Neural Monkey Documentation

Release 0.1

Jindřich Libovický, Jindřich Helcl, Tomáš Musil

Jun 08, 2017

Contents

1 Getting Started **3**

Python Module Index **87**

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

CHAPTER 1

Getting Started

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!

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.

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.

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.

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.

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*.

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'`.

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.

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
    |   |== 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!

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.sentence_encoder.SentenceEncoder
rnn_size=300
max_input_len=50
embedding_size=300
dropout_keep_prob=0.8
attention_type=decoding_function.Attention
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.sentence_encoder.SentenceEncoder
rnn_size=300
max_input_len=50
embedding_size=300
dropout_keep_prob=0.8
attention_type=decoding_function.Attention
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.

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>]
rnn_size=300
max_output_len=50
embeddings_encoder=<trans_encoder>
dropout_keep_prob=0.8
use_attention=True
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`.

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>` means that the embeddings (including embedding size) are shared with `trans_encoder`.

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.

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.

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.

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!

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.

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
    |   \== 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.

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.sentence_encoder.SentenceEncoder
name="sentence_encoder"
rnn_size=300
max_input_len=50
embedding_size=300
dropout_keep_prob=0.8
attention_type=decoding_function.Attention
data_id="source_bpe"
vocabulary=<shared_vocabulary>

[decoder]
class=decoders.decoder.Decoder
name="decoder"
encoders=[<encoder>]
rnn_size=256
embedding_size=300
dropout_keep_prob=0.8
use_attention=True
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>
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
```

Part III. - Running and Evaluation of the Experiment

The training can be run as simply as:

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

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

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

[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.

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.

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>)`)

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]` section 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.

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>
...
```

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.

With the default `gpu_allow_growth=True`, it makes sense to monitor memory consumption. Neural Monkey can include a short summary total GPU memory used in the periodic log line. Just set:

```
report_gpu_memory_consumption=True
```

The log line will then contain the information like: `MiB:0:7971/8113,1:4283/8113`. This particular message means that there are two GPU cards and the one indexed 1 has 4283 out of the total 8113 MiB occupied. Note that the information reports all GPUs on the machine, regardless `CUDA_VISIBLE_DEVICES`.

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.

API Documentation

neuralmonkey package

The neuralmonkey package is the root package of this project.

Sub-modules

neuralmonkey

neuralmonkey package

Subpackages

neuralmonkey.config package

Submodules

neuralmonkey.config.builder module

This module is responsible for instantiating objects specified by the experiment configuration

class neuralmonkey.config.builder.**ClassSymbol**(*string: str*) → None

Bases: object

Represents a class (or other callable) in configuration.

create() → typing.Any

neuralmonkey.config.builder.**build_config**(*config_dicts: typing.Dict[str, typing.Any], ignore_names: typing.Set[str], warn_unused: bool = False*) → typing.Dict[str, typing.Any]

Builds the model from the configuration

Parameters

- **config_dicts** – The parsed configuration file

- **ignore_names** – A set of names that should be ignored during the loading.
- **warn_unused** – Emit a warning if there are unused sections.

neuralmonkey.config.builder.**build_object**(*value: str, all_dicts: typing.Dict[str, typing.Any], existing_objects: typing.Dict[str, typing.Any], depth: int*) → typing.Any

Builds an object from config dictionary of its arguments. It works recursively.

Parameters

- **value** – Value that should be resolved (either a literal value or a config section name)
- **all_dicts** – Configuration dictionaries used to find configuration of unconstructed objects.
- **existing_objects** – Dictionary of already constructed objects.
- **ignore_names** – Set of names that shoud be ignored.
- **depth** – The current depth of recursion. Used to prevent an infinite
- **recursion.** –

neuralmonkey.config.builder.**instantiate_class**(*name: str, all_dicts: typing.Dict[str, typing.Any], existing_objects: typing.Dict[str, typing.Any], depth: int*) → typing.Any

Instantiate a class from the configuration

Arguments: see help(build_object)

neuralmonkey.config.configuration module

class neuralmonkey.config.configuration.**Configuration**

Bases: object

Loads the configuration file in an analogical way the python's argparse.ArgumentParser works.

add_argument(*name: str, required: bool = False, default: typing.Any = None, cond: typing.Callable[[typing.Any], bool] = None*) → None

build_model(*warn_unused=False*) → None

ignore_argument(*name: str*) → None

load_file(*path: str, changes: typing.Union[typing.List[str], NoneType] = None*) → None

make_namespace(*d_obj*) → argparse.Namespace

save_file(*path: str*) → None

neuralmonkey.config.exceptions module

Module that contains exceptions handled in config parsing and loading

exception neuralmonkey.config.exceptions.**ConfigBuildException**(*object_name: str, original_exception: Exception*) → None

Bases: Exception

Exception caused by error in loading the model

exception neuralmonkey.config.exceptions.**ConfigInvalidValueException**(*value: typing.Any*, *message: str*) → None

Bases: Exception

exception neuralmonkey.config.exceptions.**IniError**(*line: int*, *message: str*, *original_exc: typing.Union[Exception, NoneType] = None*) → None

Bases: Exception

Exception caused by error in INI file syntax

neuralmonkey.config.parsing module

Module responsible for INI parsing

neuralmonkey.config.parsing.**parse_file**(*config_file: typing.Iterable[str]*, *changes: typing.Union[typing.Iterable[str], NoneType] = None*) → typing.Tuple[typing.Dict[str, typing.Any], typing.Dict[str, typing.Any]]

Parses an INI file and creates all values

neuralmonkey.config.parsing.**write_file**(*config_dict: typing.Dict[str, typing.Any]*, *config_file: typing.IO[str]*) → None

neuralmonkey.config.utils module

This module contains helper functions that are suppoosed to be called from the configuration file because calling the functions or the class constructors directly would be inconvinent or impossible.

neuralmonkey.config.utils.**adadelta_optimizer**(***kwargs*) → tensorflow.python.training.adadelta.AdadeltaOptimizer

neuralmonkey.config.utils.**adam_optimizer**(*learning_rate: float = 0.0001*) → tensorflow.python.training.adam.AdamOptimizer

neuralmonkey.config.utils.**dataset_from_files**(**args*, ***kwargs*) → T

neuralmonkey.config.utils.**deprecated**(*func: typing.Callable[..., T]*) → typing.Callable[..., T]

neuralmonkey.config.utils.**variable**(*initial_value=0*, *trainable: bool = False*, ***kwargs*) → tensorflow.python.ops.variables.Variable

neuralmonkey.config.utils.**vocabulary_from_bpe**(**args*, ***kwargs*) → T

neuralmonkey.config.utils.**vocabulary_from_dataset**(**args*, ***kwargs*) → T

neuralmonkey.config.utils.**vocabulary_from_file**(**args*, ***kwargs*) → T

Module contents

neuralmonkey.decoders package

Submodules

neuralmonkey.decoders.beam_search_decoder module

class neuralmonkey.decoders.beam_search_decoder.**BeamSearchDecoder**(*name: str*, *parent_decoder: neuralmonkey.decoders.decoder.Decoder*, *beam_size: int*, *length_normalization: float*, *max_steps: int = None*, *save_checkpoint: str = None*, *load_checkpoint: str = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*

In-graph beam search for batch size 1.

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

beam_size

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

step(*att_objects: typing.List[neuralmonkey.decoding_function.BaseAttention]*, *bs_state: neuralmonkey.decoders.beam_search_decoder.SearchState*) → typing.Tuple[neuralmonkey.decoders.beam_search_decoder.SearchState, neuralmonkey.decoders.beam_search_decoder.SearchStepOutput]

vocabulary

class neuralmonkey.decoders.beam_search_decoder.**SearchState**(*logprob_sum*, *lengths*, *finished*, *last_word_ids*, *last_state*, *last_attns*)

Bases: tuple

finished
Alias for field number 2

last_attns
Alias for field number 5

last_state
Alias for field number 4

last_word_ids
Alias for field number 3

lengths
Alias for field number 1

logprob_sum
Alias for field number 0

class neuralmonkey.decoders.beam_search_decoder.**SearchStepOutput**(*scores*, *parent_ids*, *token_ids*)

Bases: tuple

parent_ids
Alias for field number 1

scores
Alias for field number 0

token_ids
Alias for field number 2

neuralmonkey.decoders.ctc_decoder module

class neuralmonkey.decoders.ctc_decoder.**CTCDecoder**(*name: str*, *encoder: typing.Any*, *vocabulary: neuralmonkey.vocabulary.Vocabulary*, *data_id: str*, *merge_repeated_targets: bool = False*, *merge_repeated_outputs: bool = True*, *beam_width: int = 1*, *save_checkpoint: typing.Union[str, NoneType] = None*, *load_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*

Connectionist Temporal Classification.

