref pair of tokens.

The default implementation returns 1.0 if the tokens are equal, 0.0 otherwise.

Parameters

- **hyp_token** – A prediction token.
- **ref_token** – A golden token.

Returns A score for the token hyp/ref pair.

class neuralmonkey.evaluators.mse.**PairwiseMeanSquaredErrorEvaluator** (*name: str = None*) → None

Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

Pairwise mean squared error evaluator.

For vectors of different dimension across the batch.

static compare_scores (*score2: float*) → int
Compare scores using this evaluator.

The default implementation regards the bigger score as better.

Parameters

- **score1** – The first score.
- **score2** – The second score.

Returns An int. When *score1* is better, returns 1. When *score2* is better, returns -1. When the scores are equal, returns 0.

score_instance (*hypothesis: List[float], reference: List[float]*) → float
Compute mean square error between two vectors.

neuralmonkey.evaluators.multeval module

class neuralmonkey.evaluators.multeval.**MultEvalWrapper** (*wrapper: str*, *name: str = 'MultEval'*, *encoding: str = 'utf-8'*, *metric: str = 'bleu'*, *language: str = 'en'*) → None

Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

Wrapper for mult-eval's reference BLEU and METEOR scorer.

__init__(*wrapper: str*, *name: str = 'MultEval'*, *encoding: str = 'utf-8'*, *metric: str = 'bleu'*, *language: str = 'en'*) → None
Initialize the wrapper.

Parameters

- **wrapper** – Path to multeval.sh script
- **name** – Name of the evaluator
- **encoding** – Encoding of input files
- **language** – Language of hypotheses and references
- **metric** – Evaluation metric "bleu", "ter", "meteor"

score_batch(*hypotheses: List[List[str]], references: List[List[str]]*) → float
Score a batch of hyp/ref pairs.

The default implementation of this method calls *score_instance* for each instance in the batch and returns the average score.

Parameters

- **hypotheses** – List of model predictions.
- **references** – List of golden outputs.

Returns A float.

serialize_to_bytes(*sentences: List[List[str]]*) → bytes

neuralmonkey.evaluators.rouge module

neuralmonkey.evaluators.sacrebleu module

class neuralmonkey.evaluators.sacrebleu.**SacreBLEUEvaluator**(*name: str*, *smooth: str = 'exp'*, *smooth_floor: float = 0.0*, *force: bool = False*, *lowercase: bool = False*, *tokenize: str = 'none'*, *use_effective_order: bool = False*) → None
Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

SacreBLEU evaluator wrapper.

__init__(*name: str*, *smooth: str = 'exp'*, *smooth_floor: float = 0.0*, *force: bool = False*, *lowercase: bool = False*, *tokenize: str = 'none'*, *use_effective_order: bool = False*) → None
Initialize self. See help(type(self)) for accurate signature.

score_batch(*hypotheses: List[List[str]], references: List[List[str]]*) → float
Score a batch of hyp/ref pairs.

The default implementation of this method calls *score_instance* for each instance in the batch and returns the average score.

Parameters

- **hypotheses** – List of model predictions.

- **references** – List of golden outputs.

Returns A float.

neuralmonkey.evaluators.ter module

class neuralmonkey.evaluators.ter.**TEREvaluator**(*name: str = None*) → None
Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

Compute TER using the pyter library.

static compare_scores(*score2: float*) → int
Compare scores using this evaluator.

The default implementation regards the bigger score as better.

Parameters

- **score1** – The first score.
- **score2** – The second score.

Returns An int. When *score1* is better, returns 1. When *score2* is better, returns -1. When the scores are equal, returns 0.

score_instance(*hypothesis: List[str], reference: List[str]*) → float
Score a single hyp/ref pair.

The default implementation of this method returns 1.0 when the hypothesis and the reference are equal and 0.0 otherwise.

Parameters

- **hypothesis** – The model prediction.
- **reference** – The golden output.

Returns A float.

neuralmonkey.evaluators.wer module

class neuralmonkey.evaluators.wer.**WEREvaluator**(*name: str = None*) → None
Bases: *neuralmonkey.evaluators.evaluator.Evaluator*

Compute WER (word error rate, used in speech recognition).

static compare_scores(*score2: float*) → int
Compare scores using this evaluator.

The default implementation regards the bigger score as better.

Parameters

- **score1** – The first score.
- **score2** – The second score.

Returns An int. When *score1* is better, returns 1. When *score2* is better, returns -1. When the scores are equal, returns 0.

score_batch (*hypotheses: List[List[str]], references: List[List[str]]*) → float
Score a batch of hyp/ref pairs.

The default implementation of this method calls *score_instance* for each instance in the batch and returns the average score.

Parameters

- **hypotheses** – List of model predictions.
- **references** – List of golden outputs.

Returns A float.

score_instance (*hypothesis: List[str], reference: List[str]*) → float
Score a single hyp/ref pair.

The default implementation of this method returns 1.0 when the hypothesis and the reference are equal and 0.0 otherwise.

Parameters

- **hypothesis** – The model prediction.
- **reference** – The golden output.

Returns A float.

Module contents

neuralmonkey.runners package

Submodules

neuralmonkey.runners.base_runner module

class neuralmonkey.runners.base_runner.**BaseRunner** (*output_series: str, decoder: MP*) → None
Bases: `neuralmonkey.runners.base_runner.GraphExecutor`, `typing.Generic`

Base class for runners.

Runners are graph executors that retrieve tensors from the model without changing the model parameters. Each runner has a top-level model part it relates to.

class Executable (*executor: Executor, compute_losses: bool, summaries: bool, num_sessions: int*) → None
Bases: `neuralmonkey.runners.base_runner.Executable`

next_to_execute () → Tuple[Union[Dict, List], List[Dict[tensorflow.python.framework.ops.Tensor, Union[int, float, numpy.ndarray]]]]
Get the tensors and additional feed dicts for execution.

__init__ (*output_series: str, decoder: MP*) → None
Initialize self. See help(type(self)) for accurate signature.

decoder_data_id

loss_names

class neuralmonkey.runners.base_runner.**ExecutionResult**

Bases: *neuralmonkey.runners.base_runner.ExecutionResult*

A data structure that represents the result of a graph execution.

The goal of each runner is to populate this structure and set it as its `self._result`.

outputs
A batch of outputs of the runner.

losses
A (possibly empty) list of loss values computed during the run.

scalar_summaries
A TensorFlow summary object with scalar values.

histogram_summaries
A TensorFlow summary object with histograms.

image_summaries
A TensorFlow summary object with images.

class neuralmonkey.runners.base_runner.**GraphExecutor**(*dependencies: Set[neuralmonkey.model.model_part.GenericModelPart*) → None

Bases: `neuralmonkey.model.model_part.GenericModelPart`

The abstract parent class of all graph executors.

In Neural Monkey, a graph executor is an object that retrieves tensors from the computational graph. The two major groups of graph executors are trainers and runners.

Each graph executor is an instance of *GenericModelPart* class, which means it has parameterized and feedable dependencies which reference the model part objects needed to be created in order to compute the tensors of interest (called "fetches").

Every graph executor has a method called *get_executable*, which returns an *GraphExecutor.Executable* instance, which specifies what tensors to execute and collects results from the session execution.

class Executable(*executor: Executor*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None

Bases: `typing.Generic`

Abstract base class for executables.

Executables are objects associated with the graph executors. Each executable has two main functions: *next_to_execute* and *collect_results*. These functions are called in a loop, until the executable's result has been set.

To make use of Mypy's type checking, the executables are generic and are parameterized by the type of their graph executor. Since Python does not know the concept of nested classes, each executable receives the instance of the graph executor through its constructor.

When subclassing *GraphExecutor*, it is also necessary to subclass the *Executable* class and name it *Executable*, so it overrides the definition of this class. Following this guideline, the default implementation of the *get_executable* function on the graph executor will work without the need of overriding it.

__init__(*executor: Executor*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None
Initialize self. See help(type(self)) for accurate signature.

collect_results(*results: List[Dict]*) → None

executor

next_to_execute() → Tuple[Union[Dict, List], List[Dict[tensorflow.python.framework.ops.Tensor, Union[int, float, numpy.ndarray]]]]
Get the tensors and additional feed dicts for execution.

result

set_result(*outputs: List[Any], losses: List[float], scalar_summaries: tensorflow.core.framework.summary_pb2.Summary, histogram_summaries: tensorflow.core.framework.summary_pb2.Summary, image_summaries: tensorflow.core.framework.summary_pb2.Summary*) → None

__init__(*dependencies: Set[neuralmonkey.model.model_part.GenericModelPart]*) → None
Initialize self. See help(type(self)) for accurate signature.

dependencies
Return a list of attribute names regarded as dependents.

feedables

fetches

get_executable(*compute_losses: bool, summaries: bool, num_sessions: int*) → neuralmonkey.runners.base_runner.GraphExecutor.Executable

parameterizeds

neuralmonkey.runners.base_runner.**reduce_execution_results**(*execution_results: List[neuralmonkey.runners.base_runner.Executi*
→ neuralmonkey.runners.base_runner.ExecutionResult
Aggregate execution results into one.

neuralmonkey.runners.beamsearch_runner module

class neuralmonkey.runners.beamsearch_runner.**BeamSearchRunner**(*output_series: str, decoder: neuralmonkey.decoders.beam_search_decoder.Be rank: int = 1, postprocess: Callable[[List[str]], List[str]] = None*) → None
Bases: neuralmonkey.runners.base_runner.BaseRunner

A runner which takes the output from a beam search decoder.

The runner and the beam search decoder support ensembling.

class Executable(*executor: neuralmonkey.runners.beamsearch_runner.BeamSearchRunner, compute_losses: bool, summaries: bool, num_sessions: int*) → None
Bases: neuralmonkey.runners.base_runner.Executable

__init__(*executor: neuralmonkey.runners.beamsearch_runner.BeamSearchRunner, compute_losses: bool, summaries: bool, num_sessions: int*) → None
Initialize self. See help(type(self)) for accurate signature.

collect_results(*results: List[Dict]*) → None

next_to_execute() → Tuple[Union[Dict, List], List[Dict[tensorflow.python.framework.ops.Tensor, Union[int, float, numpy.ndarray]]]]
Get the tensors and additional feed dicts for execution.

prepare_results (*output*)

__init__ (*output_series: str, decoder: neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder, rank: int = 1, postprocess: Callable[[List[str]], List[str]] = None*) → None

Initialize the beam search runner.

Parameters

- **output_series** – Name of the series produced by the runner.
- **decoder** – The beam search decoder to use.
- **rank** – The hypothesis from the beam to select. Setting rank to 1 selects the best hypothesis.
- **postprocess** – The postprocessor to apply to the output data.

fetches

loss_names

neuralmonkey.runners.beamsearch_runner.**beam_search_runner_range** (*output_series: str, decoder: neuralmonkey.decoders.beam_search_decode max_rank: int = None, postprocess: Callable[[List[str]], List[str]] = None*) → List[neuralmonkey.runners.beamsearch

Return beam search runners for a range of ranks from 1 to max_rank.

This means there is max_rank output series where the n-th series contains the n-th best hypothesis from the beam search.

Parameters

- **output_series** – Prefix of output series.
- **decoder** – Beam search decoder shared by all runners.
- **max_rank** – Maximum rank of the hypotheses.
- **postprocess** – Series-level postprocess applied on output.

Returns List of beam search runners getting hypotheses with rank from 1 to max_rank.

neuralmonkey.runners.ctc_debug_runner module

class neuralmonkey.runners.ctc_debug_runner.**CTCDebugRunner** (*output_series: str, decoder: neuralmonkey.decoders.ctc_decoder.CTCDecoder*) → None

Bases: *neuralmonkey.runners.base_runner.BaseRunner*

A runner that print out raw CTC output including the blank symbols.

class Executable (*executor: Executor, compute_losses: bool, summaries: bool, num_sessions: int*) → None

Bases: neuralmonkey.runners.base_runner.Executable

collect_results (*results: List[Dict]*) → None

__init__ (*output_series: str*, *decoder: neuralmonkey.decoders.ctc_decoder.CTCDecoder*) → None
Initialize self. See help(type(self)) for accurate signature.

fetches

loss_names

neuralmonkey.runners.label_runner module

class neuralmonkey.runners.label_runner.**LabelRunner** (*output_series: str*, *decoder: neuralmonkey.decoders.sequence_labeler.SequenceLabeler*, *postprocess: Callable[[List[List[str]]], List[List[str]]] = None*) → None
Bases: `neuralmonkey.runners.base_runner.BaseRunner`

class Executable (*executor: Executor*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None
Bases: `neuralmonkey.runners.base_runner.Executable`

collect_results (*results: List[Dict]*) → None

__init__ (*output_series: str*, *decoder: neuralmonkey.decoders.sequence_labeler.SequenceLabeler*, *postprocess: Callable[[List[List[str]]], List[List[str]]] = None*) → None
Initialize self. See help(type(self)) for accurate signature.

fetches

loss_names

neuralmonkey.runners.logits_runner module

A runner outputing logits or normalized distriution from a decoder.

class neuralmonkey.runners.logits_runner.**LogitsRunner** (*output_series: str*, *decoder: neuralmonkey.decoders.classifier.Classifier*, *normalize: bool = True*, *pick_index: int = None*, *pick_value: str = None*) → None
Bases: `neuralmonkey.runners.base_runner.BaseRunner`

A runner which takes the output from decoder.decoded_logits.

The logits / normalized probabilities are outputted as tab-separates string values. If the decoder produces a list of logits (as the recurrent decoder), the tab separated arrays are separated with commas. Alternatively, we may be interested in a single distribution dimension.

class Executable (*executor: Executor*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None
Bases: `neuralmonkey.runners.base_runner.Executable`

collect_results (*results: List[Dict]*) → None

__init__(*output_series: str, decoder: neuralmonkey.decoders.classifier.Classifier, normalize: bool = True, pick_index: int = None, pick_value: str = None*) → None

Initialize the logits runner.

Parameters

- **output_series** – Name of the series produced by the runner.
- **decoder** – A decoder having logits.
- **normalize** – Flag whether the logits should be normalized with softmax.
- **pick_index** – If not None, it specifies the index of the logit or the probability that should be on output.
- **pick_value** – If not None, it specifies a value from the decoder's vocabulary whose logit or probability should be on output.

fetches

loss_names

neuralmonkey.runners.perplexity_runner module

class neuralmonkey.runners.perplexity_runner.**PerplexityRunner**(*output_series: str, decoder: neuralmonkey.decoders.autoregressive.Autoregre*
→ None

Bases: `neuralmonkey.runners.base_runner.BaseRunner`

class Executable(*executor: Executor, compute_losses: bool, summaries: bool, num_sessions: int*) → None

Bases: neuralmonkey.runners.base_runner.Executable

collect_results(*results: List[Dict]*) → None

__init__(*output_series: str, decoder: neuralmonkey.decoders.autoregressive.AutoregressiveDecoder*) → None

Initialize self. See help(type(self)) for accurate signature.

fetches

loss_names

neuralmonkey.runners.plain_runner module

class neuralmonkey.runners.plain_runner.**PlainRunner**(*output_series: str, decoder: Union[neuralmonkey.decoders.autoregressive.Autoregres neuralmonkey.decoders.ctc_decoder.CTCDecoder, neuralmonkey.decoders.classifier.Classifier, neuralmonkey.decoders.sequence_labeler.SequenceLabeler], postprocess: Callable[[List[List[str]]], List[List[str]]] = None*) → None

Bases: `neuralmonkey.runners.base_runner.BaseRunner`

A runner which takes the output from decoder.decoded.

class Executable(*executor: Executor*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None
Bases: `neuralmonkey.runners.base_runner.Executable`

collect_results(*results: List[Dict]*) → None

__init__(*output_series: str, decoder: Union[neuralmonkey.decoders.autoregressive.AutoregressiveDecoder, neuralmonkey.decoders.ctc_decoder.CTCDecoder, neuralmonkey.decoders.classifier.Classifier, neuralmonkey.decoders.sequence_labeler.SequenceLabeler], postprocess: Callable[[List[List[str]]], List[List[str]]] = None*) → None
Initialize self. See help(type(self)) for accurate signature.

fetches

loss_names

neuralmonkey.runners.regression_runner module

class neuralmonkey.runners.regression_runner.**RegressionRunner**(*output_series: str, decoder: neuralmonkey.decoders.sequence_regressor.Sequu postprocess: Callable[[List[float]], List[float]] = None*) → None
Bases: `neuralmonkey.runners.base_runner.BaseRunner`

A runnner that takes the predictions of a sequence regressor.

class Executable(*executor: Executor*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None
Bases: `neuralmonkey.runners.base_runner.Executable`

collect_results(*results: List[Dict]*) → None

__init__(*output_series: str, decoder: neuralmonkey.decoders.sequence_regressor.SequenceRegressor, postprocess: Callable[[List[float]], List[float]] = None*) → None
Initialize self. See help(type(self)) for accurate signature.

fetches

loss_names

neuralmonkey.runners.runner module

class neuralmonkey.runners.runner.**GreedyRunner**(*output_series: str, decoder: Union[neuralmonkey.decoders.autoregressive.AutoregressiveDec neuralmonkey.decoders.classifier.Classifier], postprocess: Callable[[List[List[str]]], List[List[str]]] = None*) → None
Bases: `neuralmonkey.runners.base_runner.BaseRunner`

class Executable(*executor: Executor*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None
Bases: `neuralmonkey.runners.base_runner.Executable`

collect_results(*results: List[Dict]*) → None

next_to_execute() → Tuple[Union[Dict, List], List[Dict[tensorflow.python.framework.ops.Tensor, Union[int, float, numpy.ndarray]]]]
Get the tensors and additional feed dicts for execution.

__init__(*output_series: str, decoder: Union[neuralmonkey.decoders.autoregressive.AutoregressiveDecoder, neuralmonkey.decoders.classifier.Classifier], postprocess: Callable[[List[List[str]]], List[List[str]]] = None*) → None
Initialize self. See help(type(self)) for accurate signature.

fetches

loss_names

neuralmonkey.runners.tensor_runner module

class neuralmonkey.runners.tensor_runner.**RepresentationRunner**(*output_series: str, encoder: neuralmonkey.model.model_part.GenericModelPart, attribute: str = 'output', select_session: int = None*) → None

Bases: `neuralmonkey.runners.tensor_runner.TensorRunner`

Runner printing out representation from an encoder.