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

cost

decoded

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

input_lengths

logits

runtime_loss

train_loss

train_mode

train_targets

neuralmonkey.decoders.decoder module

class neuralmonkey.decoders.decoder.**Decoder**(*encoders: typing.List[typing.Any], vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, name: str, max_output_len: int, dropout_keep_prob: float = 1.0, rnn_size: typing.Union[int, NoneType] = None, embedding_size: typing.Union[int, NoneType] = None, output_projection: typing.Union[typing.Callable[[tensorflow.python.framework.ops.Tensor, tensorflow.python.framework.ops.Tensor, typing.List[tensorflow.python.framework.ops.Tensor]], tensorflow.python.framework.ops.Tensor], NoneType] = None, encoder_projection: typing.Union[typing.Callable[[tensorflow.python.framework.ops.Tensor, typing.Union[int, NoneType], typing.Union[typing.List[typing.Any], NoneType]], tensorflow.python.framework.ops.Tensor], NoneType] = None, use_attention: bool = False, embeddings_encoder: typing.Any = None, attention_on_input: bool = True, rnn_cell: str = 'GRU', conditional_gru: bool = False, save_checkpoint: typing.Union[str, NoneType] = None, load_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*

A class that manages parts of the computation graph that are used for the decoding.

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

Embed the input using the embedding matrix and apply dropout

Parameters **inputs** – The Tensor to be embedded and dropped out.

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

get_attention_object(*encoder, train_mode: bool*)

step(*att_objects: typing.List[neuralmonkey.decoding_function.BaseAttention], input_: tensorflow.python.framework.ops.Tensor, prev_state: tensorflow.python.framework.ops.Tensor, prev_attns: typing.List[tensorflow.python.framework.ops.Tensor]*)

neuralmonkey.decoders.encoder_projection module

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

neuralmonkey.decoders.encoder_projection.**concat_encoder_projection**(*train_mode: tensorflow.python.framework.ops.Tensor, rnn_size: typing.Union[int, NoneType] = None, encoders: typing.Union[typing.List[typing.Any], NoneType] = None*) → tensorflow.python.framework.ops.Tensor

Create the initial state by concatenating the encoders' encoded values

Parameters

- **train_mode** – tf 0-D bool Tensor specifying the training mode (not used)
- **rnn_size** – The size of the resulting vector (not used)
- **encoders** – The list of encoders

neuralmonkey.decoders.encoder_projection.**empty_initial_state**(*train_mode: tensorflow.python.framework.ops.Tensor, rnn_size: typing.Union[int, NoneType], encoders: typing.Union[typing.List[typing.Any], NoneType] = None*) → tensorflow.python.framework.ops.Tensor

Return an empty vector

Parameters

- **train_mode** – tf 0-D bool Tensor specifying the training mode (not used)
- **rnn_size** – The size of the resulting vector
- **encoders** – The list of encoders (not used)

neuralmonkey.decoders.encoder_projection.**linear_encoder_projection**(*dropout_keep_prob: float*) → typing.Callable[[tensorflow.python.framework.ops.Tensor, typing.Union[int, NoneType], typing.Union[typing.List[typing.Any], NoneType]], tensorflow.python.framework.ops.Tensor]

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.output_projection module

This module contains different variants of projection functions for RNN outputs.

neuralmonkey.decoders.output_projection.**maxout_output**(*maxout_size*)

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*, *dropout_keep_prob=None*, *train_mode: tensorflow.python.framework.ops.Tensor = None*, *activation=<function tanh>*)

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_plc** – Dropout placeholder. TODO this is not going to work with current configuration
- **activation** – The activation function to use in each layer.

neuralmonkey.decoders.output_projection.**no_deep_output**(*prev_state*, *prev_output*, *ctx_tensors*)

Compute RNN output out of the previous state and output, and the context tensors returned from attention mechanisms.

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

Parameters

- **prev_state** – Previous decoder RNN state. (Denoted s_i-1)
- **prev_output** – Embedded output of the previous step. (y_i-1)
- **ctx_tensors** – Context tensors computed by the attentions. (c_i)

Returns This function returns the concatenation of all its inputs.

neuralmonkey.decoders.sequence_classifier module

class neuralmonkey.decoders.sequence_classifier.**SequenceClassifier**(*name: str, encoders: typing.List[typing.Any], vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, layers: typing.List[int], activation_fn: typing.Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor] = <function relu>, dropout_keep_prob: float = 0.5, save_checkpoint: typing.Union[str, NoneType] = None, load_checkpoint: typing.Union[str, NoneType] = 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.

cost

decoded

decoded_logits

decoded_seq

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

gt_inputs

loss_with_decoded_ins

loss_with_gt_ins

runtime_logprobs

runtime_loss

train_loss

train_mode

neuralmonkey.decoders.sequence_labeler module

class neuralmonkey.decoders.sequence_labeler.**SequenceLabeler**(*name: str*, *encoder: neuralmonkey.encoders.sentence_encoder.SentenceEn*, *vocabulary: neuralmonkey.vocabulary.Vocabulary*, *data_id: str*, *dropout_keep_prob: float = 1.0*, *save_checkpoint: typing.Union[str, NoneType] = None*, *load_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*

Classifier assing a label to each encoder's state.

cost

decoded

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

logits

logprobs

runtime_loss

train_loss

train_mode

train_targets

train_weights

neuralmonkey.decoders.sequence_regressor module

class neuralmonkey.decoders.sequence_regressor.**SequenceRegressor**(*name: str, encoders: typing.List[typing.Any], data_id: str, layers: typing.List[int] = None, activation_fn: typing.Callable[[tensorflow.python.framework.ops.Tensor] = <function relu>, dropout_keep_prob: float = 1.0, dimension: int = 1, save_checkpoint: str = None, load_checkpoint: str = 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.

cost

decoded

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

predictions

runtime_loss

train_inputs

train_loss

train_mode

neuralmonkey.decoders.word_alignment_decoder module

class neuralmonkey.decoders.word_alignment_decoder.**WordAlignmentDecoder**(*encoder: neuralmonkey.encoders.sentence_enc*, *decoder: neuralmonkey.decoders.decoder.Deco*, *data_id: str*, *name: str*) → 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.

cost

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

Module contents

neuralmonkey.encoders package

Submodules

neuralmonkey.encoders.attentive module

class neuralmonkey.encoders.attentive.**Attentive**(*attention_type*, ***kwargs*)

Bases: object

A base class fro an attentive part of graph (typically encoder).

Objects inheriting this class are able to generate an attention object that allows a decoder to perform attention over an attention_object provided by the encoder (e.g., input word representations in case of MT or convolutional maps in case of image captioning).

create_attention_object()

Attention object that can be used in decoder.

neuralmonkey.encoders.cnn_encoder module

CNN for image processing.

class neuralmonkey.encoders.cnn_encoder.**CNNEncoder**(*name: str, data_id: str, convolutions: typing.List[typing.Tuple[int, int, typing.Union[int, NoneType]]], image_height: int, image_width: int, pixel_dim: int, fully_connected: typing.Union[typing.List[int], NoneType] = None, dropout_keep_prob: float = 0.5, attention_type: typing.Type = <class 'neuralmonkey.decoding_function.Attention'>, save_checkpoint: typing.Union[str, NoneType] = None, load_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*, *neuralmonkey.encoders.attentive.Attentive*

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.

encoded
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.

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

image_input

image_processing_layers
Do all convolutions and return the last conditional map.

Applies convolutions on the input tensor with optional max pooling. All the intermediate layers are stored in the *image_processing_layers* attribute. There is not dropout between the convolutional layers, by default the activation function is ReLU.

states

train_mode

neuralmonkey.encoders.encoder_wrapper 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.encoders.encoder_wrapper.**EncoderWrapper**(*name: str, encoders: typing.List[typing.Any], attention_type: typing.Type, attention_state_size: int, use_sentinels=False, share_attn_projections=False*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*, *neuralmonkey.encoders.attentive.Attentive*

Wrapper doing attention combination behaving as a single encoder.

This class wraps encoders and performs the attention combination in such a way that for the decoder, it looks like a single encoder capable to generate a single context vector.

create_attention_object()

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

class neuralmonkey.encoders.encoder_wrapper.**FlatMultiAttention**(**args, **kwargs*)

Bases: *neuralmonkey.encoders.encoder_wrapper.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.

attention(*decoder_state*, *decoder_prev_state*, *decoder_input*)

get_encoder_projections(*scope*)

class neuralmonkey.encoders.encoder_wrapper.**HierarchicalMultiAttention**(**args, **kwargs*) → None

Bases: *neuralmonkey.encoders.encoder_wrapper.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.

attention(*decoder_state*, *decoder_prev_state*, *decoder_input*)

class neuralmonkey.encoders.encoder_wrapper.**MultiAttention**(*encoders: typing.List[neuralmonkey.encoders.attentive.Attentive*, *attention_state_size: int, scope: typing.Union[tensorflow.python.ops.variable_scope.V str], share_projections: bool = False, use_sentinels: bool = False*) → None

Bases: *neuralmonkey.decoding_function.BaseAttention*

Base class for attention combination.

attention(*decoder_state*, *decoder_prev_state*, *decoder_input*)
Get context vector for given decoder state.

attn_size

neuralmonkey.encoders.factored_encoder module

class neuralmonkey.encoders.factored_encoder.**FactoredEncoder**(*name: str, max_input_len: int, vocabularies: typing.List[neuralmonkey.vocabulary.Vocabulary], data_ids: typing.List[str], embedding_sizes: typing.List[int], rnn_size: int, dropout_keep_prob: float = 1.0, attention_type: typing.Any = None, save_checkpoint: typing.Union[str, NoneType] = None, load_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*, *neuralmonkey.encoders.attentive.Attentive*

Generic encoder processing an arbitrary number of input sequences.

feed_dict(*dataset*, *train=False*)

neuralmonkey.encoders.imagenet_encoder module

Pre-trained ImageNet networks.

class neuralmonkey.encoders.imagenet_encoder.**ImageNet**(*name: str, data_id: str, network_type: str, attention_layer: typing.Union[str, NoneType] = None, attention_state_size: typing.Union[int, NoneType] = None, attention_type: typing.Type = <class 'neuralmonkey.decoding_function.Attention'>, fine_tune: bool = False, encoded_layer: typing.Union[str, NoneType] = None, load_checkpoint: typing.Union[str, NoneType] = None, save_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*, *neuralmonkey.encoders.attentive.Attentive*

Pre-trained ImageNet network.

HEIGHT = 224

WIDTH = 224

cnn_states

encoded

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

input_image

states

neuralmonkey.encoders.numpy_encoder module

class neuralmonkey.encoders.numpy_encoder.**PostCNNImageEncoder**(*name: str*, *input_shape: typing.List[int]*, *output_shape: int*, *data_id: str*, *attention_type: typing.Callable = None*, *save_checkpoint: typing.Union[str, NoneType] = None*, *load_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*, *neuralmonkey.encoders.attentive.Attentive*

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

class neuralmonkey.encoders.numpy_encoder.**VectorEncoder**(*name: str*, *dimension: int*, *data_id: str*, *output_shape: int = None*, *save_checkpoint: str = None*, *load_checkpoint: str = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*

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

neuralmonkey.encoders.raw_rnn_encoder module

class neuralmonkey.encoders.raw_rnn_encoder.**RNNSpec**(*size*, *direction*, *cell_type*)

Bases: tuple

cell_type

Alias for field number 2

direction
Alias for field number 1

size
Alias for field number 0

class neuralmonkey.encoders.raw_rnn_encoder.**RawRNNEncoder**(*name: str, data_id: str, input_size: int, rnn_layers: typing.List[typing.Union[typing.Tuple[int], typing.Tuple[int, str], typing.Tuple[int, str, str]]], max_input_len: typing.Union[int, NoneType] = None, dropout_keep_prob: float = 1.0, attention_type: typing.Any = None, save_checkpoint: typing.Union[str, NoneType] = None, load_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*, *neuralmonkey.encoders.attentive.Attentive*

A raw RNN encoder that gets input as a tensor.

feed_dict(*dataset: neuralmonkey.dataset.Dataset, train: bool = False*) → typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Any]
Populate the feed dictionary with the encoder inputs.

Parameters

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

neuralmonkey.encoders.sentence_cnn_encoder module

Encoder for sentences withou explicit segmentation.

class neuralmonkey.encoders.sentence_cnn_encoder.**SentenceCNNEncoder**(*name: str, vocabulary: neuralmonkey.vocabulary.Vocabulary, data_id: str, embedding_size: int, segment_size: int, highway_depth: int, rnn_size: int, filters: typing.List[typing.Tuple[int, int]], max_input_len: typing.Union[int, NoneType] = None, dropout_keep_prob: float = 1.0, attention_type: typing.Any = None, attention_fertility: int = 3, use_noisy_activations: bool = False, save_checkpoint: typing.Union[str, NoneType] = None, load_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*, *neuralmonkey.encoders.attentive.Attentive*

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 (https://arxiv.org/pdf/1610.03017.pdf)

bidirectional_rnn

cnn_encoded

1D convolution with max-pool that processing characters.

embedded_inputs

encoded

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

Populate the feed dictionary with the encoder inputs.

Encoder input placeholders:

encoder_input: Stores indices to the vocabulary, shape (batch, time)

encoder_padding: Stores the padding (ones and zeros, indicating valid words and positions after the end of sentence, shape (batch, time)

train_mode: Boolean scalar specifying the mode (train vs runtime)

Parameters

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

highway_layer

Highway net projection following the CNN.

input_mask

inputs

rnn_cells() → typing.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

sentence_lengths

states

train_mode

vocabulary_size

neuralmonkey.encoders.sentence_encoder module

class neuralmonkey.encoders.sentence_encoder.**SentenceEncoder**(*name: str*, *vocabulary: neuralmonkey.vocabulary.Vocabulary*, *data_id: str*, *embedding_size: int*, *rnn_size: int*, *attention_state_size: typing.Union[int, NoneType] = None*, *max_input_len: typing.Union[int, NoneType] = None*, *dropout_keep_prob: float = 1.0*, *attention_type: type = None*, *attention_fertility: int = 3*, *use_noisy_activations: bool = False*, *parent_encoder: typing.Union[typing.SentenceEncoder, NoneType] = None*, *save_checkpoint: typing.Union[str, NoneType] = None*, *load_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*, *neuralmonkey.encoders.attentive.Attentive*

A class that manages parts of the computation graph that are used for encoding of input sentences. It uses a bidirectional RNN.

This version of the encoder does not support factors. Should you want to use them, use FactoredEncoder instead.

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

Populate the feed dictionary with the encoder inputs.

Encoder input placeholders:

encoder_input: Stores indices to the vocabulary, shape (batch, time)

encoder_padding: Stores the padding (ones and zeros, indicating valid words and positions after the end of sentence, shape (batch, time)

train_mode: Boolean scalar specifying the mode (train vs runtime)

Parameters

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

rnn_cells() → typing.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

states_mask

vocabulary_size

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: typing.List[typing.Tuple[int, int]], max_input_len: typing.Union[int, NoneType] = None, dropout_keep_prob: float = 1.0, save_checkpoint: typing.Union[str, NoneType] = None, load_checkpoint: typing.Union[str, NoneType] = None*) → None

Bases: *neuralmonkey.model.model_part.ModelPart*

Encoder processing a sequence using a CNN.

embedded_inputs

encoded

feed_dict(*dataset: neuralmonkey.dataset.Dataset, train: bool = False*) → typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Any]
Populate the feed dictionary with the encoder inputs.

Encoder input placeholders:

encoder_input: Stores indices to the vocabulary, shape (batch, time)

encoder_padding: Stores the padding (ones and zeros, indicating valid words and positions after the end of sentence, shape (batch, time)

train_mode: Boolean scalar specifying the mode (train vs runtime)

Parameters

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

input_mask

inputs

train_mode

Module contents

neuralmonkey.evaluators package

Submodules

neuralmonkey.evaluators.accuracy module

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

Bases: object

static compare_scores(*score1: float*, *score2: float*) → int

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

Bases: object

static compare_scores(*score1: float*, *score2: float*) → int

neuralmonkey.evaluators.average module

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

Bases: object

Just average the numeric output of a runner.

neuralmonkey.evaluators.beer module

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

Bases: object

Wrapper for BEER scorer.