Use this runner to get input / other data representation out from one of Neural Monkey encoders.

__init__(*output_series: str, encoder: neuralmonkey.model.model_part.GenericModelPart, attribute: str = 'output', select_session: int = None*) → None
Initialize the representation runner.

Parameters

- **output_series** – Name of the output series with vectors.
- **encoder** – The encoder to use. This can be any `GenericModelPart` object.
- **attribute** – The name of the encoder attribute that contains the data.
- **used_session** – Id of the TensorFlow session used in case of model ensembles.

class neuralmonkey.runners.tensor_runner.**TensorRunner**(*output_series: str, modelparts: List[neuralmonkey.model.model_part.GenericModelPart], tensors: List[str], batch_dims: List[int], tensors_by_name: List[str], batch_dims_by_name: List[int], select_session: int = None, single_tensor: bool = False*) → None

Bases: `neuralmonkey.runners.base_runner.BaseRunner`

Runner class for printing tensors from a model.

Use this runner if you want to retrieve a specific tensor from the model using a given dataset. The runner generates an output data series which will contain the tensors in a dictionary of numpy arrays.

class Executable(*executor: Executor*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None

Bases: `neuralmonkey.runners.base_runner.Executable`

collect_results(*results: List[Dict]*) → None

__init__(*output_series: str, modelparts: List[neuralmonkey.model.model_part.GenericModelPart], tensors: List[str], batch_dims: List[int], tensors_by_name: List[str], batch_dims_by_name: List[int], select_session: int = None, single_tensor: bool = False*) → None

Construct a new `TensorRunner` object.

Note that at this time, one must specify the toplevel objects so that it is ensured that the graph is built. The reason for this behavior is that the graph is constructed lazily and therefore if the tensors to store are provided by indirect reference (name), the system does not know early enough that it needs to create them.

Parameters

- **output_series** – The name of the generated output data series.
- **modelparts** – A list of `GenericModelPart` objects that hold the tensors that will be retrieved.
- **tensors** – A list of names of tensors that should be retrieved.
- **batch_dims_by_ref** – A list of integers that correspond to the batch dimension in each wanted tensor.
- **tensors_by_name** – A list of tensor names to fetch. If a tensor is not in the graph, a warning is generated and the tensor is ignored.
- **batch_dims_by_name** – A list of integers that correspond to the batch dimension in each wanted tensor specified by name.
- **select_session** – An optional integer specifying the session to use in case of ensembling. When not used, tensors from all sessions are stored. In case of a single session, this option has no effect.
- **single_tensor** – If *True*, it is assumed that only one tensor is to be fetched, and the execution result will consist of this tensor only. If *False*, the result will be a dict mapping tensor names to NumPy arrays.

fetches

loss_names

neuralmonkey.runners.word_alignment_runner module

class neuralmonkey.runners.word_alignment_runner.**WordAlignmentRunner**(*output_series: str, attention: neuralmonkey.attention.base_attention, decoder: neuralmonkey.decoders.decoder.Decod*) → None

Bases: *neuralmonkey.runners.base_runner.BaseRunner*

class Executable(*executor: Executor*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None

Bases: `neuralmonkey.runners.base_runner.Executable`

collect_results(*results: List[Dict]*) → None

__init__(*output_series: str*, *attention: neuralmonkey.attention.base_attention.BaseAttention*, *decoder: neuralmonkey.decoders.decoder.Decoder*) → None

Initialize self. See help(type(self)) for accurate signature.

fetches

loss_names

Module contents

neuralmonkey.trainers package

Submodules

neuralmonkey.trainers.cross_entropy_trainer module

class neuralmonkey.trainers.cross_entropy_trainer.**CrossEntropyTrainer**(*decoders: List[Any], decoder_weights: List[Union[tensorflow.python.framework.ops.Tensor, float, NoneType]] = None, l1_weight: float = 0.0, l2_weight: float = 0.0, clip_norm: float = None, optimizer: tensorflow.python.training.optimizer.Optimizer = None, var_scopes: List[str] = None, var_collection: str = None*) → None

Bases: *neuralmonkey.trainers.generic_trainer.GenericTrainer*

__init__(*decoders: List[Any], decoder_weights: List[Union[tensorflow.python.framework.ops.Tensor, float, NoneType]] = None, l1_weight: float = 0.0, l2_weight: float = 0.0, clip_norm: float = None, optimizer: tensorflow.python.training.optimizer.Optimizer = None, var_scopes: List[str] = None, var_collection: str = None*) → None

Initialize self. See help(type(self)) for accurate signature.

`neuralmonkey.trainers.cross_entropy_trainer.`**`xent_objective`**(*decoder*, *weight=None*) → neuralmonkey.trainers.generic_trainer.Objective

Get XENT objective from decoder with cost.

neuralmonkey.trainers.delayed_update_trainer module

`class` `neuralmonkey.trainers.delayed_update_trainer.`**`DelayedUpdateTrainer`**(*batches_per_update: int, objectives: List[neuralmonkey.trainers.*, *l1_weight: float = 0.0*, *l2_weight: float = 0.0*, *clip_norm: float = None*, *optimizer: tensorflow.python.training.optimiz*, *= None*, *var_scopes: List[str] = None*, *var_collection: str = None*) → None

Bases: *`neuralmonkey.trainers.generic_trainer.GenericTrainer`*

`class Executable`(*executor: neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None

Bases: `neuralmonkey.runners.base_runner.Executable`

`__init__`(*executor: neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer*, *compute_losses: bool*, *summaries: bool*, *num_sessions: int*) → None

Initialize self. See help(type(self)) for accurate signature.

`collect_results`(*results: List[Dict]*) → None

next_to_execute() → Tuple[Union[Dict, List], List[Dict[tensorflow.python.framework.ops.Tensor, Union[int, float, numpy.ndarray]]]]
Get the tensors and additional feed dicts for execution.

__init__(*batches_per_update: int, objectives: List[neuralmonkey.trainers.generic_trainer.Objective], l1_weight: float = 0.0, l2_weight: float = 0.0, clip_norm: float = None, optimizer: tensorflow.python.training.optimizer.Optimizer = None, var_scopes: List[str] = None, var_collection: str = None*) → None
Initialize self. See help(type(self)) for accurate signature.

accumulate_ops

cumulator_counter

diff_buffer

existing_grads_and_vars

gradient_buffers

objective_buffers

raw_gradients
Return averaged gradients over buffers.

reset_ops

summaries

neuralmonkey.trainers.generic_trainer module

class neuralmonkey.trainers.generic_trainer.**GenericTrainer**(*objectives: List[neuralmonkey.trainers.generic_trainer.Obj l1_weight: float = 0.0, l2_weight: float = 0.0, clip_norm: float = None, optimizer: tensorflow.python.training.optimizer.Optimizer = None, var_scopes: List[str] = None, var_collection: str = None*) → None
Bases: *neuralmonkey.runners.base_runner.GraphExecutor*

class Executable(*executor: neuralmonkey.trainers.generic_trainer.GenericTrainer, compute_losses: bool, summaries: bool, num_sessions: int*) → None
Bases: neuralmonkey.runners.base_runner.Executable

__init__(*executor: neuralmonkey.trainers.generic_trainer.GenericTrainer, compute_losses: bool, summaries: bool, num_sessions: int*) → None
Initialize self. See help(type(self)) for accurate signature.

collect_results(*results: List[Dict]*) → None

next_to_execute() → Tuple[Union[Dict, List], List[Dict[tensorflow.python.framework.ops.Tensor, Union[int, float, numpy.ndarray]]]]
Get the tensors and additional feed dicts for execution.

__init__(*objectives: List[neuralmonkey.trainers.generic_trainer.Objective], l1_weight: float = 0.0, l2_weight: float = 0.0, clip_norm: float = None, optimizer: tensorflow.python.training.optimizer.Optimizer = None, var_scopes: List[str] = None, var_collection: str = None*) → None
Initialize self. See help(type(self)) for accurate signature.

static default_optimizer()

differentiable_loss_sum
Compute the differentiable loss (including regularization).

fetches

gradients

objective_values
Compute unweighted losses for fetching.

raw_gradients
Compute the gradients.

regularization_losses
Compute the regularization losses, e.g. L1 and L2.

summaries

train_op
Construct the training op.

var_list

class neuralmonkey.trainers.generic_trainer.**Objective**
Bases: *neuralmonkey.trainers.generic_trainer.Objective*

The training objective.

name
The name for the objective. Used in TensorBoard.

decoder
The decoder which generates the value to optimize.

loss
The loss tensor fetched by the trainer.

gradients
Manually specified gradients. Useful for reinforcement learning.

weight
The weight of this objective. The loss will be multiplied by this so the gradients can be controled in case of multiple objectives.

neuralmonkey.trainers.multitask_trainer module

class neuralmonkey.trainers.multitask_trainer.**MultitaskTrainer**(*trainers: List[neuralmonkey.trainers.generic_train*
→ None
Bases: *neuralmonkey.runners.base_runner.GraphExecutor*

Wrapper for scheduling multitask training.

The wrapper contains a list of trainer objects. They are being called in the order defined by this list thus simulating a task switching schedule.

__init__ (*trainers: List[neuralmonkey.trainers.generic_trainer.GenericTrainer]*) → None
Initialize self. See help(type(self)) for accurate signature.

fetches

get_executable (*compute_losses: bool = True, summaries: bool = True, num_sessions: int = 1*) → neuralmonkey.runners.base_runner.GraphExecutor.Executable

neuralmonkey.trainers.rl_trainer module

Training objectives for reinforcement learning.

neuralmonkey.trainers.rl_trainer.**rl_objective** (*decoder: neuralmonkey.decoders.decoder.Decoder, reward_function: Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray], subtract_baseline: bool = False, normalize: bool = False, temperature: float = 1.0, ce_smoothing: float = 0.0, alpha: float = 1.0, sample_size: int = 1*) → neuralmonkey.trainers.generic_trainer.Objective

Construct RL objective for training with sentence-level feedback.

Depending on the options the objective corresponds to: 1) sample_size = 1, normalize = False, ce_smoothing = 0.0

Bandit objective (Eq. 2) described in 'Bandit Structured Prediction for Neural Sequence-to-Sequence Learning' (http://www.aclweb.org/anthology/P17-1138) It's recommended to set subtract_baseline = True.

2. sample_size > 1, normalize = True, ce_smoothing = 0.0

   Minimum Risk Training as described in 'Minimum Risk Training for Neural Machine Translation' (http://www.aclweb.org/anthology/P16-1159) (Eq. 12).

3. sample_size > 1, normalize = False, ce_smoothing = 0.0

   The Google 'Reinforce' objective as proposed in 'Google's NMT System: Bridging the Gap between Human and Machine Translation' (https://arxiv.org/pdf/1609.08144.pdf) (Eq. 8).

4. sample_size > 1, normalize = False, ce_smoothing > 0.0

   Google's 'Mixed' objective in the above paper (Eq. 9), where ce_smoothing implements alpha.

Note that 'alpha' controls the sharpness of the normalized distribution, while 'temperature' controls the sharpness during sampling.

Parameters

- **decoder** – a recurrent decoder to sample from
- **reward_function** – any evaluator object
- **subtract_baseline** – avg reward is subtracted from obtained reward
- **normalize** – the probabilities of the samples are re-normalized
- **sample_size** – number of samples to obtain feedback for

- **ce_smoothing** – add cross-entropy loss with this coefficient to loss
- **alpha** – determines the shape of the normalized distribution
- **temperature** – the softmax temperature for sampling

Returns Objective object to be used in generic trainer

neuralmonkey.trainers.self_critical_objective module

Training objective for self-critical learning.

Self-critic learning is a modification of the REINFORCE algorithm that uses the reward of the train-time decoder output as a baseline in the update step.

For more details see: https://arxiv.org/pdf/1612.00563.pdf

neuralmonkey.trainers.self_critical_objective.**reinforce_score**(*reward: tensorflow.python.framework.ops.Tensor, baseline: tensorflow.python.framework.ops.Tensor, decoded: tensorflow.python.framework.ops.Tensor, logits: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor

Cost function whose derivative is the REINFORCE equation.

This implements the primitive function to the central equation of the REINFORCE algorithm that estimates the gradients of the loss with respect to decoder logits.

It uses the fact that the second term of the product (the difference of the word distribution and one hot vector of the decoded word) is a derivative of negative log likelihood of the decoded word. The reward function and the baseline are however treated as a constant, so they influence the derivate only multiplicatively.

neuralmonkey.trainers.self_critical_objective.**self_critical_objective**(*decoder: neuralmonkey.decoders.decoder.Deco reward_function: Callable[[numpy.ndarray, numpy.ndarray], numpy.ndarray], weight: float = None*) → neuralmonkey.trainers.generic_traine

Self-critical objective.

Parameters

- **decoder** – A recurrent decoder.

- **reward_function** – A reward function computing score in Python.
- **weight** – Mixing weight for a trainer.

Returns Objective object to be used in generic trainer.

neuralmonkey.trainers.self_critical_objective.**sentence_bleu**(*references: numpy.ndarray*, *hypotheses: numpy.ndarray*) → numpy.ndarray

Compute index-based sentence-level BLEU score.

Computes sentence level BLEU on indices outputed by the decoder, i.e. whatever the decoder uses as a unit is used a token in the BLEU computation, ignoring the tokens may be sub-word units.

neuralmonkey.trainers.self_critical_objective.**sentence_gleu**(*references: numpy.ndarray*, *hypotheses: numpy.ndarray*) → numpy.ndarray

Compute index-based GLEU score.

GLEU score is a sentence-level metric used in Google's Neural MT as a reward in reinforcement learning (https://arxiv.org/abs/1609.08144). It is a minimum of precision and recall on 1- to 4-grams.

It operates over the indices emitted by the decoder which are not necessarily tokens (could be characters or subword units).

neuralmonkey.trainers.test_multitask_trainer module

Unit tests for the multitask trainer.

class neuralmonkey.trainers.test_multitask_trainer.**TestMP**(*name: str*, *reuse: Union[neuralmonkey.model.model_part.ModelP NoneType] = None*, *save_checkpoint: str = None*, *load_checkpoint: str = None*, *initializers: List[Tuple[str, Callable]] = None*) → None

Bases: neuralmonkey.model.model_part.ModelPart

loss

class neuralmonkey.trainers.test_multitask_trainer.**TestMultitaskTrainer**(*methodName='runTest'*)

Bases: unittest.case.TestCase

setUp()

Hook method for setting up the test fixture before exercising it.

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

test_mt_trainer()

Module contents

Submodules

neuralmonkey.checking module

API checking module.

This module serves as a library of API checks used as assertions during constructing the computational graph.

exception neuralmonkey.checking.**CheckingException**
Bases: Exception

neuralmonkey.checking.**assert_same_shape**(*tensor_a: tensorflow.python.framework.ops.Tensor*, *tensor_b: tensorflow.python.framework.ops.Tensor*) → None
Check if two tensors have the same shape.

neuralmonkey.checking.**assert_shape**(*tensor: tensorflow.python.framework.ops.Tensor, expected_shape: List[Union[int, NoneType]]*) → None
Check shape of a tensor.

Parameters

- **tensor** – Tensor to be chcecked.
- **expected_shape** – Expected shape where *None* means the same as in TF and *-1* means not checking the dimension.

neuralmonkey.checking.**check_dataset_and_coders**(*dataset: neuralmonkey.dataset.Dataset*, *runners: Iterable[neuralmonkey.runners.base_runner.BaseRunner]*) → None

neuralmonkey.checkpython module

neuralmonkey.dataset module

Implementation of the dataset class.

class neuralmonkey.dataset.**BatchingScheme**(*batch_size: int*, *batch_bucket_span: int = None*, *token_level_batching: bool = False*, *bucketing_ignore_series: List[str] = None*, *use_leftover_buckets: bool = True*) → None
Bases: object

__init__(*batch_size: int*, *batch_bucket_span: int = None*, *token_level_batching: bool = False*, *bucketing_ignore_series: List[str] = None*, *use_leftover_buckets: bool = True*) → None
Construct the baching scheme.

batch_size
Number of examples in one mini-batch.

batch_bucket_span
The span of the bucket for bucketed batching.

token_level_batching
Count the batch_size per individual tokens in the batch instead of examples.

bucketing_ignore_series
Series to ignore during bucketing.

use_leftover_buckets
Whether to throw out bucket contents at the end of the epoch or to use them.

class neuralmonkey.dataset.**Dataset**(*name: str, iterators: Dict[str, Callable[[], Iterator]], outputs: Dict[str, Tuple[str, Callable[[str, Any], NoneType]]] = None, buffer_size: Tuple[int, int] = None, shuffled: bool = False*) → None

Bases: `object`

Buffered and batched dataset.

This class serves as collection of data series for particular encoders and decoders in the model.

Dataset has a number of data series, which are sequences of data (of any type) that should have the same length. The sequences are loaded in a buffer and can be loaded lazily.

Using the *batches* method, dataset yields batches, through which the data are accessed by the model.

__init__(*name: str, iterators: Dict[str, Callable[[], Iterator]], outputs: Dict[str, Tuple[str, Callable[[str, Any], NoneType]]] = None, buffer_size: Tuple[int, int] = None, shuffled: bool = False*) → None
Construct a new instance of the dataset class.

Do not call this method from the configuration directly. Instead, use the *from_files* function of this module.

The dataset iterators are provided through factory functions, which return the opened iterators when called with no arguments.

Parameters

- **name** – The name for the dataset.
- **iterators** – A series-iterator generator mapping.
- **lazy** – If False, load the data from iterators to a list and store the list in memory.
- **buffer_size** – Use this tuple as a minimum and maximum buffer size for pre-loading data. This should be (a few times) larger than the batch size used for mini-batching. When the buffer size gets under the lower threshold, it is refilled with the new data and optionally reshuffled. If the buffer size is *None*, all data is loaded into memory.
- **shuffled** – Whether to shuffle the buffer during batching.

batches(*scheme: neuralmonkey.dataset.BatchingScheme*) → Iterator[Dataset]
Split the dataset into batches.

Parameters **scheme** – *BatchingScheme* configuration object.

Returns Generator yielding the batches.

get_series(*name: str*) → Iterator
Get the data series with a given name.

Parameters **name** – The name of the series to fetch.

Returns A freshly initialized iterator over the data series.

Raises KeyError if the series does not exists.

has_series(*name: str*) → bool
Check if the dataset contains a series of a given name.

Parameters **name** – Series name

Returns True if the dataset contains the series, False otherwise.

maybe_get_series (*name: str*) → Union[Iterator, NoneType]
Get the data series with a given name, if it exists.

Parameters **name** – The name of the series to fetch.

Returns The data series or None if it does not exist.

series

subset (*start: int*, *length: int*) → neuralmonkey.dataset.Dataset
Create a subset of the dataset.

The sub-dataset will inherit the laziness and buffer size and shuffling from the parent dataset.

Parameters

- **start** – Index of the first data instance in the dataset.
- **length** – Number of instances to include in the subset.

Returns A subset *Dataset* object.

neuralmonkey.dataset.**from_files** (**args*, ***kwargs*)

neuralmonkey.dataset.**load** (*name: str, series: List[str], data: List[Union[str, List[str], Tuple[Union[str, List[str]], Callable[[List[str]], Any]], Tuple[Callable, str], Callable[[Dataset], Iterator[DataType]]]], outputs: List[Union[Tuple[str, str], Tuple[str, str, Callable[[str, Any], NoneType]]]] = None, buffer_size: int = None, shuffled: bool = False*) → neuralmonkey.dataset.Dataset
Create a dataset using specification from the configuration.

The dataset provides iterators over data series. The dataset has a buffer, which pre-fetches a given number of the data series lazily. In case the dataset is not lazy (buffer size is *None*), the iterators are built on top of in-memory arrays. Otherwise, the iterators operate on the data sources directly.

Parameters

- **name** – The name of the dataset.
- **series** – A list of names of data series the dataset contains.
- **data** – The specification of the data sources for each series.
- **outputs** – A list of output specifications.
- **buffer_size** – The size of the buffer. If set, the dataset will be loaded lazily into the buffer (useful for large datasets). The buffer size specifies the number of sequences to pre-load. This is useful for pseudo-shuffling of large data on-the-fly. Ideally, this should be (much) larger than the batch size. Note that the buffer gets refilled each time its size is less than half the *buffer_size*. When refilling, the buffer gets refilled to the specified size.
- **shuffled** – Whether to shuffle the dataset buffer (done upon refill).

neuralmonkey.dataset.**load_dataset_from_files** (*name: str*, *lazy: bool = False*, *preprocessors: List[Tuple[str, str, Callable]] = None*, ***kwargs*) → neuralmonkey.dataset.Dataset
Load a dataset from the files specified by the provided arguments.

Paths to the data are provided in a form of dictionary.

Keyword Arguments

- **name** – The name of the dataset to use. If None (default), the name will be inferred from the file names.
- **lazy** – Boolean flag specifying whether to use lazy loading (useful for large files). Note that the lazy dataset cannot be shuffled. Defaults to False.
- **preprocessor** – A callable used for preprocessing of the input sentences.
- **kwargs** – Dataset keyword argument specs. These parameters should begin with '**s_**' prefix and may end with '_out' suffix. For example, a data series 'source' which specify the source sentences should be initialized with the 's_source' parameter, which specifies the path and optinally reader of the source file. If runners generate data of the 'target' series, the output file should be initialized with the 's_target_out' parameter. Series identifiers should not contain underscores. Dataset-level preprocessors are defined with '**pre_**' prefix followed by a new series name. In case of the pre-processed series, a callable taking the dataset and returning a new series is expected as a value.

Returns The newly created dataset.

neuralmonkey.decorators module

`neuralmonkey.decorators.`**`tensor`**`(`*func*`)`

neuralmonkey.experiment module

Provides a high-level API for training and using a model.

`class` `neuralmonkey.experiment.`**`Experiment`**`(`*config_path: str, train_mode: bool = False, overwrite_output_dir: bool = False, config_changes: List[str] = None*`)` → None

Bases: `object`

`__init__``(`*config_path: str, train_mode: bool = False, overwrite_output_dir: bool = False, config_changes: List[str] = None*`)` → None

Initialize a Neural Monkey experiment.

Parameters

- **config_path** – The path to the experiment configuration file.
- **train_mode** – Indicates whether the model should be prepared for training.
- **overwrite_output_dir** – Indicates whether an existing experiment should be reused. If *True*, this overrides the setting in the configuration file.
- **config_changes** – A list of modifications that will be made to the loaded configuration file before parsing.

`build_model``()` → None

Build the configuration and the computational graph.

This function is invoked by all of the main entrypoints of the *Experiment* class (*train*, *evaluate*, *run*). It manages the building of the TensorFlow graph.

The bulding procedure is executed as follows: 1. Random seeds are set. 2. Configuration is built (instantiated) and normalized. 3. TODO(tf-data) tf.data.Dataset instance is created and registered

in the model parts. (This is not implemented yet!)

4. **Graph executors are "blessed". This causes the rest of the TF Graph** to be built.

5. Sessions are initialized using the TF Manager object.

   Raises *RuntimeError* when the model is already built.

evaluate(*dataset: neuralmonkey.dataset.Dataset*, *write_out: bool = False*, *batch_size: int = None*, *log_progress: int = 0*) → Dict[str, Any]
Run the model on a given dataset and evaluate the outputs.

Parameters

- **dataset** – The dataset on which the model will be executed.
- **write_out** – Flag whether the outputs should be printed to a file defined in the dataset object.
- **batch_size** – size of the minibatch
- **log_progress** – log progress every X seconds

Returns Dictionary of evaluation names and their values which includes the metrics applied on respective series loss and loss values from the run.

classmethod get_current() → neuralmonkey.experiment.Experiment
Return the experiment that is currently being built.

get_initializer(*var_name: str*, *default: Callable = None*) → Union[Callable, NoneType]
Return the initializer associated with the given variable name.

Calling the method marks the given initializer as used.

get_path(*filename: str*, *cont_index: int = None*) → str
Return the path to the most recent version of the given file.

load_variables(*variable_files: List[str] = None*) → None
Load variables from files.

Parameters variable_files – A list of checkpoint file prefixes. A TF checkpoint is usually three files with a common prefix. This list should have the same number of files as there are sessions in the *tf_manager* object.

model
Get configuration namespace of the experiment.

The *Experiment* stores the configuration recipe in *self.config*. When the configuration is built (meaning the classes referenced from the config file are instantiated), it is saved in the *model* property of the experiment.

Returns The built namespace config object.

Raises *RuntimeError* when the configuration model has not been built.

run_model(*dataset: neuralmonkey.dataset.Dataset*, *write_out: bool = False*, *batch_size: int = None*, *log_progress: int = 0*) → Tuple[List[neuralmonkey.runners.base_runner.ExecutionResult], Dict[str, List[Any]]]
Run the model on a given dataset.

Parameters

- **dataset** – The dataset on which the model will be executed.
- **write_out** – Flag whether the outputs should be printed to a file defined in the dataset object.
- **batch_size** – size of the minibatch
- **log_progress** – log progress every X seconds

Returns A list of ‘ExecutionResult‘s and a dictionary of the output series.

train() → None
Train model specified by this experiment.

This function is one of the main functions (entrypoints) called on the experiment. It builds the model (if needed) and runs the training procedure.

Raises *RuntimeError* when the experiment is not intended for training.

update_initializers(*initializers: Iterable[Tuple[str, Callable]]*) → None
Update the dictionary mapping variable names to initializers.

neuralmonkey.experiment.**create_config**(*train_mode: bool = True*) → neuralmonkey.config.configuration.Configuration

neuralmonkey.experiment.**save_git_info**(*git_commit_file: str*, *git_diff_file: str*, *branch: str = 'HEAD'*, *repo_dir: str = None*) → None

neuralmonkey.experiment.**visualize_embeddings**(*sequences: List[neuralmonkey.model.sequence.EmbeddedFactorSeq*
output_dir: str) → None

neuralmonkey.functions module

Collection of various functions and function wrappers.

neuralmonkey.functions.**inverse_sigmoid_decay**(*param*, *rate*, *min_value: float = 0.0*, *max_value: float = 1.0*, *name: Union[str, NoneType] = None*, *dtype=tf.float32*) → tensorflow.python.framework.ops.Tensor
Compute an inverse sigmoid decay: k/(k+exp(x/k)).

The result will be scaled to the range (min_value, max_value).

Parameters

- **param** – The parameter x from the formula.
- **rate** – Non-negative k from the formula.

neuralmonkey.functions.**noam_decay**(*learning_rate: float*, *model_dimension: int*, *warmup_steps: int*) → tensorflow.python.framework.ops.Tensor
Return decay function as defined in Vaswani et al., 2017, Equation 3.

https://arxiv.org/abs/1706.03762

lrate = (d_model)^-0.5 * min(step_num^-0.5, step_num * warmup_steps^-1.5)

Parameters

- **model_dimension** – Size of the hidden states of decoder and encoder
- **warmup_steps** – Number of warm-up steps

neuralmonkey.functions.**piecewise_function**(*param*, *values*, *changepoints*, *name=None*, *dtype=tf.float32*)
Compute a piecewise function.

Parameters

- **param** – The function parameter.
- **values** – List of function values (numbers or tensors).

- **changepoints** – Sorted list of points where the function changes from one value to the next. Must be one item shorter than *values*.

neuralmonkey.learning_utils module

`neuralmonkey.learning_utils.`**`evaluation`**(*evaluators*, *batch*, *runners*, *execution_results*, *result_data*)

Evaluate the model outputs.

Parameters

- **evaluators** – List of tuples of series and evaluation functions.
- **batch** – Batch of data against which the evaluation is done.
- **runners** – List of runners (contains series ids and loss names).
- **execution_results** – Execution results that include the loss values.
- **result_data** – Dictionary from series names to list of outputs.

Returns Dictionary of evaluation names and their values which includes the metrics applied on respective series loss and loss values from the run.

`neuralmonkey.learning_utils.`**`print_final_evaluation`**(*name: str*, *eval_result: Dict[str, float]*) → None

Print final evaluation from a test dataset.

`neuralmonkey.learning_utils.`**`run_on_dataset`**(*tf_manager: neuralmonkey.tf_manager.TensorFlowManager*, *runners: List[neuralmonkey.runners.base_runner.BaseRunner]*, *dataset: neuralmonkey.dataset.Dataset*, *postprocess: Union[List[Tuple[str, Callable]], NoneType]*, *batching_scheme: neuralmonkey.dataset.BatchingScheme*, *write_out: bool = False*, *log_progress: int = 0*) → Tuple[List[neuralmonkey.runners.base_runner.ExecutionResult], Dict[str, List[Any]]]

Apply the model on a dataset and optionally write outputs to files.

This function processes the dataset in batches and optionally prints out the execution progress.

Parameters

- **tf_manager** – TensorFlow manager with initialized sessions.
- **runners** – A function that runs the code
- **dataset** – The dataset on which the model will be executed.
- **evaluators** – List of evaluators that are used for the model evaluation if the target data are provided.
- **postprocess** – Dataset-level postprocessors
- **write_out** – Flag whether the outputs should be printed to a file defined in the dataset object.
- **batching_scheme** – Scheme used for batching.
- **log_progress** – log progress every X seconds
- **extra_fetches** – Extra tensors to evaluate for each batch.

Returns Tuple of resulting sentences/numpy arrays, and evaluation results if they are available which are dictionary function -> value.

`neuralmonkey.learning_utils.`**`training_loop`**(*tf_manager: neuralmonkey.tf_manager.TensorFlowManager, epochs: int, trainers: List[Union[neuralmonkey.trainers.generic_trainer.GenericTrainer, neuralmonkey.trainers.multitask_trainer.MultitaskTrainer]], batching_scheme: neuralmonkey.dataset.BatchingScheme, runners_batching_scheme: neuralmonkey.dataset.BatchingScheme, log_directory: str, evaluators: List[Union[Tuple[str, Any], Tuple[str, str, Any]]], main_metric: str, runners: List[neuralmonkey.runners.base_runner.BaseRunner], train_dataset: neuralmonkey.dataset.Dataset, val_datasets: List[neuralmonkey.dataset.Dataset], test_datasets: Union[List[neuralmonkey.dataset.Dataset], NoneType], log_timer: Callable[[int, float], bool], val_timer: Callable[[int, float], bool], val_preview_input_series: Union[List[str], NoneType], val_preview_output_series: Union[List[str], NoneType], val_preview_num_examples: int, postprocess: Union[List[Tuple[str, Callable]], NoneType], train_start_offset: int, initial_variables: Union[str, List[str], NoneType], final_variables: str*) → None

Execute the training loop for given graph and data.

Parameters

- **`tf_manager`** – TensorFlowManager with initialized sessions.
- **`epochs`** – Number of epochs for which the algoritm will learn.
- **`trainer`** – The trainer object containg the TensorFlow code for computing the loss and optimization operation.
- **`batch_size`** – Number of examples in one mini-batch.
- **`batching_scheme`** – Batching scheme specification. Cannot be provided when batch_size is specified.
- **`log_directory`** – Directory where the TensordBoard log will be generated. If None, nothing will be done.
- **`evaluators`** – List of evaluators. The last evaluator is used as the main. An evaluator is a tuple of the name of the generated series, the name of the dataset series the generated one is evaluated with and the evaluation function. If only one series names is provided, it means the generated and dataset series have the same name.
- **`runners`** – List of runners for logging and evaluation runs
- **`train_dataset`** – Dataset used for training
- **`val_dataset`** – used for validation. Can be Dataset or a list of datasets. The last dataset is used as the main one for storing best results. When using multiple datasets. It is recommended to name them for better Tensorboard visualization.

- **test_datasets** – List of datasets used for testing
- **logging_period** – after how many batches should the logging happen. It can also be defined as a time period in format like: 3s; 4m; 6h; 1d; 3m15s; 3seconds; 4minutes; 6hours; 1days
- **validation_period** – after how many batches should the validation happen. It can also be defined as a time period in same format as logging
- **val_preview_input_series** – which input series to preview in validation
- **val_preview_output_series** – which output series to preview in validation
- **val_preview_num_examples** – how many examples should be printed during validation
- **train_start_offset** – how many lines from the training dataset should be skipped. The training starts from the next batch.
- **runners_batch_size** – batch size of runners. Reuses the training batching scheme with bucketing turned off.
- **initial_variables** – variables used for initialization, for example for continuation of training. Provide it with a path to your model directory and its checkpoint file group common prefix, e.g. "variables.data", or "variables.data.3" in case of multiple checkpoints per experiment.
- **postprocess** – A function which takes the dataset with its output series and generates additional series from them.

neuralmonkey.logging module

class neuralmonkey.logging.**Logging**
Bases: object

static debug(*label: str = None*)

debug_disabled_for = ['']

static debug_enabled()

debug_enabled_for = ['none']

static log(*color: str = 'yellow'*) → None
Log a message with a colored timestamp.

log_file = None

static log_print() → None
Print a string both to console and a log file is it is defined.

static notice() → None
Log a notice with a colored timestamp.

static print_header(*path: str*) → None
Print the title of the experiment and a set of arguments it uses.

static set_log_file() → None
Set up the file where the logging will be done.

strict_mode = ''

static warn() → None
Log a warning.

neuralmonkey.logging.**debug**(*message: str*, *label: str = None*)

neuralmonkey.logging.**debug_enabled**(*label: str = None*)

neuralmonkey.logging.**log**(*message: str*, *color: str = 'yellow'*) → None
Log a message with a colored timestamp.

neuralmonkey.logging.**log_print**(*text: str*) → None
Print a string both to console and a log file is it is defined.

neuralmonkey.logging.**notice**(*message: str*) → None
Log a notice with a colored timestamp.

neuralmonkey.logging.**warn**(*message: str*) → None
Log a warning.

neuralmonkey.run module

neuralmonkey.run.**load_runtime_config**(*config_path: str*) → argparse.Namespace
Load a runtime configuration file.

neuralmonkey.run.**main**() → None

neuralmonkey.tf_manager module

TensorFlow Manager.

TensorFlow manager is a helper object in Neural Monkey which manages TensorFlow sessions, execution of the computation graph, and saving and restoring of model variables.

class neuralmonkey.tf_manager.**TensorFlowManager**(*num_sessions: int*, *num_threads: int*, *save_n_best: int = 1*, *minimize_metric: bool = False*, *gpu_allow_growth: bool = True*, *per_process_gpu_memory_fraction: float = 1.0*, *enable_tf_debug: bool = False*) → None

Bases: object

Inteface between computational graph, data and TF sessions.

sessions
List of active Tensorflow sessions.

__init__(*num_sessions: int*, *num_threads: int*, *save_n_best: int = 1*, *minimize_metric: bool = False*, *gpu_allow_growth: bool = True*, *per_process_gpu_memory_fraction: float = 1.0*, *enable_tf_debug: bool = False*) → None
Initialize a TensorflowManager.

At this moment the graph must already exist. This method initializes required number of TensorFlow sessions and initializes them with provided variable files if they are provided.

Parameters

- **num_sessions** – Number of sessions to be initialized.
- **num_threads** – Number of threads sessions will run in.

- **save_n_best** – How many best models to keep
- **minimize_metric** – Whether the best model is the one with the lowest or the highest score
- **gpu_allow_growth** – TF to allocate incrementally, not all at once.
- **per_process_gpu_memory_fraction** – Limit TF memory use.

best_vars_file

execute(*batch: neuralmonkey.dataset.Dataset, feedables: Set[neuralmonkey.model.feedable.Feedable], runners: Sequence[neuralmonkey.runners.base_runner.GraphExecutor], train: bool = False, compute_losses: bool = True, summaries: bool = True*) → List[neuralmonkey.runners.base_runner.ExecutionResult]

Execute runners on a batch of data.

First, extract executables from the provided runners, telling the runners whether to compute also losses and summaries. Second, until all executables are satisfied (have the *result* attribute set), run the executables on the batch.

Parameters

- **batch** – A batch of data.
- **execution_scripts** – List of runners to execute.
- **train** – Training mode flag (this value is fed to the *train_mode* placeholders in model parts).
- **compute_losses** – Flag to runners whether run loss operations.
- **summaries** – Flag to runners whether to run summary operations.