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

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

neuralmonkey.evaluators.bleu module

class neuralmonkey.evaluators.bleu.**BLEUEvaluator**(*n: int = 4*, *deduplicate: bool = False*, *name: typing.Union[str, NoneType] = None*) → None

Bases: object

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

Computes BLEU on a corpus with multiple references using uniform weights. 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 compare_scores(*score1: float*, *score2: float*) → int

static deduplicate_sentences(*sentences: typing.List[typing.List[str]]*) → typing.List[typing.List[str]]

static effective_reference_length(*hypotheses: typing.List[typing.List[str]]*, *references_list: typing.List[typing.List[typing.List[str]]]*) → int

Computes the effective reference corpus length (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(*counters: typing.List[collections.Counter]*) → collections.Counter

Merge counters using maximum values

static minimum_reference_length(*hypotheses: typing.List[typing.List[str]]*, *references_list: typing.List[typing.List[typing.List[str]]]*) → int

Computes the effective reference corpus length (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(*hypotheses: typing.List[typing.List[str]]*, *references_list: typing.List[typing.List[typing.List[typing.List[str]]]]*, *n: int*, *case_sensitive: bool*) → typing.Tuple[float, int]

Computes 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 (*sentence: typing.List[str], 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

neuralmonkey.evaluators.bleu_ref module

class neuralmonkey.evaluators.bleu_ref.**BLEUReferenceImplWrapper** (*wrapper*, *name='BLEU'*, *encoding='utf-8'*)

Bases: object

Wrapper for TectoMT's wrapper for reference NIST and BLEU scorer

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

neuralmonkey.evaluators.edit_distance module

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

Bases: object

static compare_scores (*score1: float*, *score2: float*) → int

static ratio (*str1: str*, *str2: str*) → float

neuralmonkey.evaluators.f1_bio module

class neuralmonkey.evaluators.f1_bio.**F1Evaluator** (*name: str = 'F1 measure'*) → None

Bases: object

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 (*seq: typing.List[str]*) → typing.Set[str]

static f1_score (*decoded: typing.List[str], reference: typing.List[str]*) → float

neuralmonkey.evaluators.gleu module

class neuralmonkey.evaluators.gleu.**GLEUEvaluator** (*n: int = 4*, *deduplicate: bool = False*, *name: typing.Union[str, NoneType] = None*) → None

Bases: object

Sentence-level evaluation metric that correlates 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.

static gleu(*hypotheses: typing.List[typing.List[str]], references: typing.List[typing.List[typing.List[str]]], ngrams: int = 4, case_sensitive: bool = True*) → float

Computes 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.

static total_precision_recall(*hypotheses: typing.List[typing.List[str]], references_list: typing.List[typing.List[typing.List[str]]], ngrams: int, case_sensitive: bool*) → typing.Tuple[float, float]

Computes the modified n-gram precision and recall 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)
- **ngrams** – n-gram order
- **case_sensitive** – Whether to perform case-sensitive computation

neuralmonkey.evaluators.mse module

class neuralmonkey.evaluators.mse.**MeanSquaredErrorEvaluator**(*name: str = 'MeanSquaredError'*) → None

Bases: object

static compare_scores(*score1: float*, *score2: float*) → int

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: object

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

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

neuralmonkey.evaluators.ter module

class neuralmonkey.evaluators.ter.**TEREvalutator**(*name: str = 'TER'*) → None
Bases: object

Compute TER using the pyter library.

neuralmonkey.evaluators.wer module

class neuralmonkey.evaluators.wer.**WEREvaluator**(*name: str = 'WER'*) → None
Bases: object

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

Module contents

neuralmonkey.model package

Submodules

neuralmonkey.model.model_part module

Basic functionality of all model parts.

class neuralmonkey.model.model_part.**ModelPart**(*name: str*, *save_checkpoint: typing.Union[str, NoneType] = None*, *load_checkpoint: typing.Union[str, NoneType] = None*) → None
Bases: object

Base class of all model parts.

feed_dict(*dataset: neuralmonkey.dataset.Dataset*, *train: bool*) → typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Any]
Prepare feed dicts for part's placeholders from a dataset.

load(*session: tensorflow.python.client.session.Session*) → None
Load model part from a checkpoint file.

name
Name of the model part and its variable scope.

save(*session: tensorflow.python.client.session.Session*) → None
Save model part to a checkpoint file.

use_scope()
Return a context manager that (re)opens the model part's variable and name scope.

Module contents

neuralmonkey.nn package

Submodules

neuralmonkey.nn.highway module

This module implements the highway networks.

neuralmonkey.nn.highway.**highway**(*inputs*, *activation=<function relu>*, *scope='HighwayNetwork'*)

Simple highway layer

y = H(x, Wh) * T(x, Wt) + x * C(x, Wc)

where:

C(x, Wc) = 1 - T(x, Wt)

Parameters

- **inputs** – A tensor or list of tensors. It should be 2D tensors with equal length in the first dimension (batch size)
- **activation** – Activation function of the linear part of the formula H(x, Wh).
- **scope** – The name of the scope used for the variables.

Returns A tensor of shape tf.shape(inputs)

neuralmonkey.nn.mlp module

class neuralmonkey.nn.mlp.**MultilayerPerceptron**(*mlp_input: tensorflow.python.framework.ops.Tensor*, *layer_configuration: typing.List[int]*, *dropout_keep_prob: float*, *output_size: int*, *train_mode: tensorflow.python.framework.ops.Tensor*, *activation_fn: typing.Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor] = <function relu>*, *name: str = 'multilayer_perceptron'*) → None

Bases: object

General implementation of the multilayer perceptron.

classification

softmax

neuralmonkey.nn.noisy_gru_cell module

class neuralmonkey.nn.noisy_gru_cell.**NoisyGRUCell**(*num_units: int*, *training*) → None

Bases: tensorflow.python.ops.rnn_cell_impl._RNNCell

Gated Recurrent Unit cell (cf. http://arxiv.org/abs/1406.1078) with noisy activation functions (http://arxiv.org/abs/1603.00391). The theano code is availble at https://github.com/caglar/noisy_units.

It is based on the TensorFlow implementatin of GRU just the activation function are changed for the noisy ones.

output_size

state_size

neuralmonkey.nn.noisy_gru_cell.**noisy_activation**(*x*, *generic*, *linearized*, *training*, *alpha: float = 1.1*, *c: float = 0.5*)

Implements the noisy activation with Half-Normal Noise for Hard-Saturation functions. See http://arxiv.org/abs/1603.00391, Algorithm 1.

Parameters

- **x** – Tensor which is an input to the activation function
- **generic** – The generic formulation of the activation function. (denoted as h in the paper)
- **linearized** – Linearization of the activation based on the first-order Tailor expansion around zero. (denoted as u in the paper)
- **training** – A boolean tensor telling whether we are in the training stage (and the noise is sampled) or in runtime when the expactation is used instead.
- **alpha** – Mixing hyper-parameter. The leakage rate from the linearized function to the nonlinear one.
- **c** – Standard deviation of the sampled noise.

neuralmonkey.nn.noisy_gru_cell.**noisy_sigmoid**(*x*, *training*)

neuralmonkey.nn.noisy_gru_cell.**noisy_tanh**(*x*, *training*)

neuralmonkey.nn.ortho_gru_cell module

class neuralmonkey.nn.ortho_gru_cell.**OrthoGRUCell**(*num_units*, *input_size=None*, *activation=<function tanh>*, *reuse=None*)

Bases: tensorflow.contrib.rnn.python.ops.core_rnn_cell_impl.GRUCell

Classic GRU cell but initialized using random orthogonal matrices

neuralmonkey.nn.pervasive_dropout_wrapper module

class neuralmonkey.nn.pervasive_dropout_wrapper.**PervasiveDropoutWrapper**(*cell*, *mask*, *scale*) → None

Bases: tensorflow.python.ops.rnn_cell_impl._RNNCell

output_size

state_size

neuralmonkey.nn.projection module

This module implements various types of projections.

neuralmonkey.nn.projection.**linear**(*inputs: tensorflow.python.framework.ops.Tensor*, *size: int*, *scope: str = 'LinearProjection'*)

Simple linear projection

y = Wx + b

Parameters

- **inputs** – A tensor or list of tensors. It should be 2D tensors with equal length in the first dimension (batch size)
- **size** – The size of dimension 1 of the output tensor.
- **scope** – The name of the scope used for the variables.