Returns A list of *ExecutionResult* tuples, one for each executable (runner).

init_saving(*vars_prefix: str*) → None

initialize_model_parts(*runners: Sequence[neuralmonkey.runners.base_runner.GraphExecutor], save: bool = False*) → None

Initialize model parts variables from their checkpoints.

initialize_sessions() → None

restore(*variable_files: Union[str, List[str]]*) → None

restore_best_vars() → None

save(*variable_files: Union[str, List[str]]*) → None

validation_hook(*score: float, epoch: int, batch: int*) → None

neuralmonkey.tf_manager.**get_default_tf_manager**() → neuralmonkey.tf_manager.TensorFlowManager

neuralmonkey.tf_utils module

A set of helper functions for TensorFlow.

neuralmonkey.tf_utils.**append_tensor**(*tensor: tensorflow.python.framework.ops.Tensor, appendval: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor

Append an `N`-D Tensor to an `(N+1)`-D Tensor.

Parameters

- **tensor** – The original Tensor
- **appendval** – The Tensor to add

Returns An `(N+1)`-D Tensor with `appendval` on the last position.

`neuralmonkey.tf_utils.`**`gather_flat`**(*x: tensorflow.python.framework.ops.Tensor*, *indices: tensorflow.python.framework.ops.Tensor*, *batch_size: Union[int, tensorflow.python.framework.ops.Tensor] = 1*, *beam_size: Union[int, tensorflow.python.framework.ops.Tensor] = 1*) → tensorflow.python.framework.ops.Tensor

Gather values from the flattened (shape=[batch * beam, . . .]) input.

This function expects a flattened tensor with first dimension of size *batch x beam* elements. Using the given batch and beam size, it reshapes the input tensor to a tensor of shape `(batch, beam, ...)` and gather the values from it using the index tensor.

Parameters

- **x** – A flattened `Tensor` from which to gather values.
- **indices** – Index tensor.
- **batch_size** – The size of the batch.
- **beam_size** – The size of the beam.

Returns The `Tensor` of gathered values.

`neuralmonkey.tf_utils.`**`get_initializer`**(*var_name: str*, *default: Callable = None*) → Union[Callable, NoneType]

Return the initializer associated with the given variable name.

The name of the current variable scope is prepended to the variable name.

This should only be called during model building.

`neuralmonkey.tf_utils.`**`get_shape_list`**(*x: tensorflow.python.framework.ops.Tensor*) → List[Union[int, tensorflow.python.framework.ops.Tensor]]

Return list of dims, statically where possible.

Compute the static shape of a tensor. Where the dimension is not static (e.g. batch or time dimension), symbolic Tensor is returned.

Based on tensor2tensor.

Parameters **x** – The `Tensor` to process.

Returns A list of integers and Tensors.

`neuralmonkey.tf_utils.`**`get_state_shape_invariants`**(*state: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.tensor_shape.TensorShape

Return the shape invariant of a tensor.

This function computes the loosened shape invariant of a state tensor. Only invariant dimension is the state size dimension, which is the last.

Based on tensor2tensor.

Parameters **state** – The state tensor.

Returns A `TensorShape` object with all but the last dimensions set to `None`.

`neuralmonkey.tf_utils.`**`get_variable`**(*name: str*, *shape: List[int] = None*, *dtype: tensorflow.python.framework.dtypes.DType = None*, *initializer: Callable = None*, ***kwargs*) → tensorflow.python.ops.variables.Variable

Get an existing variable with these parameters or create a new one.

This is a wrapper around *tf.get_variable*. The *initializer* parameter is treated as a default which can be overriden by a call to *update_initializers*.

This should only be called during model building.

`neuralmonkey.tf_utils.`**`layer_norm`**(*x: tensorflow.python.framework.ops.Tensor*, *epsilon: float = 1e-06*) → tensorflow.python.framework.ops.Tensor

Layer normalize the tensor x, averaging over the last dimension.

Implementation based on tensor2tensor.

Parameters

- **`x`** – The `Tensor` to normalize.
- **`epsilon`** – The smoothing parameter of the normalization.

Returns The normalized tensor.

`neuralmonkey.tf_utils.`**`partial_transpose`**(*x: tensorflow.python.framework.ops.Tensor*, *indices: List[int]*) → tensorflow.python.framework.ops.Tensor

Do a transpose on a subset of tensor dimensions.

Compute a permutation of first k dimensions of a tensor.

Parameters

- **`x`** – The `Tensor` to transpose.
- **`indices`** – The permutation of the first k dimensions of `x`.

Returns The transposed tensor.

`neuralmonkey.tf_utils.`**`tf_print`**(*tensor: tensorflow.python.framework.ops.Tensor*, *message: str = None*, *debug_label: str = None*) → tensorflow.python.framework.ops.Tensor

Print the value of a tensor to the debug log.

Better than tf.Print, logs to console only when the "tensorval" debug subject is turned on.

Idea found at: https://stackoverflow.com/a/39649614

Parameters **`tensor`** – The tensor whose value to print

Returns As tf.Print, this function returns a tensor identical to the input tensor, with the printing side-effect added.

`neuralmonkey.tf_utils.`**`update_initializers`**(*initializers: Iterable[Tuple[str, Callable]]*) → None

neuralmonkey.train module

Training script for sequence to sequence learning.

`neuralmonkey.train.`**`main`**() → None

neuralmonkey.vocabulary module

Vocabulary class module.

This module implements the Vocabulary class and the helper functions that can be used to obtain a Vocabulary instance.

class neuralmonkey.vocabulary.**Vocabulary**(*tokenized_text: List[str] = None*, *unk_sample_prob: float = 0.0*) → None

Bases: `collections.abc.Sized`

`__init__`(*tokenized_text: List[str] = None*, *unk_sample_prob: float = 0.0*) → None
Create a new instance of a vocabulary.

Parameters **`tokenized_text`** – The initial list of words to add.

`add_characters`(*word: str*) → None

`add_tokenized_text`(*tokenized_text: List[str]*) → None
Add words from a list to the vocabulary.

Parameters **`tokenized_text`** – The list of words to add.

`add_word`(*word: str*, *occurences: int = 1*) → None
Add a word to the vocablulary.

Parameters

- **`word`** – The word to add. If it's already there, increment the count.
- **`occurences`** – increment the count of word by the number of occurences

`get_unk_sampled_word_index`(*word*)
Return index of the specified word with sampling of unknown words.

This method returns the index of the specified word in the vocabulary. If the frequency of the word in the vocabulary is 1 (the word was only seen once in the whole training dataset), with probability of self.unk_sample_prob, generate the index of the unknown token instead.

Parameters **`word`** – The word to look up.

Returns Index of the word, index of the unknown token if sampled, or index of the unknown token if the word is not present in the vocabulary.

`get_word_index`(*word: str*) → int
Return index of the specified word.

Parameters **`word`** – The word to look up.

Returns Index of the word or index of the unknown token if the word is not present in the vocabulary.

`log_sample`(*size: int = 5*) → None
Log a sample of the vocabulary.

Parameters **`size`** – How many sample words to log.

`save_wordlist`(*path: str*, *overwrite: bool = False*, *save_frequencies: bool = False*, *encoding: str = 'utf-8'*) → None
Save the vocabulary as a wordlist.

The file is ordered by the ids of words. This function is used mainly for embedding visualization.

Parameters

- **`path`** – The path to save the file to.

- **overwrite** – Flag whether to overwrite existing file. Defaults to False.
- **save_frequencies** – flag if frequencies should be stored. This parameter adds header into the output file.

Raises

- FileExistsError if the file exists and overwrite flag is
- disabled.

sentences_to_tensor(*sentences: List[List[str]], max_len: int = None, pad_to_max_len: bool = True, train_mode: bool = False, add_start_symbol: bool = False, add_end_symbol: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray]

Generate the tensor representation for the provided sentences.

Parameters

- **sentences** – List of sentences as lists of tokens.
- **max_len** – If specified, all sentences will be truncated to this length.
- **pad_to_max_len** – If True, the tensor will be padded to *max_len*, even if all of the sentences are shorter. If False, the shape of the tensor will be determined by the maximum length of the sentences in the batch.
- **train_mode** – Flag whether we are training or not (enables/disables unk sampling).
- **add_start_symbol** – If True, the <*s*> token will be added to the beginning of each sentence vector. Enabling this option extends the maximum length by one.
- **add_end_symbol** – If True, the </*s*> token will be added to the end of each sentence vector, provided that the sentence is shorter than *max_len*. If not, the end token is not added. Unlike *add_start_symbol*, enabling this option **does not alter** the maximum length.

Returns

A tuple of a sentence tensor and a padding weight vector.

The shape of the tensor representing the sentences is either *(batch_max_len, batch_size)* or *(batch_max_len+1, batch_size)*, depending on the value of the *add_start_symbol* argument. *batch_max_len* is the length of the longest sentence in the batch (including the optional </*s*> token), limited by *max_len* (if specified).

The shape of the padding vector is the same as of the sentence vector.

truncate(*size: int*) → None

Truncate the vocabulary to the requested size.

The infrequent tokens are discarded.

Parameters **size** – The final size of the vocabulary

truncate_by_min_freq(*min_freq: int*) → None

Truncate the vocabulary only keeping words with a minimum frequency.

Parameters **min_freq** – The minimum frequency of included words.

vectors_to_sentences(*vectors: Union[List[numpy.ndarray], numpy.ndarray]*) → List[List[str]]

Convert vectors of indexes of vocabulary items to lists of words.

Parameters **vectors** – List of vectors of vocabulary indices.

Returns List of lists of words.

neuralmonkey.vocabulary.**from_dataset**(*datasets: List[neuralmonkey.dataset.Dataset], series_ids: List[str], max_size: int, save_file: str = None, overwrite: bool = False, min_freq: Union[int, NoneType] = None, unk_sample_prob: float = 0.5*) → neuralmonkey.vocabulary.Vocabulary

Load a vocabulary from a dataset with an option to save it.

Parameters

- **datasets** – A list of datasets from which to create the vocabulary
- **series_ids** – A list of ids of series of the datasets that should be used producing the vocabulary
- **max_size** – The maximum size of the vocabulary
- **save_file** – A file to save the vocabulary to. If None (default), the vocabulary will not be saved.
- **overwrite** – Overwrite existing file.
- **min_freq** – Do not include words with frequency smaller than this.
- **unk_sample_prob** – The probability with which to sample unks out of words with frequency 1. Defaults to 0.5.

Returns The new Vocabulary instance.

neuralmonkey.vocabulary.**from_file**(**args, **kwargs*) → neuralmonkey.vocabulary.Vocabulary

neuralmonkey.vocabulary.**from_nematus_json**(*path: str, max_size: int = None, pad_to_max_size: bool = False*) → neuralmonkey.vocabulary.Vocabulary

Load vocabulary from Nematus JSON format.

The JSON format is a flat dictionary that maps words to their index in the vocabulary.

Parameters

- **path** – Path to the file.
- **max_size** – Maximum vocabulary size including 'unk' and 'eos' symbols, but not including <pad> and <s> symbol.
- **pad_to_max_size** – If specified, the vocabulary is padded with dummy symbols up to the specified maximum size.

neuralmonkey.vocabulary.**from_t2t_vocabulary**(*path: str, encoding: str = 'utf-8'*) → neuralmonkey.vocabulary.Vocabulary

Load a vocabulary generated during tensor2tensor training.

Parameters

- **path** – The path to the vocabulary file.
- **encoding** – The encoding of the vocabulary file (defaults to UTF-8).

Returns The new Vocabulary instantce.

neuralmonkey.vocabulary.**from_wordlist**(*path: str, encoding: str = 'utf-8', contains_header: bool = True, contains_frequencies: bool = True*) → neuralmonkey.vocabulary.Vocabulary

Load a vocabulary from a wordlist.

The file can contain either list of words with no header. Or it can contain words and their counts separated by tab and a header on the first line.

Parameters

- **path** – The path to the wordlist file
- **encoding** – The encoding of the wordlist file (defaults to UTF-8)
- **contains_header** – if the file have a header on first line
- **contains_frequencies** – if the file contains frequencies in second column

Returns The new Vocabulary instance.

neuralmonkey.vocabulary.**initialize_vocabulary**(*directory: str*, *name: str*, *datasets: List[neuralmonkey.dataset.Dataset] = None*, *series_ids: List[str] = None*, *max_size: int = None*) → neuralmonkey.vocabulary.Vocabulary

Initialize a vocabulary.

This function is supposed to initialize vocabulary when called from the configuration file. It first checks whether the vocabulary is already loaded on the provided path and if not, it tries to generate it from the provided dataset.

Parameters

- **directory** – Directory where the vocabulary should be stored.
- **name** – Name of the vocabulary which is also the name of the file it is stored it.
- **datasets** – A a list of datasets from which the vocabulary can be created.
- **series_ids** – A list of ids of series of the datasets that should be used for producing the vocabulary.
- **max_size** – The maximum size of the vocabulary

Returns The new vocabulary

neuralmonkey.vocabulary.**is_special_token**(*word: str*) → bool

Check whether word is a special token (such as <pad> or <s>).

Parameters **word** – The word to check

Returns True if the word is special, False otherwise.

Module contents

The neuralmonkey package is the root package of this project.

1.7 Visualization

1.7.1 LogBook

Neural Monkey LogBook is a simple web application for preview the outputs of the experiments in the browser.

The experiment data are stored in a directory structure, where each experiment directory contains the experiment configuration, state of the git repository, the experiment was executed with, detailed log of the computation and other files necessary to execute the model that has been trained.

LogBook is meant as a complement to using *TensorBoard*, whose summaries are stored in the same directory structure.

How to run it

You can run the server using the following command:

```
bin/neuralmonkey-logbook --logdir=<experiments> --port=<port> --host=<host>
```

where *<experiments>* is the directory where the experiments are listed and *<port>* is the number of the port the server will run on, and *<host>* is the IP address of the host (defaults to 127.0.0.1, if you want the logbook to be visible to other computers in the network, set the host to 0.0.0.0)

Then you can navigate in your browser to *http://localhost:<port>* to view the experiment logs.

1.7.2 TensorBoard

You can use *TensorBoard <https://www.tensorflow.org/versions/r0.9/how_tos/summaries_and_tensorboard/index.html>* to visualize your TensorFlow graph, see summaries of quantitative metrics about the execution of your graph, and show additional data like images that pass through it.

You can start it by following command:

```
tensorboard --logdir=<experiments>
```

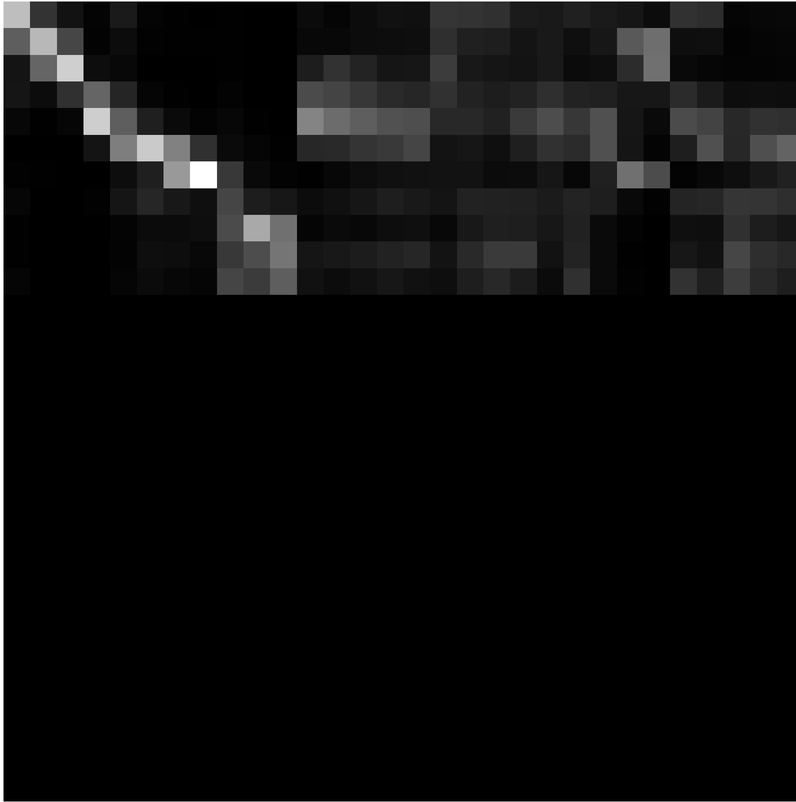
And then you can navigate in your browser to *http://localhost:6006/* (or if the TensorBoard assigns different port) and view all the summaries about your experiment.

How to read TensorBoard

The *step* in the TensorBoard is describing how many inputs (not batches) was processed.

1.7.3 Attention Visualization

If you are using an attention decoder, visualization of the soft alignment of each sentence in the first validation batch will appear in the *Images* tab in *TensorBoard*. The images might look like this:



Here, the source sentence is on the vertical axis and the target sentence on the horizontal axis. The size of each image is `max_output_len * max_input_len` so most of the time, there will be some blank rows at the bottom and some trailing columns with "phantom" attention (corresponding to positions after the end of the output sentence).

You can use the `tf_save_images.py` script to save the whole history of images as a sequence of PNG files:

```
# For the first sentence in the batch
scripts/tf_save_images.py events.out attention_0/image/0 --prefix images/attention_0_
```

Use `feh` to view the images as a time-lapse:

```
feh -g 300x300 -Z --force-aliasing --slideshow-delay 0.2 images/attention_0_*.png
```

Or enlarge them and turn them into an animated GIF using:

```
convert images/attention_0_*.png -scale 300x300 images/attention_0.gif
```

1.8 Advanced Features

1.8.1 Byte Pair Encoding

This is explained in *the machine translation tutorial*.

1.8.2 Dropout

Neural networks with a large number of parameters have a serious problem with an overfitting. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural

network during training. This prevents units from co-adapting too much. But during the test time, the dropout is turned off. More information in https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

If you want to enable dropout on an encoder or on the decoder, you can simply add dropout_keep_prob to the particular section:

```
[encoder]
class=encoders.recurrent.SentenceEncoder
dropout_keep_prob=0.8
...
```

or:

```
[decoder]
class=decoders.decoder.Decoder
dropout_keep_prob=0.8
...
```

1.8.3 Pervasive Dropout

Detailed information in https://arxiv.org/abs/1512.05287

If you want allow dropout on the recurrent layer of your encoder, you can add use_pervasive_dropout parameter into it and then the dropout probability will be used:

```
[encoder]
class=encoders.recurrent.SentenceEncoder
dropout_keep_prob=0.8
use_pervasive_dropout=True
...
```

1.8.4 Attention Seeded by GIZA++ Word Alignments

todo: OC to reference the paper and describe how to use this in NM

1.9 Use SGE cluster array job for inference

To speed up the inference, the `neuralmonkey-run` binary provides the `--grid` option, which can be used when running the program as a SGE array job.

The `run` script make use of the `SGE_TASK_ID` and `SGE_TASK_STEPSIZE` environment variables that are set in each computing node of the array job. If the `--grid` option is supplied and these variables are present, it runs the inference only on a subset of the dataset, specified by the variables.

Consider this example `test_data.ini`:

```
[main]
test_datasets=[<dataset>]
variables=["path/to/variables.data"]

[dataset]
class=dataset.load_dataset_from_files
s_source="data/source.en"
s_target_out="out/target.de"
```

If we want to run a model configured in `model.ini` on this dataset, we can do:

```
neuralmonkey-run model.ini test_data.ini
```

And the program executes the model on the dataset loaded from `data/source.en` and stores the results in `out/target.de`.

If the source file is large or if you use a slow inference method (such as beam search), you may want to split the source file into smaller parts and execute the model on all of them in parallel. If you have access to a SGE cluster, you don't have to do it manually - just create an array job and supply the `--grid` option to the program. Now, suppose that the source file contains 100,000 sentences and you want to split it to 100 parts and run it on cluster. To accomplish this, just run:

```
qsub <qsub_options> -t 1-100000:1000 -b y \
"neuralmonkey-run --grid model.ini test_data.ini"
```

This will submit 100 jobs to your cluster. Each job will use its `SGE_TASK_ID` and `SGE_TASK_STEPSIZE` parameters to determine its part of the data to process. It then runs the inference only on the subset of the dataset and stores the result in a suffixed file.

For example, if the `SGE_TASK_ID` is 3, the `SGE_TASK_STEPSIZE` is 100, and the `--grid` option is specified, the inference will be run on lines 201 to 300 of the file `data/source.en` and the output will be written to `out/target.de.0000000200`.

After all the jobs are finished, you just need to manually run:

```
cat out/target.de.* > out/target.de
```

and delete the intermediate files. (Careful when your file has more than 10^10 lines - you need to concatenate the intermediate files in the right order!)

1.10 GPU Benchmarks

We have done some benchmarks on our department to find out differences between GPUs and we have decided to shared them here. Therefore they do not test speed of Neural Monkey, but they test different GPU cards with the same setup in Neural Monkey.

The benchmark test consisted of one epoch of Machine Translation training in Neural Monkey on a set of fixed data. The size of the model nicely fit into the 2GB memory, therefore GPUs with more memory could have better results with bigger models in comparison to CPUs. All GPUs have CUDA8.0

Setup (cc=cuda capability)	Running time
GeForce GTX 1080; cc6.1	9:55:58
GeForce GTX 1080; cc6.1	10:19:40
GeForce GTX 1080; cc6.1	12:34:34
GeForce GTX 1080; cc6.1	13:01:05
GeForce GTX Titan Z; cc3.5	16:05:24
Tesla K40c; cc3.5	22:41:01
Tesla K40c; cc3.5	22:43:10
Tesla K40c; cc3.5	24:19:45
16 cores Intel Xeon Sandy Bridge 2012 CPU	46:33:14
16 cores Intel Xeon Sandy Bridge 2012 CPU	52:36:56
Quadro K2000; cc3.0	59:47:58
8 cores Intel Xeon Sandy Bridge 2012 CPU	60:39:17
GeForce GT 630; cc3.0	103:42:30
8 cores Intel Xeon Westmere 2010 CPU	134:41:22

1.11 Development Guidelines

1.11.1 Github Workflow

This is a brief document about the Neural Monkey development workflow. Its primary aim is to describe the environment around the Github repository (e.g. continuous integration tests, documentation), pull requests, code-review, etc.

This document is written chronologically, from the point of view of a contributor.

Creating an issue

Everytime there is a need to change the codebase, the contributor should create a corresponing issue in the Github repository.

The name of the issue should be comprehensive, and should summarize the issue in less than 10 words. In the issue description, all the relevant information should be mentioned, and, if applicable, a sketch of the solution should be given so the fashion and method of the solution can be subject to further discussion.

Labels

There is a number of label tags to use to provide an easier way to orient among the issues. Here is an explanation of some of them, so they are not used incorrectly (notably, there is a slight difference between "enhancement" and "feature").

- bug: Use when there is something wrong in the current codebase that needs to be fixed. For example, "Random seeds are not working"
- documentation: Use when the main topic of the issue or pull request is to contribute to the documentation (be it a rst document or a request for more docstrings)
- tests: Similarly to documentation, use if the main topic of the issue is to write a test or to do changes to the testing process itself.
- feature: A request for implementing a feature regarding the training of the models or the models themselves, e.g. "Minimum risk training" or "Implementation of conditional GRU".

- enhancement: A request for implementing a feature to Neural Monkey aimed at improving the user experience with the package, e.g. "GPU profiling" or "Logging of config building".
- help wanted: Used as an additional label, which specify that solving the issue is suitable either for new contributors or for researchers who want to try out a feature, which would be otherwise implemented after a longer time.
- refactor: Refactor issues are requests for cleaning the codebase, using better ways to achieve the same results, conforming to a future API, etc. For example, "Rewrite decoder using decorators"

Todo: Replace text with label pictures from Github

Selecting an issue to work on and assigning people

Note: If you want to start working on something and don't have a preference, check out the issues labeled "Help wanted"

When you decide to work on an issue, assign yourself to it and describe your plans on how you will proceed (in case there is no solution sketch provided in the issue description). This way, others may comment on your plans prior to the work, which can save a lot of time.

Please make sure that you put all additional information as a comment to the issue in case the issue has been discussed elsewhere.

Creating a branch

Prior to writing code (or at least before the first commit), you should create a branch for solution of the issue. This command creates a new branch called `your_branch_name` and switches your working copy to that branch:

```
$ git checkout -b your_branch_name
```

Writing code

On the new branch, you can make changes and commit, until your solution is done.

It is worth noting that we are trying to keep our code clean by enforcing some code writing rules and guidelines. These are automatically check by Travis CI on each push to the Github repository. Here is a list of tools used to check the quality of the code:

- pylint
- pycodestyle
- mypy
- markdownlint

Todo: provide short description to the tools, check that markdownlint has correct URL

You can run the tests on your local machine by using scripts (and requirements) from the `tests/` directory of this package,

This is a usual mantra that you can use for committing and pushing to the remote branch in the repository:

```
$ git add .
$ git commit -m 'your commit message'
$ git push origin your_branch_name
```

Note: If you are working on a branch with someone else, it is always a good idea to do a `git pull --rebase` before pushing. This command updates your branch with remote changes and apply your new commits on top of them.

Warning: If your commit message contains the string `[ci skip]` the continuous integration tests are not run. However, try not to use this feature unless you know what you're doing.

Creating a pull request

Whenever you want to add a feature or push a bugfix, you should make a new pull request, which can be reviewed and merged by someone else. The typical workflow should be as follows:

1. Create a new branch, make your changes and push them to the repository.
2. You should now see the new branch on the Github project page. When you open the branch page, click on "Create Pull request" button.
3. When the pull request is created, the continuous integration tests are run on Travis. You can see the status of the test run on the pull request page. There is also a link to Travis so you can inspect the results of the test run, and make additional changes in order to make the tests successful, if needed. Additionally to the code quality checking tools, unit and regression tests are run as well.

When you create a pull request, assign one or two people to do the review.

Code review and merging

Your pull requests should always be subject to code review. After you create the pull request, select one or two contributors and assign them to make a review.

This phase consists of discussion about the introduced changes, suggestions, and another requirements made by the reviewers. Anyone who wants to do a review can contribute, the reviewer roles are not considered exclusive.

After all of the reviewers' comments have been addressed and the reviewers approved the pull request, the pull request can be merged. It is usually a good idea to rebase the code to the recent version of master. Assuming your working copy is switched to the **master** branch, do:

```
$ git pull --rebase
$ git checkout your_branch_name
$ git rebase master
```

These commands first update your local copy of master from the remote repository, then switch your working copy to the `your_branch_name` branch, and then rebases the branch on the updated master.

Rebasing is a process in which commits from a branch (`your_branch_name`) are applied on a second branch (master), and the new HEAD is marked as the first branch.

Warning: Rebasing is a process which overwrites history. Therefore be absolutely sure that you know what are you doing. Usually if you work on a branch alone, rebasing is a safe procedure.

When the branch is rebased, you have to force-push it to the repository:

```
$ git push -f origin your_branch_name
```

This command overwrites the your branch in the remote repository with your local branch (which is now rebased on master, and therefore, up-to-date)

Note: You can use rebasing also for updating your branch to work with newer versions of master instead of merging the master in the branch. Bear in mind though, that you should force-push these updates, so no-one works on the outdated version of the branch.

Finally, one more round of tests is run and if everything is OK, you can click the "Merge pull request" button, which executes the merge. You can also click another button to delete the `your_branch_name` branch from the repository after the merge.

Documentation

Documentation related to GitHub is written in Markdown files, Python documentation using reStructuredText. This concerns both the standalone documents (in `/docs/`) and the docstrings in source code.

Style of the Markdown files is automatically checked using Markdownlint.

1.11.2 Running tests

Every time a commit is pushed to the Github repository, the tests are run on Travis CI.

If you want to run the tests locally, install the required tools:

```
(nm)$ pip install --upgrade -r <(cat tests/*_requirements.txt)
```

Test scripts

Test scripts located in the *tests* directory:

- *tests_run.sh* runs training with small dataset and *small.ini* configuration
- *unit-tests_run.sh* runs unit tests
- *lint_run.sh* runs pylint
- *mypy_run.sh* runs mypy

All the scripts should be run from the main directory of the repository. There is also *run_tests.sh* in the main directory, that runs all the tests above.

Python Module Index

n

neuralmonkey, 134
neuralmonkey.attention, 41
neuralmonkey.attention.base_attention, 21
neuralmonkey.attention.combination, 24
neuralmonkey.attention.coverage, 27
neuralmonkey.attention.feed_forward, 28
neuralmonkey.attention.namedtuples, 29
neuralmonkey.attention.scaled_dot_product, 30
neuralmonkey.attention.stateful_context, 36
neuralmonkey.attention.transformer_cross_layer, 37
neuralmonkey.checking, 118
neuralmonkey.checkpython, 118
neuralmonkey.dataset, 118
neuralmonkey.decoders, 66
neuralmonkey.decoders.autoregressive, 41
neuralmonkey.decoders.beam_search_decoder, 45
neuralmonkey.decoders.classifier, 49
neuralmonkey.decoders.ctc_decoder, 50
neuralmonkey.decoders.decoder, 52
neuralmonkey.decoders.encoder_projection, 55
neuralmonkey.decoders.output_projection, 56
neuralmonkey.decoders.sequence_labeler, 58
neuralmonkey.decoders.sequence_regressor, 60
neuralmonkey.decoders.transformer, 61
neuralmonkey.decoders.word_alignment_decoder, 65
neuralmonkey.decorators, 121
neuralmonkey.encoders, 90
neuralmonkey.encoders.attentive, 66
neuralmonkey.encoders.cnn_encoder, 67
neuralmonkey.encoders.facebook_conv, 70
neuralmonkey.encoders.imagenet_encoder, 72
neuralmonkey.encoders.numpy_stateful_filler, 74
neuralmonkey.encoders.pooling, 76
neuralmonkey.encoders.raw_rnn_encoder, 77
neuralmonkey.encoders.recurrent, 79
neuralmonkey.encoders.sentence_cnn_encoder, 83
neuralmonkey.encoders.sequence_cnn_encoder, 85
neuralmonkey.encoders.transformer, 87
neuralmonkey.evaluators.accuracy, 90
neuralmonkey.evaluators.average, 91
neuralmonkey.evaluators.beer, 91
neuralmonkey.evaluators.bleu, 92
neuralmonkey.evaluators.chrf, 93
neuralmonkey.evaluators.edit_distance, 94
neuralmonkey.evaluators.evaluator, 95
neuralmonkey.evaluators.f1_bio, 96
neuralmonkey.evaluators.gleu, 96
neuralmonkey.evaluators.mse, 97
neuralmonkey.evaluators.multeval, 98
neuralmonkey.evaluators.sacrebleu, 99
neuralmonkey.evaluators.ter, 100
neuralmonkey.evaluators.wer, 100
neuralmonkey.experiment, 121
neuralmonkey.functions, 123
neuralmonkey.learning_utils, 124
neuralmonkey.logging, 126
neuralmonkey.run, 127
neuralmonkey.runners, 111
neuralmonkey.runners.base_runner, 101
neuralmonkey.runners.beamsearch_runner, 103

neuralmonkey.runners.ctc_debug_runner, 104
neuralmonkey.runners.label_runner, 105
neuralmonkey.runners.logits_runner, 105
neuralmonkey.runners.perplexity_runner, 106
neuralmonkey.runners.plain_runner, 106
neuralmonkey.runners.regression_runner, 107
neuralmonkey.runners.runner, 107
neuralmonkey.runners.tensor_runner, 108
neuralmonkey.runners.word_alignment_runner, 109
neuralmonkey.tf_manager, 127
neuralmonkey.tf_utils, 128
neuralmonkey.train, 130
neuralmonkey.trainers, 118
neuralmonkey.trainers.cross_entropy_trainer, 111
neuralmonkey.trainers.delayed_update_trainer, 112
neuralmonkey.trainers.generic_trainer, 113
neuralmonkey.trainers.multitask_trainer, 114
neuralmonkey.trainers.rl_trainer, 115
neuralmonkey.trainers.self_critical_objective, 116
neuralmonkey.trainers.test_multitask_trainer, 117
neuralmonkey.vocabulary, 131

Index

Symbols

__init__() (neuralmonkey.attention.base_attention.BaseAttention method), 22
__init__() (neuralmonkey.attention.combination.FlatMultiAttention method), 24
__init__() (neuralmonkey.attention.combination.HierarchicalMultiAttention method), 26
__init__() (neuralmonkey.attention.combination.MultiAttention method), 27
__init__() (neuralmonkey.attention.coverage.CoverageAttention method), 28
__init__() (neuralmonkey.attention.feed_forward.Attention method), 28
__init__() (neuralmonkey.attention.scaled_dot_product.MultiHeadAttention method), 31
__init__() (neuralmonkey.attention.scaled_dot_product.ScaledDotProdAttention method), 33
__init__() (neuralmonkey.attention.stateful_context.StatefulContext method), 36
__init__() (neuralmonkey.dataset.BatchingScheme method), 118
__init__() (neuralmonkey.dataset.Dataset method), 119
__init__() (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder method), 42
__init__() (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder method), 46
__init__() (neuralmonkey.decoders.classifier.Classifier method), 49
__init__() (neuralmonkey.decoders.ctc_decoder.CTCDecoder method), 50
__init__() (neuralmonkey.decoders.decoder.Decoder method), 52
__init__() (neuralmonkey.decoders.sequence_labeler.SequenceLabeler method), 58
__init__() (neuralmonkey.decoders.sequence_regressor.SequenceRegressor method), 60
__init__() (neuralmonkey.decoders.transformer.TransformerDecoder method), 62
__init__() (neuralmonkey.decoders.word_alignment_decoder.WordAlignmentDecoder method), 65
__init__() (neuralmonkey.encoders.attentive.AttentiveEncoder method), 66
__init__() (neuralmonkey.encoders.cnn_encoder.CNNEncoder method), 67
__init__() (neuralmonkey.encoders.cnn_encoder.CNNTemporalView method), 69
__init__() (neuralmonkey.encoders.facebook_conv.SentenceEncoder method), 71
__init__() (neuralmonkey.encoders.imagenet_encoder.ImageNet method), 72
__init__() (neuralmonkey.encoders.numpy_stateful_filler.SpatialFiller method), 74
__init__() (neuralmonkey.encoders.numpy_stateful_filler.StatefulFiller method), 75
__init__() (neuralmonkey.encoders.pooling.SequencePooling method), 77
__init__() (neuralmonkey.encoders.raw_rnn_encoder.RawRNNEncoder method), 77
__init__() (neuralmonkey.encoders.recurrent.DeepSentenceEncoder method), 79
__init__() (neuralmonkey.encoders.recurrent.FactoredEncoder method), 80
__init__() (neuralmonkey.encoders.recurrent.RecurrentEncoder method), 81
__init__() (neuralmonkey.encoders.recurrent.SentenceEncoder method), 82
__init__() (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEn method), 84
__init__() (neuralmonkey.encoders.sequence_cnn_encoder.SequenceCNNE method), 86
__init__() (neuralmonkey.encoders.transformer.TransformerEncoder method), 88
__init__() (neuralmonkey.encoders.transformer.TransformerLayer method), 90
__init__() (neuralmonkey.evaluators.beer.BeerWrapper method), 91
__init__() (neuralmonkey.evaluators.bleu.BLEUEvaluator method), 92
__init__() (neuralmonkey.evaluators.chrf.ChrFEvaluator

method), 93
__init__() (neuralmonkey.evaluators.evaluator.Evaluator method), 95
__init__() (neuralmonkey.evaluators.gleu.GLEUEvaluator method), 97
__init__() (neuralmonkey.evaluators.multeval.MultEvalWrapper method), 98
__init__() (neuralmonkey.evaluators.sacrebleu.SacreBLEUEvaluator method), 99
__init__() (neuralmonkey.experiment.Experiment method), 121
__init__() (neuralmonkey.runners.base_runner.BaseRunner method), 101
__init__() (neuralmonkey.runners.base_runner.GraphExecutor method), 103
__init__() (neuralmonkey.runners.base_runner.GraphExecutor.Executable method), 102
__init__() (neuralmonkey.runners.beamsearch_runner.BeamSearchRunner method), 104
__init__() (neuralmonkey.runners.beamsearch_runner.BeamSearchRunner.Executable method), 103
__init__() (neuralmonkey.runners.ctc_debug_runner.CTCDebugRunner method), 105
__init__() (neuralmonkey.runners.label_runner.LabelRunner method), 105
__init__() (neuralmonkey.runners.logits_runner.LogitsRunner method), 105
__init__() (neuralmonkey.runners.perplexity_runner.PerplexityRunner method), 106
__init__() (neuralmonkey.runners.plain_runner.PlainRunner method), 107
__init__() (neuralmonkey.runners.regression_runner.RegressionRunner method), 107
__init__() (neuralmonkey.runners.runner.GreedyRunner method), 108
__init__() (neuralmonkey.runners.tensor_runner.RepresentationRunner method), 108
__init__() (neuralmonkey.runners.tensor_runner.TensorRunner method), 109
__init__() (neuralmonkey.runners.word_alignment_runner.WordAlignmentRunner method), 110
__init__() (neuralmonkey.tf_manager.TensorFlowManager method), 127
__init__() (neuralmonkey.trainers.cross_entropy_trainer.CrossEntropyTrainer method), 111
__init__() (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer method), 113
__init__() (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer.Executable method), 112
__init__() (neuralmonkey.trainers.generic_trainer.GenericTrainer method), 113
__init__() (neuralmonkey.trainers.generic_trainer.GenericTrainer.Executable method), 113
__init__() (neuralmonkey.trainers.multitask_trainer.MultitaskTrainer method), 114
__init__() (neuralmonkey.vocabulary.Vocabulary method), 131