Returns A tensor of shape batch x size

neuralmonkey.nn.projection.**maxout**(*inputs: tensorflow.python.framework.ops.Tensor*, *size: int*, *scope: str = 'MaxoutProjection'*)

Implementation of Maxout layer (Goodfellow et al., 2013) http://arxiv.org/pdf/1302.4389.pdf

z = Wx + b y_i = max(z_{2i-1}, z_{2i})

Parameters

- **inputs** – A tensor or list of tensors. It should be 2D tensors with equal length in the first dimension (batch size)
- **size** – The size of dimension 1 of the output tensor.
- **scope** – The name of the scope used for the variables

Returns A tensor of shape batch x size

neuralmonkey.nn.projection.**multilayer_projection**(*input_: tensorflow.python.framework.ops.Tensor*, *layer_sizes: typing.List[int]*, *train_mode: tensorflow.python.framework.ops.Tensor*, *activation: typing.Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor] = <function relu>*, *dropout_keep_prob: float = 1.0*, *scope: str = 'mlp'*)

neuralmonkey.nn.projection.**nonlinear**(*inputs: tensorflow.python.framework.ops.Tensor*, *size: int*, *activation: typing.Callable[[tensorflow.python.framework.ops.Tensor], tensorflow.python.framework.ops.Tensor]*, *scope: str = 'NonlinearProjection'*)

Linear projection with non-linear activation function

y = activation(Wx + b)

Parameters

- **inputs** – A tensor or list of tensors. It should be 2D tensors with equal length in the first dimension (batch size)
- **size** – The size of the second dimension (index 1) of the output tensor
- **scope** – The name of the scope used for the variables

Returns A tensor of shape batch x size

neuralmonkey.nn.utils module

This module provides utility functions used across the package.

neuralmonkey.nn.utils.**dropout**(*variable: tensorflow.python.framework.ops.Tensor*, *keep_prob: float*, *train_mode: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor

Performs dropout on a variable, depending on mode.

Parameters

- **variable** – The variable to be dropped out
- **keep_prob** – The probability of keeping a value in the variable
- **train_mode** – A bool Tensor specifying whether to dropout or not

Module contents

neuralmonkey.processors package

Submodules

neuralmonkey.processors.alignment module

class neuralmonkey.processors.alignment.**WordAlignmentPreprocessor**(*source_len*, *target_len*, *dtype=<class 'numpy.float32'>*, *normalize=True*, *zero_based=True*)

Bases: object

A preprocessor for word alignments in a text format.

One of the following formats is expected:

s1-t1 s2-t2 ...

s1:1/w1 s2:t2/w2 ...

where each *s* and *t* is the index of a word in the source and target sentence, respectively, and *w* is the corresponding weight. If the weight is not given, it is assumend to be 1. The separators - and *:* are interchangeable.

The output of the preprocessor is an alignment matrix of the fixed shape (target_len, source_len) for each sentence.

neuralmonkey.processors.bpe module

class neuralmonkey.processors.bpe.**BPEPostprocessor**(*separator: str = '@@'*) → None

Bases: object

decode(*sentence: typing.List[str]*) → typing.List[str]

class neuralmonkey.processors.bpe.**BPEPreprocessor**(*merge_file: str*, *separator: str = '@@'*, *encoding: str = 'utf-8'*) → None

Bases: object

Wrapper class for Byte-Pair Encoding.

Paper: https://arxiv.org/abs/1508.07909 Code: https://github.com/rsennrich/subword-nmt

neuralmonkey.processors.editops module

class neuralmonkey.processors.editops.**Postprocess**(*source_id: str, edits_id: str, result_postprocess: typing.Callable[[typing.Iterable[typing.List[str]]], typing.Iterable[typing.List[str]]] = None*) → None

Bases: object

Proprocessor applying edit operations on a series.

class neuralmonkey.processors.editops.**Preprocess**(*source_id: str*, *target_id: str*) → None

Bases: object

Preprocessor transorming two series into series of edit operations.

neuralmonkey.processors.editops.**convert_to_edits**(*source: typing.List[str], target: typing.List[str]*) → typing.List[str]

neuralmonkey.processors.editops.**reconstruct**(*source: typing.List[str], edits: typing.List[str]*) → typing.List[str]

neuralmonkey.processors.german module

class neuralmonkey.processors.german.**GermanPostprocessor**(*compounding=True*, *contracting=True*, *pronouns=True*)

Bases: object

decode(*sentence*)

class neuralmonkey.processors.german.**GermanPreprocessor**(*compounding=True*, *contracting=True*, *pronouns=True*)

Bases: object

neuralmonkey.processors.helpers module

neuralmonkey.processors.helpers.**pipeline**(*processors: typing.List[typing.Callable]*) → typing.Callable

Concatenate processors.

neuralmonkey.processors.helpers.**postprocess_char_based**(*sentences: typing.List[typing.List[str]]*) → typing.List[typing.List[str]]

neuralmonkey.processors.helpers.**preprocess_char_based**(*sentence: typing.List[str]*) → typing.List[str]

neuralmonkey.processors.helpers.**untruecase**(*sentences: typing.List[typing.List[str]]*) → typing.Generator[[typing.List[str], NoneType], NoneType]

neuralmonkey.processors.speech module

Module contents

neuralmonkey.readers package

Submodules

neuralmonkey.readers.audio_reader module

neuralmonkey.readers.image_reader module

neuralmonkey.readers.image_reader.**image_reader**(*prefix=''*, *pad_w: typing.Union[int, NoneType] = None*, *pad_h: typing.Union[int, NoneType] = None*, *rescale: bool = False*, *mode: str = 'RGB'*) → typing.Callable

Get a reader of images loading them from a list of pahts.

Parameters

- **prefix** – Prefix of the paths that are listed in a image files.
- **pad_w** – Width to which the images will be padded/cropped/resized.
- **pad_h** – Height to with the images will be padded/corpped/resized.
- **rescale** – If true, bigger images will be rescaled to the pad_w x pad_h size. Otherwise, they will be cropped from the middle.
- **mode** – Scipy image loading mode, see scipy documentation for more details.

Returns The reader function that takes a list of image paths (relative to provided prefix) and returns a list of images as numpy arrays of shape pad_h x pad_w x number of channels.

neuralmonkey.readers.image_reader.**imagenet_reader**(*prefix: str*, *target_width: int = 227*, *target_height: int = 227*) → typing.Callable

Load and prepare image the same way as Caffe scripts.

neuralmonkey.readers.numpy_reader module

neuralmonkey.readers.numpy_reader.**numpy_reader**(*files: typing.List[str]*)

neuralmonkey.readers.plain_text_reader module

neuralmonkey.readers.plain_text_reader.**UtfPlainTextReader**(*files: typing.List[str]*) → typing.Iterable[typing.List[str]]

neuralmonkey.readers.plain_text_reader.**get_plain_text_reader**(*encoding: str = 'utf-8'*)

Get reader for space-separated tokenized text.

neuralmonkey.readers.string_vector_reader module

neuralmonkey.readers.string_vector_reader.**FloatVectorReader**(*files: typing.List[str]*) → typing.Iterable[typing.List[numpy.ndarray]]

neuralmonkey.readers.string_vector_reader.**IntVectorReader**(*files: typing.List[str]*) → typing.Iterable[typing.List[numpy.ndarray]]

neuralmonkey.readers.string_vector_reader.**get_string_vector_reader**(*dtype: typing.Type = <class 'numpy.float32'>*, *columns: int = None*)

Get a reader for vectors encoded as whitespace-separated numbers

neuralmonkey.readers.utils module

Module contents

neuralmonkey.runners package

Submodules

neuralmonkey.runners.base_runner module

class neuralmonkey.runners.base_runner.**BaseRunner**(*output_series: str*, *decoder*) → None

Bases: object

decoder_data_id

get_executable(*compute_losses=False*, *summaries=True*) → neuralmonkey.runners.base_runner.Executable

loss_names

class neuralmonkey.runners.base_runner.**Executable**

Bases: object

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]

class neuralmonkey.runners.base_runner.**ExecutionResult**(*outputs*, *losses*, *scalar_summaries*, *histogram_summaries*, *image_summaries*)

Bases: tuple

histogram_summaries

Alias for field number 3

image_summaries

Alias for field number 4

losses

Alias for field number 1

outputs
Alias for field number 0

scalar_summaries
Alias for field number 2

neuralmonkey.runners.base_runner.**collect_encoders**(*coder*)
Collect recusively all encoders and decoders.

neuralmonkey.runners.base_runner.**reduce_execution_results**(*execution_results: typing.List[neuralmonkey.runners.base_runner.Exec* → neuralmonkey.runners.base_runner.ExecutionResult
Aggregate execution results into one.