A

accumulate_ops (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer attribute), 113
AccuracyEvaluator (class in neuralmonkey.evaluators.accuracy), 90
AccuracySeqLevelEvaluator (class in neuralmonkey.evaluators.accuracy), 90
add_characters() (neuralmonkey.vocabulary.Vocabulary method), 131
add_tokenized_text() (neuralmonkey.vocabulary.Vocabulary method), 131
add_word() (neuralmonkey.vocabulary.Vocabulary method), 131
alignment_target (neuralmonkey.decoders.word_alignment_decoder.WordAlignmentDecoder attribute), 65
append_tensor() (in module neuralmonkey.tf_utils), 128
apply_net (neuralmonkey.encoders.imagenet_encoder.ImageNetSpec attribute), 73
assert_same_shape() (in module neuralmonkey.checking), 118
assert_shape() (in module neuralmonkey.checking), 118
Attention (class in neuralmonkey.attention.feed_forward), 28
attention() (in module neuralmonkey.attention.scaled_dot_product), 33
attention() (neuralmonkey.attention.base_attention.BaseAttention method), 22
attention() (neuralmonkey.attention.combination.FlatMultiAttention method), 25
attention() (neuralmonkey.attention.combination.HierarchicalMultiAttention method), 26
attention() (neuralmonkey.attention.combination.MultiAttention method), 27
attention() (neuralmonkey.attention.feed_forward.Attention method), 28
attention() (neuralmonkey.attention.scaled_dot_product.MultiHeadAttention method), 31
attention() (neuralmonkey.attention.stateful_context.StatefulContext method), 37
attention_histories (neuralmonkey.decoders.decoder.RNNHistories attribute), 54
attention_loop_states (neuralmonkey.decoders.beam_search_decoder.BeamSearchOutput attribute), 48
attention_mask (neuralmonkey.attention.feed_forward.Attention

attribute), 29
attention_mask (neuralmonkey.attention.stateful_context.StatefulContext attribute), 37
attention_states (neuralmonkey.attention.feed_forward.Attention attribute), 29
attention_states (neuralmonkey.attention.stateful_context.StatefulContext attribute), 37
attention_weights (neuralmonkey.encoders.attentive.AttentiveEncoder attribute), 67
AttentionLoopState (class in neuralmonkey.attention.namedtuples), 29
AttentiveEncoder (class in neuralmonkey.encoders.attentive), 66
attn_size (neuralmonkey.attention.combination.MultiAttention attribute), 27
attn_v (neuralmonkey.attention.combination.MultiAttention attribute), 27
AutoregressiveDecoder (class in neuralmonkey.decoders.autoregressive), 42
AverageEvaluator (class in neuralmonkey.evaluators.average), 91

B

BaseAttention (class in neuralmonkey.attention.base_attention), 22
BaseRunner (class in neuralmonkey.runners.base_runner), 101
BaseRunner.Executable (class in neuralmonkey.runners.base_runner), 101
batch_bucket_span (neuralmonkey.dataset.BatchingScheme attribute), 118
batch_norm_callback() (neuralmonkey.encoders.cnn_encoder.CNNEncoder method), 68
batch_size (neuralmonkey.dataset.BatchingScheme attribute), 118
batches() (neuralmonkey.dataset.Dataset method), 119
BatchingScheme (class in neuralmonkey.dataset), 118
beam_search_runner_range() (in module neuralmonkey.runners.beamsearch_runner), 104
BeamSearchDecoder (class in neuralmonkey.decoders.beam_search_decoder), 46
BeamSearchLoopState (class in neuralmonkey.decoders.beam_search_decoder), 47
BeamSearchOutput (class in neuralmonkey.decoders.beam_search_decoder), 48
BeamSearchRunner (class in neuralmonkey.runners.beamsearch_runner), 103
BeamSearchRunner.Executable (class in neuralmonkey.runners.beamsearch_runner), 103
BeerWrapper (class in neuralmonkey.evaluators.beer), 91
best_vars_file (neuralmonkey.tf_manager.TensorFlowManager attribute), 128
bias_term (neuralmonkey.attention.feed_forward.Attention attribute), 29
bidirectional_rnn (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 85
bleu() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 92
BLEUEvaluator (class in neuralmonkey.evaluators.bleu), 92
bucketing_ignore_series (neuralmonkey.dataset.BatchingScheme attribute), 118
build_model() (neuralmonkey.experiment.Experiment method), 121

C

cell_type (neuralmonkey.encoders.recurrent.RNNSpec attribute), 81
check_dataset_and_coders() (in module neuralmonkey.checking), 118
check_lengths() (in module neuralmonkey.evaluators.evaluator), 96
CheckingException, 118
child_loop_states (neuralmonkey.attention.namedtuples.HierarchicalLoopState attribute), 30
chr_p() (neuralmonkey.evaluators.chrf.ChrFEvaluator method), 93
chr_r() (neuralmonkey.evaluators.chrf.ChrFEvaluator method), 93
ChrFEvaluator (class in neuralmonkey.evaluators.chrf), 93
chunk2set() (neuralmonkey.evaluators.f1_bio.F1Evaluator static method), 96
Classifier (class in neuralmonkey.decoders.classifier), 49
cnn_encoded (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNN attribute), 85
CNNEncoder (class in neuralmonkey.encoders.cnn_encoder), 67
CNNTemporalView (class in neuralmonkey.encoders.cnn_encoder), 69
collect_results() (neuralmonkey.runners.base_runner.GraphExecutor.Executable method), 102
collect_results() (neuralmonkey.runners.beamsearch_runner.BeamSearchRunner.Executal method), 103

collect_results() (neuralmonkey.runners.ctc_debug_runner.CTCDebugRunner.Executable method), 104
collect_results() (neuralmonkey.runners.label_runner.LabelRunner.Executable method), 105
collect_results() (neuralmonkey.runners.logits_runner.LogitsRunner.Executable method), 105
collect_results() (neuralmonkey.runners.perplexity_runner.PerplexityRunner.Executable method), 106
collect_results() (neuralmonkey.runners.plain_runner.PlainRunner.Executable method), 107
collect_results() (neuralmonkey.runners.regression_runner.RegressionRunner.Executable method), 107
collect_results() (neuralmonkey.runners.runner.GreedyRunner.Executable method), 107
collect_results() (neuralmonkey.runners.tensor_runner.TensorRunner.Executable method), 109
collect_results() (neuralmonkey.runners.word_alignment_runner.WordAlignmentRunner.Executable method), 110
collect_results() (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer.Executable method), 112
collect_results() (neuralmonkey.trainers.generic_trainer.GenericTrainer.Executable method), 113
compare_scores() (neuralmonkey.evaluators.edit_distance.EditDistanceEvaluator static method), 94
compare_scores() (neuralmonkey.evaluators.evaluator.Evaluator static method), 95
compare_scores() (neuralmonkey.evaluators.mse.MeanSquaredErrorEvaluator static method), 97
compare_scores() (neuralmonkey.evaluators.mse.PairwiseMeanSquaredErrorEvaluator static method), 98
compare_scores() (neuralmonkey.evaluators.ter.TEREvaluator static method), 100
compare_scores() (neuralmonkey.evaluators.wer.WEREvaluator static method), 100
concat_encoder_projection() (in module neuralmonkey.decoders.encoder_projection), 55
constants (neuralmonkey.decoders.autoregressive.LoopState attribute), 45
context_vector_size (neuralmonkey.attention.base_attention.BaseAttention attribute), 22
context_vector_size (neuralmonkey.attention.combination.FlatMultiAttention attribute), 25
context_vector_size (neuralmonkey.attention.combination.HierarchicalMultiAttention attribute), 26
context_vector_size (neuralmonkey.attention.feed_forward.Attention attribute), 29
context_vector_size (neuralmonkey.attention.scaled_dot_product.MultiHeadAttention attribute), 32
context_vector_size (neuralmonkey.attention.stateful_context.StatefulContext attribute), 37
contexts (neuralmonkey.attention.namedtuples.AttentionLoopState attribute), 29
contexts (neuralmonkey.attention.namedtuples.MultiHeadLoopState attribute), 30
cost (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 43
cost (neuralmonkey.decoders.classifier.Classifier attribute), 49
cost (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 51
cost (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 59
cost (neuralmonkey.decoders.sequence_regressor.SequenceRegressor attribute), 61
cost (neuralmonkey.decoders.word_alignment_decoder.WordAlignmentDecoder attribute), 65
CoverageAttention (class in neuralmonkey.attention.coverage), 27
create_config() (in module neuralmonkey.experiment), 123
cross_attention_sublayer() (neuralmonkey.encoders.transformer.TransformerEncoder method), 89
CrossEntropyTrainer (class in neuralmonkey.trainers.cross_entropy_trainer), 111
CTCDebugRunner (class in neuralmonkey.runners.ctc_debug_runner), 104
CTCDebugRunner.Executable (class in neuralmonkey.runners.ctc_debug_runner), 104
CTCDecoder (class in neuralmonkey.decoders.ctc_decoder), 50
cumulator_counter (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer attribute), 113

D

Dataset (class in neuralmonkey.dataset), 119
debug() (in module neuralmonkey.logging), 127
debug() (neuralmonkey.logging.Logging static method), 126
debug_disabled_for (neuralmonkey.logging.Logging attribute), 126
debug_enabled() (in module neuralmonkey.logging), 127
debug_enabled() (neuralmonkey.logging.Logging static method), 126
debug_enabled_for (neuralmonkey.logging.Logging attribute), 126
decoded (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 43
decoded (neuralmonkey.decoders.classifier.Classifier attribute), 49
decoded (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 51
decoded (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 59
decoded (neuralmonkey.decoders.sequence_regressor.SequenceRegressor attribute), 61
decoded (neuralmonkey.decoders.word_alignment_decoder.WordAlignmentDecoder attribute), 65
decoded_logits (neuralmonkey.decoders.classifier.Classifier attribute), 49
decoded_seq (neuralmonkey.decoders.classifier.Classifier attribute), 49
decoded_symbols (neuralmonkey.decoders.transformer.TransformerHistories attribute), 64
Decoder (class in neuralmonkey.decoders.decoder), 52
decoder (neuralmonkey.trainers.generic_trainer.Objective attribute), 114
decoder_data_id (neuralmonkey.runners.base_runner.BaseRunner attribute), 101
decoder_loop_state (neuralmonkey.decoders.beam_search_decoder.BeamSearchLoopState attribute), 48
decoder_outputs (neuralmonkey.decoders.autoregressive.DecoderHistories attribute), 45
decoder_state (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder attribute), 46
DecoderConstants (class in neuralmonkey.decoders.autoregressive), 44
DecoderFeedables (class in neuralmonkey.decoders.autoregressive), 44
DecoderHistories (class in neuralmonkey.decoders.autoregressive), 45
decoding_b (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 43
decoding_b (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 59
decoding_loop() (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder method), 43
decoding_loop() (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder method), 46
decoding_residual_w (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 59
decoding_w (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 43
decoding_w (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 59
deduplicate_sentences() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 92
DeepSentenceEncoder (class in neuralmonkey.encoders.recurrent), 79
default_optimizer() (neuralmonkey.trainers.generic_trainer.GenericTrainer static method), 114
DelayedUpdateTrainer (class in neuralmonkey.trainers.delayed_update_trainer), 112
DelayedUpdateTrainer.Executable (class in neuralmonkey.trainers.delayed_update_trainer), 112
dependencies (neuralmonkey.encoders.cnn_encoder.CNNTemporalView attribute), 69
dependencies (neuralmonkey.encoders.transformer.TransformerEncoder attribute), 89
dependencies (neuralmonkey.runners.base_runner.GraphExecutor attribute), 103
diff_buffer (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer attribute), 113
differentiable_loss_sum (neuralmonkey.trainers.generic_trainer.GenericTrainer attribute), 114
dimension (neuralmonkey.decoders.transformer.TransformerDecoder attribute), 63
direction (neuralmonkey.encoders.recurrent.RNNSpec attribute), 81

E

EditDistanceEvaluator (class in neuralmonkey.evaluators.edit_distance), 94
effective_reference_length() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 92
embed_input_symbol() (neuralmonkey.decoders.decoder.Decoder method), 53

embed_inputs() (neuralmonkey.decoders.transformer.TransformerDecoder method), 63
embedded_inputs (neuralmonkey.encoders.sequence_cnn_encoder.SequenceCNNEncoder attribute), 87
embedded_train_inputs (neuralmonkey.decoders.transformer.TransformerDecoder attribute), 63
embedding_matrix (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 43
embedding_size (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 43
empty_attention_loop_state() (in module neuralmonkey.attention.base_attention), 23
empty_initial_state() (in module neuralmonkey.decoders.encoder_projection), 55
empty_multi_head_loop_state() (in module neuralmonkey.attention.scaled_dot_product), 34
encoder_attention_sublayer() (neuralmonkey.decoders.transformer.TransformerDecoder method), 64
encoder_inputs (neuralmonkey.encoders.transformer.TransformerEncoder attribute), 89
encoder_projection (neuralmonkey.decoders.decoder.Decoder attribute), 53
evaluate() (neuralmonkey.experiment.Experiment method), 122
evaluation() (in module neuralmonkey.learning_utils), 124
Evaluator (class in neuralmonkey.evaluators.evaluator), 95
execute() (neuralmonkey.tf_manager.TensorFlowManager method), 128
ExecutionResult (class in neuralmonkey.runners.base_runner), 101
executor (neuralmonkey.runners.base_runner.GraphExecutor.Executable attribute), 102
existing_grads_and_vars (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer attribute), 113
expand_to_beam() (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder method), 46
Experiment (class in neuralmonkey.experiment), 121

F

F1Evaluator (class in neuralmonkey.evaluators.f1_bio), 96
FactoredEncoder (class in neuralmonkey.encoders.recurrent), 80
feed_dict() (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder method), 43
feed_dict() (neuralmonkey.decoders.classifier.Classifier method), 49
feed_dict() (neuralmonkey.decoders.ctc_decoder.CTCDecoder method), 51
feed_dict() (neuralmonkey.decoders.sequence_labeler.SequenceLabeler method), 59
feed_dict() (neuralmonkey.decoders.sequence_regressor.SequenceRegressor method), 61
feed_dict() (neuralmonkey.decoders.word_alignment_decoder.WordAlignm method), 66
feed_dict() (neuralmonkey.encoders.cnn_encoder.CNNEncoder method), 68
feed_dict() (neuralmonkey.encoders.imagenet_encoder.ImageNet method), 72
feed_dict() (neuralmonkey.encoders.numpy_stateful_filler.SpatialFiller method), 74
feed_dict() (neuralmonkey.encoders.numpy_stateful_filler.StatefulFiller method), 75
feed_dict() (neuralmonkey.encoders.raw_rnn_encoder.RawRNNEncoder method), 78
feed_dict() (neuralmonkey.encoders.sequence_cnn_encoder.SequenceCNNE method), 87
feedables (neuralmonkey.decoders.autoregressive.LoopState attribute), 45
feedables (neuralmonkey.runners.base_runner.GraphExecutor attribute), 103
feedforward_sublayer() (neuralmonkey.decoders.transformer.TransformerDecoder method), 64
feedforward_sublayer() (neuralmonkey.encoders.transformer.TransformerEncoder method), 89
fetches (neuralmonkey.runners.base_runner.GraphExecutor attribute), 103
fetches (neuralmonkey.runners.beamsearch_runner.BeamSearchRunner attribute), 104
fetches (neuralmonkey.runners.ctc_debug_runner.CTCDebugRunner attribute), 105
fetches (neuralmonkey.runners.label_runner.LabelRunner attribute), 105
fetches (neuralmonkey.runners.logits_runner.LogitsRunner attribute), 106
fetches (neuralmonkey.runners.perplexity_runner.PerplexityRunner attribute), 106
fetches (neuralmonkey.runners.plain_runner.PlainRunner attribute), 107
fetches (neuralmonkey.runners.regression_runner.RegressionRunner attribute), 107
fetches (neuralmonkey.runners.runner.GreedyRunner attribute), 108

fetches (neuralmonkey.runners.tensor_runner.TensorRunner attribute), 109
fetches (neuralmonkey.runners.word_alignment_runner.WordAlignmentRunner attribute), 110
fetches (neuralmonkey.trainers.generic_trainer.GenericTrainer attribute), 114
fetches (neuralmonkey.trainers.multitask_trainer.MultitaskTrainer attribute), 115
finalize_loop() (neuralmonkey.attention.base_attention.BaseAttention method), 22
finalize_loop() (neuralmonkey.attention.combination.FlatMultiAttention method), 25
finalize_loop() (neuralmonkey.attention.combination.HierarchicalMultiAttention method), 26
finalize_loop() (neuralmonkey.attention.feed_forward.Attention method), 29
finalize_loop() (neuralmonkey.attention.scaled_dot_product.MultiHeadAttention method), 32
finalize_loop() (neuralmonkey.attention.stateful_context.StatefulContext method), 37
finalize_loop() (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder method), 43
finalize_loop() (neuralmonkey.decoders.decoder.Decoder method), 53
finished (neuralmonkey.decoders.autoregressive.DecoderFeedables attribute), 44
finished (neuralmonkey.decoders.beam_search_decoder.SearchState attribute), 48
flat() (in module neuralmonkey.attention.transformer_cross_layer), 37
FlatMultiAttention (class in neuralmonkey.attention.combination), 24
from_dataset() (in module neuralmonkey.vocabulary), 132
from_file() (in module neuralmonkey.vocabulary), 133
from_files() (in module neuralmonkey.dataset), 120
from_nematus_json() (in module neuralmonkey.vocabulary), 133
from_t2t_vocabulary() (in module neuralmonkey.vocabulary), 133
from_wordlist() (in module neuralmonkey.vocabulary), 133