neuralmonkey.runners.beamsearch_runner module

class neuralmonkey.runners.beamsearch_runner.**BeamSearchExecutable**(*rank: int, all_encoders: typing.List[neuralmonkey.model.model_pa bs_outputs: typing.List[neuralmonkey.decoders.beam_s vocabulary: neuralmonkey.vocabulary.Vocabulary, postprocess: typing.Union[typing.Callable, NoneType]*) → None
Bases: *neuralmonkey.runners.base_runner.Executable*

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]

class neuralmonkey.runners.beamsearch_runner.**BeamSearchRunner**(*output_series: str, decoder: neuralmonkey.decoders.beam_search_decoder.Beam rank: int = 1, postprocess: typing.Callable[[typing.List[str]], typing.List[str]] = None*) → None
Bases: *neuralmonkey.runners.base_runner.BaseRunner*

decoder_data_id

get_executable(*compute_losses: bool = False, summaries: bool = True*) → neuralmonkey.runners.beamsearch_runner.BeamSearchExecutable

loss_names

neuralmonkey.runners.beamsearch_runner.**beam_search_runner_range**(*output_series: str, decoder: neuralmonkey.decoders.beam_search_decoder max_rank: int = None, postprocess: typing.Callable[[typing.List[str]], typing.List[str]] = None*) → typing.List[neuralmonkey.runners.beamsea

A list of 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.label_runner module

class neuralmonkey.runners.label_runner.**LabelRunExecutable**(*all_coders, fetches, vocabulary, postprocess*)

Bases: *neuralmonkey.runners.base_runner.Executable*

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]

Get the feedables and tensors to run.

class neuralmonkey.runners.label_runner.**LabelRunner**(*output_series: str, decoder: typing.Any, postprocess: typing.Callable[[typing.List[str]], typing.List[str]] = None*) → None

Bases: *neuralmonkey.runners.base_runner.BaseRunner*

get_executable(*compute_losses=False, summaries=True*)

loss_names

neuralmonkey.runners.logits_runner module

A runner outputing logits or normalized distriution from a decoder.

class neuralmonkey.runners.logits_runner.**LogitsExecutable**(*all_coders: typing.List[neuralmonkey.model.model_part.ModelPar fetches: typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]], vocabulary: neuralmonkey.vocabulary.Vocabulary, normalize: bool = True, pick_index: int = None*) → None

Bases: *neuralmonkey.runners.base_runner.Executable*

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]

Get the feedables and tensors to run.

class neuralmonkey.runners.logits_runner.**LogitsRunner**(*output_series: str, decoder: typing.Any, 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.

get_executable(*compute_losses: bool = False, summaries: bool = True*) → neuralmonkey.runners.logits_runner.LogitsExecutable

loss_names

neuralmonkey.runners.perplexity_runner module

class neuralmonkey.runners.perplexity_runner.**PerplexityExecutable**(*all_coders: typing.List[neuralmonkey.model.model_pa xent_op: tensorflow.python.framework.ops.Tensor*) → None

Bases: *neuralmonkey.runners.base_runner.Executable*

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]

Get the feedables and tensors to run.

class neuralmonkey.runners.perplexity_runner.**PerplexityRunner**(*output_series: str, decoder: neuralmonkey.decoders.decoder.Decoder*) → None

Bases: *neuralmonkey.runners.base_runner.BaseRunner*

get_executable(*compute_losses=False*, *summaries=True*) → neuralmonkey.runners.perplexity_runner.PerplexityExecutable

loss_names

neuralmonkey.runners.plain_runner module

class neuralmonkey.runners.plain_runner.**PlainExecutable**(*all_coders*, *fetches*, *vocabulary*, *postprocess*) → None

Bases: *neuralmonkey.runners.base_runner.Executable*

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]

Get the feedables and tensors to run.

class neuralmonkey.runners.plain_runner.**PlainRunner**(*output_series: str*, *decoder: typing.Any*, *postprocess: typing.Callable[[typing.List[str]], typing.List[str]] = None*) → None

Bases: *neuralmonkey.runners.base_runner.BaseRunner*

A runner which takes the output from decoder.decoded.

get_executable(*compute_losses=False*, *summaries=True*)

loss_names

neuralmonkey.runners.regression_runner module

class neuralmonkey.runners.regression_runner.**RegressionRunExecutable**(*all_coders: typing.List[neuralmonkey.model.mode*, *fetches: typing.Dict[str, tensorflow.python.framework.ops.Tensor]*, *postprocess: typing.Callable[[float], float] = None*) → None

Bases: *neuralmonkey.runners.base_runner.Executable*

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]

Get the feedables and tensors to run.

class neuralmonkey.runners.regression_runner.**RegressionRunner**(*output_series: str, decoder: neuralmonkey.decoders.sequence_regressor.Sequenc postprocess: typing.Callable[[float], float] = None*) → None

Bases: *neuralmonkey.runners.base_runner.BaseRunner*

get_executable(*compute_losses: bool = False, summaries=True*) → neuralmonkey.runners.base_runner.Executable

loss_names

neuralmonkey.runners.representation_runner module

A runner that prints out the input representation from an encoder.

class neuralmonkey.runners.representation_runner.**RepresentationExecutable**(*prev_coders: typing.List[neuralmonkey.mode encoded: tensorflow.python.framework.ops. used_session: int*) → None

Bases: *neuralmonkey.runners.base_runner.Executable*

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]

class neuralmonkey.runners.representation_runner.**RepresentationRunner**(*output_series: str, encoder: neuralmonkey.model.model_part.Model used_session: int = 0*) → None

Bases: *neuralmonkey.runners.base_runner.BaseRunner*

Runner printing out representation from a encoder.

Using this runner is the way how to get input / other data representation out from Neural Monkey.

get_executable(*compute_losses=False, summaries=True*) → neuralmonkey.runners.representation_runner.RepresentationExecutable

loss_names

neuralmonkey.runners.rnn_runner module

Running of a recurrent decoder.

This module aggragates what is necessary to run efficiently a recurrent decoder. Unlike the default runner which assumes all outputs are independent on each other, this one does not make any of these assumptions. It implements model ensembling and beam search.

The TensorFlow session is invoked for every single output of the decoder separately which allows ensembling from all sessions and do the beam pruning before the a next output is emmited.

class neuralmonkey.runners.rnn_runner.**BeamBatch**(*decoded*, *logprobs*)
Bases: tuple

decoded
Alias for field number 0

logprobs
Alias for field number 1

class neuralmonkey.runners.rnn_runner.**ExpandedBeamBatch**(*beam_batch*, *next_logprobs*)
Bases: tuple

beam_batch
Alias for field number 0

next_logprobs
Alias for field number 1

class neuralmonkey.runners.rnn_runner.**RuntimeRnnExecutable**(*all_coders*, *decoder*, *initial_fetches*, *vocabulary*, *beam_scoring_f*, *postprocess*, *beam_size=1*, *compute_loss=True*)
Bases: *neuralmonkey.runners.base_runner.Executable*

Run and ensemble the RNN decoder step by step.

collect_results(*results: typing.List[typing.Dict]*) → None
Process what the TF session returned.

Only a single time step is always processed at once. First, distributions from all sessions are aggregated.

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]
Get the feedables and tensors to run.

It takes a beam batch that should be expanded the next and preprare an additional feed_dict based on the hypotheses history.

class neuralmonkey.runners.rnn_runner.**RuntimeRnnRunner**(*output_series: str*, *decoder*, *beam_size: int = 1*, *beam_scoring_f=<function likelihood_beam_score>*, *postprocess: typing.Callable[[typing.List[str]], typing.List[str]] = None*) → None
Bases: *neuralmonkey.runners.base_runner.BaseRunner*

Prepare running the RNN decoder step by step.

get_executable(*compute_losses=False*, *summaries=True*)

loss_names

neuralmonkey.runners.rnn_runner.**likelihood_beam_score**(*decoded*, *logprobs*)
Score the beam by normalized probaility.

neuralmonkey.runners.rnn_runner.**n_best**(*n: int*, *expanded: typing.List[neuralmonkey.runners.rnn_runner.ExpandedBeamBatch]*, *scoring_function*) → typing.List[neuralmonkey.runners.rnn_runner.BeamBatch]
Take n-best from expanded beam search hypotheses.

To do the scoring we need to "reshape" the hypohteses. Before the scoring the hypothesis are split into beam batches by their position in the beam. To do the scoring, however, they need to be organized by the instances. After the scoring, only _n_ hypotheses is kept for each isntance. These are again split by their position in the beam.

Parameters

- **n** – Beam size.
- **expanded** – List of batched expanded hypotheses.
- **scoring_function** – A function

Returns List of BeamBatches ready for new expansion.

neuralmonkey.runners.runner module

class neuralmonkey.runners.runner.**GreedyRunExecutable**(*all_coders*, *fetches*, *vocabulary*, *postprocess*) → None
Bases: *neuralmonkey.runners.base_runner.Executable*

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]
Get the feedables and tensors to run.

class neuralmonkey.runners.runner.**GreedyRunner**(*output_series: str*, *decoder: typing.Any*, *postprocess: typing.Callable[[typing.List[str]], typing.List[str]] = None*) → None
Bases: *neuralmonkey.runners.base_runner.BaseRunner*

get_executable(*compute_losses=False*, *summaries=True*)

loss_names

neuralmonkey.runners.word_alignment_runner module

class neuralmonkey.runners.word_alignment_runner.**WordAlignmentRunner**(*output_series: str*, *encoder: neuralmonkey.model.model_part.ModelPa*
decoder: neuralmonkey.decoders.decoder.Decoder) → None

Bases: *neuralmonkey.runners.base_runner.BaseRunner*

get_executable(*compute_losses=False*, *summaries=True*)

loss_names

class neuralmonkey.runners.word_alignment_runner.**WordAlignmentRunnerExecutable**(*all_coders*, *fetches*)

Bases: *neuralmonkey.runners.base_runner.Executable*

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]

Get the feedables and tensors to run.

Module contents

neuralmonkey.tests package

Submodules

neuralmonkey.tests.test_bleu module

class neuralmonkey.tests.test_bleu.**TestBLEU**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_bleu()

test_empty_decoded()

test_empty_reference()

test_empty_sentence()

test_identical()

neuralmonkey.tests.test_config module

Tests the config parsing module.

class neuralmonkey.tests.test_config.**TestParsing**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_splitter_bad_brackets()

neuralmonkey.tests.test_config.**test_splitter_gen**(*a*, *b*)

neuralmonkey.tests.test_dataset module

class neuralmonkey.tests.test_dataset.**TestDataset**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_nonexistent_file()

neuralmonkey.tests.test_decoder module

Unit tests for the decoder. (Tests only initialization so far)

class neuralmonkey.tests.test_decoder.**TestDecoder**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_init()

neuralmonkey.tests.test_encoders_init module

Test init methods of encoders.

class neuralmonkey.tests.test_encoders_init.**TestEncodersInit**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_post_cnn_encoder()

test_sentence_cnn_encoder()

test_sentence_encoder()

test_vector_encoder()

neuralmonkey.tests.test_encoders_init.**traverse_combinations**(*params: typing.Dict[str, typing.List[typing.Any]], partial_params: typing.Dict[str, typing.Any]*) → typing.Iterable[typing.Dict[str, typing.Any]]

neuralmonkey.tests.test_eval_wrappers module

class neuralmonkey.tests.test_eval_wrappers.**TestAccuracyEvaluator**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

test_seq_level_acc()

test_word_level_acc()

class neuralmonkey.tests.test_eval_wrappers.**TestExternalEvaluators**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_beer()

test_f1()

test_gleu()

test_multeval_bleu()

test_multeval_meteor()

test_multeval_ter()

neuralmonkey.tests.test_functions module

Unit tests for functions.py.

class neuralmonkey.tests.test_functions.**TestPiecewiseFunction**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_piecewise_constant()

neuralmonkey.tests.test_model_part module

Test ModelPart class.

class neuralmonkey.tests.test_model_part.**Test**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Test capabilities of model part.

test_save_and_load()
Try to save and load encoder.

neuralmonkey.tests.test_nn_utils module

class neuralmonkey.tests.test_nn_utils.**TestDropout**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_invalid_keep_prob()
Tests invalid dropout values

test_keep_prob()
Counts dropped items and compare with the expectation

test_train_false()
Checks that dropout is not used when not training

neuralmonkey.tests.test_readers module

Unit tests for readers

class neuralmonkey.tests.test_readers.**TestStringVectorReader**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

tearDown()

test_columns()

test_reader()

neuralmonkey.tests.test_ter module

class neuralmonkey.tests.test_ter.**TestBLEU**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_empty_decoded()

test_empty_reference()

test_empty_sentence()

test_identical()

test_ter()

neuralmonkey.tests.test_vocabulary module

class neuralmonkey.tests.test_vocabulary.**TestVocabulary**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_all_words_in()

test_min_freq()

test_padding()

test_there_and_back_self()

test_unknown_word()

test_weights()

Module contents

neuralmonkey.trainers package

Submodules

neuralmonkey.trainers.cross_entropy_trainer module

class neuralmonkey.trainers.cross_entropy_trainer.**CrossEntropyTrainer**(*decoders: typing.List[typing.Any], decoder_weights: typing.Union[typing.List[typing.Union[float, NoneType]], NoneType] = None, l1_weight=0.0, l2_weight=0.0, clip_norm=False, optimizer=None, global_step=None*) → None

Bases: *neuralmonkey.trainers.generic_trainer.GenericTrainer*

neuralmonkey.trainers.cross_entropy_trainer.**xent_objective**(*decoder, weight=None*) → neuralmonkey.trainers.generic_trainer.Objective

Get XENT objective from decoder with cost.

neuralmonkey.trainers.generic_trainer module

class neuralmonkey.trainers.generic_trainer.**GenericTrainer**(*objectives: typing.List[neuralmonkey.trainers.generic_trainer.Ob l1_weight: float = 0.0, l2_weight: float = 0.0, clip_norm: typing.Union[float, NoneType] = None, optimizer=None, global_step=None*) → None

Bases: object

get_executable(*compute_losses=True, summaries=True*) → neuralmonkey.runners.base_runner.Executable

class neuralmonkey.trainers.generic_trainer.**Objective**(*name, decoder, loss, gradients, weight*)

Bases: tuple

decoder
Alias for field number 1

gradients
Alias for field number 3

loss
Alias for field number 2

name
Alias for field number 0

weight
Alias for field number 4

class neuralmonkey.trainers.generic_trainer.**TrainExecutable**(*all_coders*, *train_op*, *losses*, *scalar_summaries*, *histogram_summaries*)

Bases: *neuralmonkey.runners.base_runner.Executable*

collect_results(*results: typing.List[typing.Dict]*) → None

next_to_execute() → typing.Tuple[typing.List[typing.Any], typing.Union[typing.Dict, typing.List], typing.Dict[tensorflow.python.framework.ops.Tensor, typing.Union[int, float, numpy.ndarray]]]

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.Dec reward_function: typing.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).

Module contents

Submodules

neuralmonkey.checking module

This module servers 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: typing.List[typing.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: typing.Iterable[neuralmonkey.runners.base_runner.BaseRunner]*) → None

neuralmonkey.dataset module

Implementation of the dataset class.

class neuralmonkey.dataset.**Dataset**(*name: str, series: typing.Dict[str, typing.List], series_outputs: typing.Dict[str, str]*) → None
Bases: collections.abc.Sized

This class serves as collection for data series for particular encoders and decoders in the model. If it is not provided a parent dataset, it also manages the vocabularies inferred from the data.

A data series is either a list of strings or a numpy array.

add_series(*name: str, series: typing.List[typing.Any]*) → None

batch_dataset(*batch_size: int*) → typing.Iterable[typing.Dataset]
Split the dataset into a list of batched datasets.

Parameters **batch_size** – The size of a batch.

Returns Generator yielding batched datasets.

batch_serie(*serie_name: str*, *batch_size: int*) → typing.Iterable[typing.Iterable]
Split a data serie into batches.

Parameters

- **serie_name** – The name of the series
- **batch_size** – The size of a batch

Returns Generator yielding batches of the data from the serie.

get_series(*name: str*, *allow_none: bool = False*) → typing.Iterable
Get the data series with a given name.

Parameters

- **name** – The name of the series to fetch.
- **allow_none** – If True, return None if the series does not exist.

Returns The data series.

Raises KeyError if the series does not exists and allow_none is False

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.

series_ids

shuffle() → None
Shuffle the dataset randomly

subset(*start: int*, *length: int*) → neuralmonkey.dataset.Dataset

class neuralmonkey.dataset.**LazyDataset**(*name: str, series_paths_and_readers: typing.Dict[str, typing.Tuple[typing.List[str], typing.Callable[[typing.List[str]], typing.Any]]], series_outputs: typing.Dict[str, str], preprocessors: typing.List[typing.Tuple[str, str, typing.Callable]] = None*) → None
Bases: *neuralmonkey.dataset.Dataset*

Implements the lazy dataset.