G

gather_flat() (in module neuralmonkey.tf_utils), 129
GenericTrainer (class in neuralmonkey.trainers.generic_trainer), 113
GenericTrainExecutable (class in neuralmonkey.trainers.generic_trainer), 113
get_alexnet() (in module neuralmonkey.encoders.imagenet_encoder), 73
get_attention_mask() (in module neuralmonkey.attention.base_attention), 23
get_attention_states() (in module neuralmonkey.attention.base_attention), 23
get_body() (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder method), 44
get_body() (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder method), 47
get_body() (neuralmonkey.decoders.decoder.Decoder method), 54
get_body() (neuralmonkey.decoders.transformer.TransformerDecoder method), 64
get_current() (neuralmonkey.experiment.Experiment class method), 122
get_default_tf_manager() (in module neuralmonkey.tf_manager), 128
get_encoder_projections() (neuralmonkey.attention.combination.FlatMultiAttention method), 25
get_energies() (neuralmonkey.attention.coverage.CoverageAttention method), 28
get_energies() (neuralmonkey.attention.feed_forward.Attention method), 29
get_executable() (neuralmonkey.runners.base_runner.GraphExecutor method), 103
get_executable() (neuralmonkey.trainers.multitask_trainer.MultitaskTrainer method), 115
get_initial_loop_state() (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder method), 44
get_initial_loop_state() (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder method), 47
get_initial_loop_state() (neuralmonkey.decoders.decoder.Decoder method), 54
get_initial_loop_state() (neuralmonkey.decoders.transformer.TransformerDecoder method), 64
get_initializer() (in module neuralmonkey.tf_utils), 129
get_initializer() (neuralmonkey.experiment.Experiment method), 122
get_logits() (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder method), 44
get_path() (neuralmonkey.experiment.Experiment method), 122

get_resnet_by_type() (in module neuralmonkey.encoders.imagenet_encoder), 73
get_series() (neuralmonkey.dataset.Dataset method), 119
get_shape_list() (in module neuralmonkey.tf_utils), 129
get_state_shape_invariants() (in module neuralmonkey.tf_utils), 129
get_unk_sampled_word_index() (neuralmonkey.vocabulary.Vocabulary method), 131
get_variable() (in module neuralmonkey.tf_utils), 129
get_vgg_by_type() (in module neuralmonkey.encoders.imagenet_encoder), 73
get_word_index() (neuralmonkey.vocabulary.Vocabulary method), 131
gleu() (neuralmonkey.evaluators.gleu.GLEUEvaluator static method), 97
GLEUEvaluator (class in neuralmonkey.evaluators.gleu), 96
go_symbols (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
gradient_buffers (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer attribute), 113
gradients (neuralmonkey.trainers.generic_trainer.GenericTrainer attribute), 114
gradients (neuralmonkey.trainers.generic_trainer.Objective attribute), 114
GraphExecutor (class in neuralmonkey.runners.base_runner), 102
GraphExecutor.Executable (class in neuralmonkey.runners.base_runner), 102
GreedyRunner (class in neuralmonkey.runners.runner), 107
GreedyRunner.Executable (class in neuralmonkey.runners.runner), 107

H

has_series() (neuralmonkey.dataset.Dataset method), 119
head_weights (neuralmonkey.attention.namedtuples.MultiHeadLoopState attribute), 30
hidden_features (neuralmonkey.attention.feed_forward.Attention attribute), 29
hierarchical() (in module neuralmonkey.attention.transformer_cross_layer), 38
HierarchicalLoopState (class in neuralmonkey.attention.namedtuples), 29
HierarchicalMultiAttention (class in neuralmonkey.attention.combination), 25
highway_layer (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 85
histogram_summaries (neuralmonkey.runners.base_runner.ExecutionResult attribute), 102
histories (neuralmonkey.attention.base_attention.BaseAttention attribute), 22
histories (neuralmonkey.decoders.autoregressive.LoopState attribute), 45

I

image_input (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 68
image_mask (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 68
image_processing_layers (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 68
image_size (neuralmonkey.encoders.imagenet_encoder.ImageNetSpec attribute), 73
image_summaries (neuralmonkey.runners.base_runner.ExecutionResult attribute), 102
ImageNet (class in neuralmonkey.encoders.imagenet_encoder), 72
ImageNetSpec (class in neuralmonkey.encoders.imagenet_encoder), 73
init_saving() (neuralmonkey.tf_manager.TensorFlowManager method), 128
initial_loop_state() (neuralmonkey.attention.base_attention.BaseAttention method), 23
initial_loop_state() (neuralmonkey.attention.combination.FlatMultiAttention method), 25
initial_loop_state() (neuralmonkey.attention.combination.HierarchicalMultiAttention method), 27
initial_loop_state() (neuralmonkey.attention.feed_forward.Attention method), 29
initial_loop_state() (neuralmonkey.attention.scaled_dot_product.MultiHeadAttention method), 32
initial_loop_state() (neuralmonkey.attention.stateful_context.StatefulContext method), 37
initial_state (neuralmonkey.decoders.decoder.Decoder attribute), 54
initialize_model_parts() (neuralmonkey.tf_manager.TensorFlowManager method), 128
initialize_sessions() (neuralmonkey.tf_manager.TensorFlowManager method), 128

initialize_vocabulary() (in module neuralmonkey.vocabulary), 134
input_image (neuralmonkey.encoders.imagenet_encoder.ImageNet attribute), 72
input_mask (neuralmonkey.decoders.transformer.TransformerHistories attribute), 64
input_plus_attention() (neuralmonkey.decoders.decoder.Decoder method), 54
input_symbol (neuralmonkey.decoders.autoregressive.DecoderFeedables attribute), 45
inverse_sigmoid_decay() (in module neuralmonkey.functions), 123
is_special_token() (in module neuralmonkey.vocabulary), 134

K

key_projection_matrix (neuralmonkey.attention.feed_forward.Attention attribute), 29

L

LabelRunner (class in neuralmonkey.runners.label_runner), 105
LabelRunner.Executable (class in neuralmonkey.runners.label_runner), 105
last_dec_loop_state (neuralmonkey.decoders.beam_search_decoder.BeamSearchOutput attribute), 48
last_search_state (neuralmonkey.decoders.beam_search_decoder.BeamSearchOutput attribute), 48
last_search_step_output (neuralmonkey.decoders.beam_search_decoder.BeamSearchOutput attribute), 48
layer() (neuralmonkey.decoders.transformer.TransformerDecoder method), 64
layer() (neuralmonkey.encoders.transformer.TransformerEncoder method), 89
layer_norm() (in module neuralmonkey.tf_utils), 130
lengths (neuralmonkey.decoders.beam_search_decoder.SearchState attribute), 48
linear_encoder_projection() (in module neuralmonkey.decoders.encoder_projection), 55
load() (in module neuralmonkey.dataset), 120
load_dataset_from_files() (in module neuralmonkey.dataset), 120
load_runtime_config() (in module neuralmonkey.run), 127
load_variables() (neuralmonkey.experiment.Experiment method), 122
log() (in module neuralmonkey.logging), 127
log() (neuralmonkey.logging.Logging static method), 126
log_file (neuralmonkey.logging.Logging attribute), 126
log_print() (in module neuralmonkey.logging), 127
log_print() (neuralmonkey.logging.Logging static method), 126
log_sample() (neuralmonkey.vocabulary.Vocabulary method), 131
Logging (class in neuralmonkey.logging), 126
logits (neuralmonkey.decoders.autoregressive.DecoderHistories attribute), 45
logits (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 51
logits (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 59
LogitsRunner (class in neuralmonkey.runners.logits_runner), 105
LogitsRunner.Executable (class in neuralmonkey.runners.logits_runner), 105
logprob_sum (neuralmonkey.decoders.beam_search_decoder.SearchState attribute), 48
logprobs (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 59
loop_continue_criterion() (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder method), 44
loop_continue_criterion() (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder method), 47
loop_state (neuralmonkey.attention.namedtuples.HierarchicalLoopState attribute), 30
LoopState (class in neuralmonkey.decoders.autoregressive), 45
loss (neuralmonkey.trainers.generic_trainer.Objective attribute), 114
loss (neuralmonkey.trainers.test_multitask_trainer.TestMP attribute), 117
loss_names (neuralmonkey.runners.base_runner.BaseRunner attribute), 101
loss_names (neuralmonkey.runners.beamsearch_runner.BeamSearchRunner attribute), 104
loss_names (neuralmonkey.runners.ctc_debug_runner.CTCDebugRunner attribute), 105
loss_names (neuralmonkey.runners.label_runner.LabelRunner attribute), 105
loss_names (neuralmonkey.runners.logits_runner.LogitsRunner attribute), 106
loss_names (neuralmonkey.runners.perplexity_runner.PerplexityRunner attribute), 106
loss_names (neuralmonkey.runners.plain_runner.PlainRunner attribute), 107
loss_names (neuralmonkey.runners.regression_runner.RegressionRunner attribute), 107
loss_names (neuralmonkey.runners.runner.GreedyRunner attribute), 108
loss_names (neuralmonkey.runners.tensor_runner.TensorRunner attribute), 109

loss_names (neuralmonkey.runners.word_alignment_runner.WordAlignmentRunner attribute), 110
loss_with_decoded_ins (neuralmonkey.decoders.classifier.Classifier attribute), 50
loss_with_gt_ins (neuralmonkey.decoders.classifier.Classifier attribute), 50
losses (neuralmonkey.runners.base_runner.ExecutionResult attribute), 102

M

main() (in module neuralmonkey.run), 127
main() (in module neuralmonkey.train), 130
mask (neuralmonkey.decoders.autoregressive.DecoderHistories attribute), 45
mask_energies() (in module neuralmonkey.attention.scaled_dot_product), 35
mask_future() (in module neuralmonkey.attention.scaled_dot_product), 35
maxout_output() (in module neuralmonkey.decoders.output_projection), 56
maybe_get_series() (neuralmonkey.dataset.Dataset method), 120
MeanSquaredErrorEvaluator (class in neuralmonkey.evaluators.mse), 97
merge_max_counters() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 92
minimum_reference_length() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 92
mlp_output() (in module neuralmonkey.decoders.output_projection), 56
modality_matrix (neuralmonkey.encoders.transformer.TransformerEncoder attribute), 89
model (neuralmonkey.experiment.Experiment attribute), 122
modified_ngram_precision() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 92
MultEvalWrapper (class in neuralmonkey.evaluators.multeval), 98
MultiAttention (class in neuralmonkey.attention.combination), 27
MultiHeadAttention (class in neuralmonkey.attention.scaled_dot_product), 30
MultiHeadLoopState (class in neuralmonkey.attention.namedtuples), 30
MultitaskTrainer (class in neuralmonkey.trainers.multitask_trainer), 114

N

name (neuralmonkey.evaluators.evaluator.Evaluator attribute), 95
name (neuralmonkey.trainers.generic_trainer.Objective attribute), 114
nematus_output() (in module neuralmonkey.decoders.output_projection), 57
nematus_projection() (in module neuralmonkey.decoders.encoder_projection), 55
neuralmonkey (module), 21, 134
neuralmonkey.attention (module), 41
neuralmonkey.attention.base_attention (module), 21
neuralmonkey.attention.combination (module), 24
neuralmonkey.attention.coverage (module), 27
neuralmonkey.attention.feed_forward (module), 28
neuralmonkey.attention.namedtuples (module), 29
neuralmonkey.attention.scaled_dot_product (module), 30
neuralmonkey.attention.stateful_context (module), 36
neuralmonkey.attention.transformer_cross_layer (module), 37
neuralmonkey.checking (module), 118
neuralmonkey.checkpython (module), 118
neuralmonkey.dataset (module), 118
neuralmonkey.decoders (module), 66
neuralmonkey.decoders.autoregressive (module), 41
neuralmonkey.decoders.beam_search_decoder (module), 45
neuralmonkey.decoders.classifier (module), 49
neuralmonkey.decoders.ctc_decoder (module), 50
neuralmonkey.decoders.decoder (module), 52
neuralmonkey.decoders.encoder_projection (module), 55
neuralmonkey.decoders.output_projection (module), 56
neuralmonkey.decoders.sequence_labeler (module), 58
neuralmonkey.decoders.sequence_regressor (module), 60
neuralmonkey.decoders.transformer (module), 61
neuralmonkey.decoders.word_alignment_decoder (module), 65
neuralmonkey.decorators (module), 121
neuralmonkey.encoders (module), 90
neuralmonkey.encoders.attentive (module), 66
neuralmonkey.encoders.cnn_encoder (module), 67
neuralmonkey.encoders.facebook_conv (module), 70
neuralmonkey.encoders.imagenet_encoder (module), 72
neuralmonkey.encoders.numpy_stateful_filler (module), 74
neuralmonkey.encoders.pooling (module), 76
neuralmonkey.encoders.raw_rnn_encoder (module), 77
neuralmonkey.encoders.recurrent (module), 79
neuralmonkey.encoders.sentence_cnn_encoder (module), 83
neuralmonkey.encoders.sequence_cnn_encoder (module), 85
neuralmonkey.encoders.transformer (module), 87
neuralmonkey.evaluators.accuracy (module), 90

neuralmonkey.evaluators.average (module), 91
neuralmonkey.evaluators.beer (module), 91
neuralmonkey.evaluators.bleu (module), 92
neuralmonkey.evaluators.chrf (module), 93
neuralmonkey.evaluators.edit_distance (module), 94
neuralmonkey.evaluators.evaluator (module), 95
neuralmonkey.evaluators.f1_bio (module), 96
neuralmonkey.evaluators.gleu (module), 96
neuralmonkey.evaluators.mse (module), 97
neuralmonkey.evaluators.multeval (module), 98
neuralmonkey.evaluators.sacrebleu (module), 99
neuralmonkey.evaluators.ter (module), 100
neuralmonkey.evaluators.wer (module), 100
neuralmonkey.experiment (module), 121
neuralmonkey.functions (module), 123
neuralmonkey.learning_utils (module), 124
neuralmonkey.logging (module), 126
neuralmonkey.run (module), 127
neuralmonkey.runners (module), 111
neuralmonkey.runners.base_runner (module), 101
neuralmonkey.runners.beamsearch_runner (module), 103
neuralmonkey.runners.ctc_debug_runner (module), 104
neuralmonkey.runners.label_runner (module), 105
neuralmonkey.runners.logits_runner (module), 105
neuralmonkey.runners.perplexity_runner (module), 106
neuralmonkey.runners.plain_runner (module), 106
neuralmonkey.runners.regression_runner (module), 107
neuralmonkey.runners.runner (module), 107
neuralmonkey.runners.tensor_runner (module), 108
neuralmonkey.runners.word_alignment_runner (module), 109
neuralmonkey.tf_manager (module), 127
neuralmonkey.tf_utils (module), 128
neuralmonkey.train (module), 130
neuralmonkey.trainers (module), 118
neuralmonkey.trainers.cross_entropy_trainer (module), 111
neuralmonkey.trainers.delayed_update_trainer (module), 112
neuralmonkey.trainers.generic_trainer (module), 113
neuralmonkey.trainers.multitask_trainer (module), 114
neuralmonkey.trainers.rl_trainer (module), 115
neuralmonkey.trainers.self_critical_objective (module), 116
neuralmonkey.trainers.test_multitask_trainer (module), 117
neuralmonkey.vocabulary (module), 131
next_to_execute() (neuralmonkey.runners.base_runner.BaseRunner.Executable method), 101
next_to_execute() (neuralmonkey.runners.base_runner.GraphExecutor.Executable method), 102
next_to_execute() (neuralmonkey.runners.beamsearch_runner.BeamSearchRunner.Executal method), 103
next_to_execute() (neuralmonkey.runners.runner.GreedyRunner.Executable method), 107
next_to_execute() (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer.E method), 112
next_to_execute() (neuralmonkey.trainers.generic_trainer.GenericTrainer.Executable method), 113
ngram_counts() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 93
noam_decay() (in module neuralmonkey.functions), 123
nonlinear_output() (in module neuralmonkey.decoders.output_projection), 57
notice() (in module neuralmonkey.logging), 127
notice() (neuralmonkey.logging.Logging static method), 126

O

Objective (class in neuralmonkey.trainers.generic_trainer), 114
objective_buffers (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer attribute), 113
objective_values (neuralmonkey.trainers.generic_trainer.GenericTrainer attribute), 114
order_embeddings (neuralmonkey.encoders.facebook_conv.SentenceEncoder attribute), 71
ordered_embedded_inputs (neuralmonkey.encoders.facebook_conv.SentenceEncoder attribute), 71
output (neuralmonkey.encoders.attentive.AttentiveEncoder attribute), 67
output (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 68
output (neuralmonkey.encoders.cnn_encoder.CNNTemporalView attribute), 69
output (neuralmonkey.encoders.facebook_conv.SentenceEncoder attribute), 71
output (neuralmonkey.encoders.imagenet_encoder.ImageNet attribute), 72
output (neuralmonkey.encoders.numpy_stateful_filler.SpatialFiller attribute), 74
output (neuralmonkey.encoders.numpy_stateful_filler.StatefulFiller attribute), 76
output (neuralmonkey.encoders.pooling.SequenceAveragePooling attribute), 76

output (neuralmonkey.encoders.pooling.SequenceMaxPooling attribute), 76
output (neuralmonkey.encoders.raw_rnn_encoder.RawRNNEncoder attribute), 78
output (neuralmonkey.encoders.recurrent.RecurrentEncoder attribute), 82
output (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 85
output (neuralmonkey.encoders.sequence_cnn_encoder.SequenceCNNEncoder attribute), 87
output (neuralmonkey.encoders.transformer.TransformerEncoder attribute), 89
output_dimension (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
output_dimension (neuralmonkey.decoders.decoder.Decoder attribute), 54
output_dimension (neuralmonkey.decoders.transformer.TransformerDecoder attribute), 64
output_projection (neuralmonkey.decoders.decoder.Decoder attribute), 54
output_projection_spec (neuralmonkey.decoders.decoder.Decoder attribute), 54
outputs (neuralmonkey.decoders.autoregressive.DecoderHistories attribute), 45
outputs (neuralmonkey.runners.base_runner.ExecutionResult attribute), 102

P

PairwiseMeanSquaredErrorEvaluator (class in neuralmonkey.evaluators.mse), 98
parallel() (in module neuralmonkey.attention.transformer_cross_layer), 39
parameterizeds (neuralmonkey.runners.base_runner.GraphExecutor attribute), 103
partial_transpose() (in module neuralmonkey.tf_utils), 130
PerplexityRunner (class in neuralmonkey.runners.perplexity_runner), 106
PerplexityRunner.Executable (class in neuralmonkey.runners.perplexity_runner), 106
piecewise_function() (in module neuralmonkey.functions), 123
plain_convolution() (in module neuralmonkey.encoders.cnn_encoder), 69
PlainRunner (class in neuralmonkey.runners.plain_runner), 106
PlainRunner.Executable (class in neuralmonkey.runners.plain_runner), 107
pooling() (in module neuralmonkey.encoders.cnn_encoder), 70
position_signal() (in module neuralmonkey.encoders.transformer), 90
predictions (neuralmonkey.decoders.sequence_regressor.SequenceRegresso
attribute), 61
prepare_results() (neuralmonkey.runners.beamsearch_runner.BeamSearchRunner.Executal
method), 103
prev_contexts (neuralmonkey.decoders.decoder.RNNFeedables attribute), 54
prev_logits (neuralmonkey.decoders.autoregressive.DecoderFeedables attribute), 45
prev_logprobs (neuralmonkey.decoders.beam_search_decoder.SearchState attribute), 48
prev_rnn_output (neuralmonkey.decoders.decoder.RNNFeedables attribute), 54
prev_rnn_state (neuralmonkey.decoders.decoder.RNNFeedables attribute), 54
print_final_evaluation() (in module neuralmonkey.learning_utils), 124
print_header() (neuralmonkey.logging.Logging static method), 126
projection_bias_vector (neuralmonkey.attention.feed_forward.Attention attribute), 29

Q

query_projection_matrix (neuralmonkey.attention.feed_forward.Attention attribute), 29

R

raw_gradients (neuralmonkey.trainers.delayed_update_trainer.DelayedUpda
attribute), 113
raw_gradients (neuralmonkey.trainers.generic_trainer.GenericTrainer attribute), 114
RawRNNEncoder (class in neuralmonkey.encoders.raw_rnn_encoder), 77
RecurrentEncoder (class in neuralmonkey.encoders.recurrent), 81
reduce_execution_results() (in module neuralmonkey.runners.base_runner), 103
ref_alignment (neuralmonkey.decoders.word_alignment_decoder.WordAlig
attribute), 66
RegressionRunner (class in neuralmonkey.runners.regression_runner), 107
RegressionRunner.Executable (class in neuralmonkey.runners.regression_runner), 107

regularization_losses (neuralmonkey.trainers.generic_trainer.GenericTrainer attribute), 114
reinforce_score() (in module neuralmonkey.trainers.self_critical_objective), 116
RepresentationRunner (class in neuralmonkey.runners.tensor_runner), 108
reset_ops (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer attribute), 113
residual_block() (in module neuralmonkey.encoders.cnn_encoder), 70
restore() (neuralmonkey.tf_manager.TensorFlowManager method), 128
restore_best_vars() (neuralmonkey.tf_manager.TensorFlowManager method), 128
result (neuralmonkey.runners.base_runner.GraphExecutor.Executable attribute), 103
rl_objective() (in module neuralmonkey.trainers.rl_trainer), 115
rnn (neuralmonkey.encoders.recurrent.DeepSentenceEncoder attribute), 80
rnn (neuralmonkey.encoders.recurrent.RecurrentEncoder attribute), 82
rnn_cells() (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder method), 85
rnn_input (neuralmonkey.encoders.recurrent.RecurrentEncoder attribute), 82
rnn_layer() (in module neuralmonkey.encoders.recurrent), 83
rnn_size (neuralmonkey.decoders.decoder.Decoder attribute), 54
RNNFeedables (class in neuralmonkey.decoders.decoder), 54
RNNHistories (class in neuralmonkey.decoders.decoder), 54
RNNSpec (class in neuralmonkey.encoders.recurrent), 81
run_model() (neuralmonkey.experiment.Experiment method), 122
run_on_dataset() (in module neuralmonkey.learning_utils), 124
runtime_logits (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
runtime_logprobs (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
runtime_logprobs (neuralmonkey.decoders.classifier.Classifier attribute), 50
runtime_loop_result (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
runtime_loss (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
runtime_loss (neuralmonkey.decoders.classifier.Classifier attribute), 50
runtime_loss (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 51
runtime_loss (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 59
runtime_loss (neuralmonkey.decoders.sequence_regressor.SequenceRegress attribute), 61
runtime_loss (neuralmonkey.decoders.word_alignment_decoder.WordAlign attribute), 66
runtime_mask (neuralmonkey.decoders.autoregressive.AutoregressiveDecod attribute), 44
runtime_output_states (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
runtime_outputs (neuralmonkey.decoders.word_alignment_decoder.WordAlignmentDeco attribute), 66
runtime_xents (neuralmonkey.decoders.autoregressive.AutoregressiveDecod attribute), 44

S

SacreBLEUEvaluator (class in neuralmonkey.evaluators.sacrebleu), 99
save() (neuralmonkey.tf_manager.TensorFlowManager method), 128
save_git_info() (in module neuralmonkey.experiment), 123
save_wordlist() (neuralmonkey.vocabulary.Vocabulary method), 131
scalar_summaries (neuralmonkey.runners.base_runner.ExecutionResult attribute), 102
ScaledDotProdAttention (class in neuralmonkey.attention.scaled_dot_product), 32
scope (neuralmonkey.encoders.imagenet_encoder.ImageNetSpec attribute), 73
score_batch() (neuralmonkey.evaluators.beer.BeerWrapper method), 91
score_batch() (neuralmonkey.evaluators.bleu.BLEUEvaluator method), 93
score_batch() (neuralmonkey.evaluators.edit_distance.EditDistanceEvaluato method), 94
score_batch() (neuralmonkey.evaluators.evaluator.Evaluator method), 95
score_batch() (neuralmonkey.evaluators.evaluator.SequenceEvaluator method), 95
score_batch() (neuralmonkey.evaluators.gleu.GLEUEvaluator method), 97
score_batch() (neuralmonkey.evaluators.multeval.MultEvalWrapper method), 99
score_batch() (neuralmonkey.evaluators.sacrebleu.SacreBLEUEvaluator method), 99

score_batch() (neuralmonkey.evaluators.wer.WEREvaluator method), 100
score_instance() (neuralmonkey.evaluators.average.AverageEvaluator method), 91
score_instance() (neuralmonkey.evaluators.chrf.ChrFEvaluator method), 93
score_instance() (neuralmonkey.evaluators.edit_distance.EditDistanceEvaluator method), 94
score_instance() (neuralmonkey.evaluators.evaluator.Evaluator method), 95
score_instance() (neuralmonkey.evaluators.f1_bio.F1Evaluator method), 96
score_instance() (neuralmonkey.evaluators.mse.PairwiseMeanSquaredErrorEvaluator method), 98
score_instance() (neuralmonkey.evaluators.ter.TEREvaluator method), 100
score_instance() (neuralmonkey.evaluators.wer.WEREvaluator method), 101
score_token() (neuralmonkey.evaluators.evaluator.SequenceEvaluator method), 96
score_token() (neuralmonkey.evaluators.mse.MeanSquaredErrorEvaluator method), 98
scores (neuralmonkey.decoders.beam_search_decoder.SearchResults attribute), 48
search_results (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder attribute), 47
search_results (neuralmonkey.decoders.beam_search_decoder.BeamSearchLoopState attribute), 48
search_state (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder attribute), 47
search_state (neuralmonkey.decoders.beam_search_decoder.BeamSearchLoopState attribute), 47
SearchResults (class in neuralmonkey.decoders.beam_search_decoder), 48
SearchState (class in neuralmonkey.decoders.beam_search_decoder), 48
self_attention_sublayer() (neuralmonkey.decoders.transformer.TransformerDecoder method), 64
self_attention_sublayer() (neuralmonkey.encoders.transformer.TransformerEncoder method), 89
self_critical_objective() (in module neuralmonkey.trainers.self_critical_objective), 116
sentence_bleu() (in module neuralmonkey.trainers.self_critical_objective), 117
sentence_gleu() (in module neuralmonkey.trainers.self_critical_objective), 117
SentenceCNNEncoder (class in neuralmonkey.encoders.sentence_cnn_encoder), 83
SentenceEncoder (class in neuralmonkey.encoders.facebook_conv), 70
SentenceEncoder (class in neuralmonkey.encoders.recurrent), 82
sentences_to_tensor() (neuralmonkey.vocabulary.Vocabulary method), 132
SequenceAveragePooling (class in neuralmonkey.encoders.pooling), 76
SequenceCNNEncoder (class in neuralmonkey.encoders.sequence_cnn_encoder), 85
SequenceEvaluator (class in neuralmonkey.evaluators.evaluator), 95
SequenceLabeler (class in neuralmonkey.decoders.sequence_labeler), 58
SequenceMaxPooling (class in neuralmonkey.encoders.pooling), 76
SequencePooling (class in neuralmonkey.encoders.pooling), 76
SequenceRegressor (class in neuralmonkey.decoders.sequence_regressor), 60
serial() (in module neuralmonkey.attention.transformer_cross_layer), 39
serialize_to_bytes() (neuralmonkey.evaluators.beer.BeerWrapper method), 91
serialize_to_bytes() (neuralmonkey.evaluators.multeval.MultEvalWrapper method), 99
series (neuralmonkey.dataset.Dataset attribute), 120
sessions (neuralmonkey.tf_manager.TensorFlowManager attribute), 127
set_log_file() (neuralmonkey.logging.Logging static method), 126
set_result() (neuralmonkey.runners.base_runner.GraphExecutor.Executable method), 103
setUp() (neuralmonkey.trainers.test_multitask_trainer.TestMultitaskTrainer method), 117
setUpClass() (neuralmonkey.trainers.test_multitask_trainer.TestMultitaskTr class method), 117
similarity_bias_vector (neuralmonkey.attention.feed_forward.Attention

attribute), 29
single() (in module neuralmonkey.attention.transformer_cross_layer), 40
size (neuralmonkey.encoders.recurrent.RNNSpec attribute), 81
spatial_mask (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 68
spatial_mask (neuralmonkey.encoders.imagenet_encoder.ImageNet attribute), 72
spatial_mask (neuralmonkey.encoders.numpy_stateful_filler.SpatialFiller attribute), 75
spatial_states (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 68
spatial_states (neuralmonkey.encoders.imagenet_encoder.ImageNet attribute), 73
spatial_states (neuralmonkey.encoders.numpy_stateful_filler.SpatialFiller attribute), 75
SpatialFiller (class in neuralmonkey.encoders.numpy_stateful_filler), 74
split_for_heads() (in module neuralmonkey.attention.scaled_dot_product), 36
state_size (neuralmonkey.attention.feed_forward.Attention attribute), 29
state_size (neuralmonkey.attention.stateful_context.StatefulContext attribute), 37
StatefulContext (class in neuralmonkey.attention.stateful_context), 36
StatefulFiller (class in neuralmonkey.encoders.numpy_stateful_filler), 75
step (neuralmonkey.decoders.autoregressive.DecoderFeedables attribute), 44
strict_mode (neuralmonkey.logging.Logging attribute), 126
subset() (neuralmonkey.dataset.Dataset method), 120
summaries (neuralmonkey.trainers.delayed_update_trainer.DelayedUpdateTrainer attribute), 113
summaries (neuralmonkey.trainers.generic_trainer.GenericTrainer attribute), 114

T

target_modality_embedding (neuralmonkey.encoders.transformer.TransformerEncoder attribute), 89
temporal_mask (neuralmonkey.encoders.attentive.AttentiveEncoder attribute), 67
temporal_mask (neuralmonkey.encoders.cnn_encoder.CNNTemporalView attribute), 69
temporal_mask (neuralmonkey.encoders.facebook_conv.SentenceEncoder attribute), 71
temporal_mask (neuralmonkey.encoders.raw_rnn_encoder.RawRNNEncoder attribute), 78
temporal_mask (neuralmonkey.encoders.recurrent.RecurrentEncoder attribute), 82
temporal_mask (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 85
temporal_mask (neuralmonkey.encoders.transformer.TransformerEncoder attribute), 89
temporal_mask (neuralmonkey.encoders.transformer.TransformerLayer attribute), 90
temporal_states (neuralmonkey.encoders.attentive.AttentiveEncoder attribute), 67
temporal_states (neuralmonkey.encoders.cnn_encoder.CNNTemporalView attribute), 69
temporal_states (neuralmonkey.encoders.facebook_conv.SentenceEncoder attribute), 71
temporal_states (neuralmonkey.encoders.raw_rnn_encoder.RawRNNEncoder attribute), 78
temporal_states (neuralmonkey.encoders.recurrent.RecurrentEncoder attribute), 82
temporal_states (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 85
temporal_states (neuralmonkey.encoders.transformer.TransformerEncoder attribute), 89
temporal_states (neuralmonkey.encoders.transformer.TransformerLayer attribute), 90
tensor() (in module neuralmonkey.decorators), 121
TensorFlowManager (class in neuralmonkey.tf_manager), 127
TensorRunner (class in neuralmonkey.runners.tensor_runner), 108
TensorRunner.Executable (class in neuralmonkey.runners.tensor_runner), 108
TEREvaluator (class in neuralmonkey.evaluators.ter), 100
test_mt_trainer() (neuralmonkey.trainers.test_multitask_trainer.TestMultitaskTrainer method), 117
TestMP (class in neuralmonkey.trainers.test_multitask_trainer), 117
TestMultitaskTrainer (class in neural-

monkey.trainers.test_multitask_trainer), 117
tf_print() (in module neuralmonkey.tf_utils), 130
token_ids (neuralmonkey.decoders.beam_search_decoder.SearchResults attribute), 48
token_level_batching (neuralmonkey.dataset.BatchingScheme attribute), 118
total_precision_recall() (neuralmonkey.evaluators.gleu.GLEUEvaluator static method), 97
train() (neuralmonkey.experiment.Experiment method), 123
train_inputs (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
train_inputs (neuralmonkey.decoders.autoregressive.DecoderConstants attribute), 44
train_logits (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
train_logits (neuralmonkey.decoders.transformer.TransformerDecoder attribute), 64
train_logprobs (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
train_loop_result (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
train_loss (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
train_loss (neuralmonkey.decoders.classifier.Classifier attribute), 50
train_loss (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 51
train_loss (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 59
train_loss (neuralmonkey.decoders.sequence_regressor.SequenceRegressor attribute), 61
train_loss (neuralmonkey.decoders.word_alignment_decoder.WordAlignmentDecoder attribute), 66
train_mask (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
train_op (neuralmonkey.trainers.generic_trainer.GenericTrainer attribute), 114
train_output_states (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
train_targets (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 51
train_xents (neuralmonkey.decoders.autoregressive.AutoregressiveDecoder attribute), 44
training_loop() (in module neuralmonkey.learning_utils), 125
TransformerDecoder (class in neuralmonkey.decoders.transformer), 61
TransformerEncoder (class in neuralmonkey.encoders.transformer), 87
TransformerHistories (class in neuralmonkey.decoders.transformer), 64
TransformerLayer (class in neuralmonkey.encoders.transformer), 90
truncate() (neuralmonkey.vocabulary.Vocabulary method), 132
truncate_by_min_freq() (neuralmonkey.vocabulary.Vocabulary method), 132

U

update_initializers() (in module neuralmonkey.tf_utils), 130
update_initializers() (neuralmonkey.experiment.Experiment method), 123
use_leftover_buckets (neuralmonkey.dataset.BatchingScheme attribute), 119

V

validation_hook() (neuralmonkey.tf_manager.TensorFlowManager method), 128
var_list (neuralmonkey.trainers.generic_trainer.GenericTrainer attribute), 114
vectors_to_sentences() (neuralmonkey.vocabulary.Vocabulary method), 132
visualize_attention() (neuralmonkey.attention.base_attention.BaseAttention method), 23
visualize_attention() (neuralmonkey.attention.scaled_dot_product.MultiHeadAttention method), 32
visualize_attention() (neuralmonkey.attention.stateful_context.StatefulContext method), 37
visualize_embeddings() (in module neuralmonkey.experiment), 123
Vocabulary (class in neuralmonkey.vocabulary), 131
vocabulary (neuralmonkey.decoders.beam_search_decoder.BeamSearchDec attribute), 47

W

warn() (in module neuralmonkey.logging), 127
warn() (neuralmonkey.logging.Logging static method), 126
weight (neuralmonkey.trainers.generic_trainer.Objective attribute), 114
weights (neuralmonkey.attention.namedtuples.AttentionLoopState attribute), 29
WEREvaluator (class in neuralmonkey.evaluators.wer), 100

WordAlignmentDecoder (class in neuralmonkey.decoders.word_alignment_decoder), 65
WordAlignmentRunner (class in neuralmonkey.runners.word_alignment_runner), 109
WordAlignmentRunner.Executable (class in neuralmonkey.runners.word_alignment_runner), 109

X

xent_objective() (in module neuralmonkey.trainers.cross_entropy_trainer), 111