The main difference between this implementation and the default one is that the contents of the file are not fully loaded to the memory. Instead, everytime the function `get_series` is called, a new file handle is created and a generator which yields lines from the file is returned.

add_series(*name: str, series: typing.Iterable[typing.Any]*) → None

get_series(*name: str*, *allow_none: bool = False*) → typing.Iterable
Get the data series with a given name.

This function opens a new file handle and returns a generator which yields preprocessed lines from the file.

Parameters

- **name** – The name of the series to fetch.
- **allow_none** – If True, return None if the series does not exist.

Returns The data series.

Raises KeyError if the series does not exists and allow_none is False

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.

series_ids

shuffle() → None
Does nothing, not in-memory shuffle is impossible.

TODO: this is related to the `__len__` method.

subset (*start: int*, *length: int*) → neuralmonkey.dataset.Dataset

neuralmonkey.dataset.**load_dataset_from_files** (*name: str = None*, *lazy: bool = False*, *preprocessors: typing.List[typing.Tuple[str, str, typing.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.

Raises Exception when no input files are provided.

neuralmonkey.decoding_function module

Module which implements decoding functions using multiple attentions for RNN decoders.

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

class neuralmonkey.decoding_function.**Attention** (*attention_states: tensorflow.python.framework.ops.Tensor*, *scope: str*, *attention_state_size: int = None*, *input_weights: tensorflow.python.framework.ops.Tensor = None*, *attention_fertility: int = None*) → None

Bases: *neuralmonkey.decoding_function.BaseAttention*

attention (*decoder_state: tensorflow.python.framework.ops.Tensor*, *decoder_prev_state: tensorflow.python.framework.ops.Tensor*, *_*) → tensorflow.python.framework.ops.Tensor

put attention masks on att_states_reshaped using hidden_features and query.

get_logits (*y*)

class neuralmonkey.decoding_function.**BaseAttention** (*scope: str*, *attention_states: tensorflow.python.framework.ops.Tensor*, *attention_state_size: int*, *input_weights: tensorflow.python.framework.ops.Tensor = None*) → None

Bases: object

attention(*decoder_state: tensorflow.python.framework.ops.Tensor, decoder_prev_state: tensorflow.python.framework.ops.Tensor, decoder_input: tensorflow.python.framework.ops.Tensor*) → tensorflow.python.framework.ops.Tensor

Get context vector for given decoder state.

class neuralmonkey.decoding_function.**CoverageAttention**(*attention_states: tensorflow.python.framework.ops.Tensor, scope: str, input_weights: tensorflow.python.framework.ops.Tensor = None, attention_fertility: int = 5*) → None

Bases: *neuralmonkey.decoding_function.Attention*

get_logits(*y*)

class neuralmonkey.decoding_function.**RecurrentAttention**(*scope: str, attention_states: tensorflow.python.framework.ops.Tensor, input_weights: tensorflow.python.framework.ops.Tensor, attention_state_size: int, **kwargs*) → None

Bases: *neuralmonkey.decoding_function.BaseAttention*

From article **'Recurrent Neural Machine Translation '<https://arxiv.org/pdf/1607.08725v1.pdf>'_**

In time i of the decoder with state s_i-1, and encoder states h_j, we run a bidirectional RNN with initial state set to

c_0 = tanh(V*s_i-1 + b_0)

Then we run the GRU net (in paper just forward, we do bidi) and we get N+1 hidden states c_0 ... c_N

to compute the context vector, they try either last state or mean of all the states. Last state was better in their experiments so that's what we're gonna use.

attention(*decoder_state: tensorflow.python.framework.ops.Tensor, decoder_prev_state: tensorflow.python.framework.ops.Tensor, _*) → tensorflow.python.framework.ops.Tensor

neuralmonkey.decorators module

neuralmonkey.decorators.**tensor**(*func*)

neuralmonkey.functions module

neuralmonkey.functions.**inverse_sigmoid_decay**(*param, rate, min_value: float = 0.0, max_value: float = 1.0, name: typing.Union[str, NoneType] = None, dtype=tf.float32*) → tensorflow.python.framework.ops.Tensor

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.**piecewise_function**(*param*, *values*, *changepoints*, *name=None*, *dtype=tf.float32*)

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*, *dataset*, *runners*, *execution_results*, *result_data*)

Evaluate the model outputs.

Parameters

- **evaluators** – List of tuples of series and evaluation functions.
- **dataset** – Dataset 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: typing.Dict[str, float]*) → None

Print final evaluation from a test dataset.

neuralmonkey.learning_utils.**run_on_dataset**(*tf_manager: neuralmonkey.tf_manager.TensorFlowManager, runners: typing.List[neuralmonkey.runners.base_runner.BaseRunner], dataset: neuralmonkey.dataset.Dataset, postprocess: typing.Union[typing.List[typing.Tuple[str, typing.Callable]], NoneType], write_out: bool = False, batch_size: typing.Union[int, NoneType] = None*) → typing.Tuple[typing.List[neuralmonkey.runners.base_runner.ExecutionRes typing.Dict[str, typing.List[typing.Any]]]

Apply the model on a dataset and optionally write outputs to files.

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** – an object to use as postprocessing of the
- **write_out** – Flag whether the outputs should be printed to a file defined in the dataset object.
- **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, trainer: neuralmonkey.trainers.generic_trainer.GenericTrainer, batch_size: int, log_directory: str, evaluators: typing.List[typing.Union[typing.Tuple[str, typing.Any], typing.Tuple[str, str, typing.Any]]], runners: typing.List[neuralmonkey.runners.base_runner.BaseRunner], train_dataset: neuralmonkey.dataset.Dataset, val_dataset: typing.Union[neuralmonkey.dataset.Dataset, typing.List[neuralmonkey.dataset.Dataset]], test_datasets: typing.Union[typing.List[neuralmonkey.dataset.Dataset], NoneType] = None, logging_period: typing.Union[str, int] = 20, validation_period: typing.Union[str, int] = 500, val_preview_input_series: typing.Union[typing.List[str], NoneType] = None, val_preview_output_series: typing.Union[typing.List[str], NoneType] = None, val_preview_num_examples: int = 15, train_start_offset: int = 0, runners_batch_size: typing.Union[int, NoneType] = None, initial_variables: typing.Union[str, typing.List[str], NoneType] = None, postprocess: typing.Union[typing.List[typing.Tuple[str, typing.Callable]], NoneType] = None*) → None

Performs the training loop for given graph and data. :param tf_manager: TensorFlowManager with initialized sessions. :param epochs: Number of epochs for which the algoritm will learn. :param trainer: The trainer object containg the TensorFlow code for computing

the loss and optimization operation.

Parameters

- **batch_size** – number of examples in one mini-batch
- **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.
- **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. It is the same as batch_size if not specified
- **initial_variables** – variables used for initialization, for example for continuation of training
- **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(*message: str*, *label: typing.Union[str, NoneType] = None*)

debug_disabled = ['']

debug_enabled = ['none']

static log(*message: str*, *color: str = 'yellow'*) → None
Logs message with a colored timestamp.

log_file = None

static log_print(*text: str*) → None
Prints a string both to console and a log file is it is defined.

static notice(*message: str*) → None
Logs notice with a colored timestamp.

static print_header(*title: str*, *path: str*) → None
Prints the title of the experiment and the set of arguments it uses.

static set_log_file(*path: str*) → None
Sets up the file where the logging will be done.

strict_mode = None

static **warn**(*message: str*) → None
Logs a warning.

neuralmonkey.logging.**debug**(*message: str*, *label: typing.Union[str, NoneType] = None*)

neuralmonkey.logging.**log**(*message: str*, *color: str = 'yellow'*) → None
Logs message with a colored timestamp.

neuralmonkey.logging.**log_print**(*text: str*) → None
Prints a string both to console and a log file is it is defined.

neuralmonkey.logging.**notice**(*message: str*) → None
Logs notice with a colored timestamp.

neuralmonkey.logging.**warn**(*message: str*) → None
Logs a warning.

neuralmonkey.run module

neuralmonkey.run.**default_variable_file**(*output_dir*)

neuralmonkey.run.**initialize_for_running**(*output_dir*, *tf_manager*, *variable_files*) → None
Restore either default variables of from configuration.

Parameters

- **output_dir** – Training output directory.
- **tf_manager** – TensorFlow manager.
- **variable_files** – Files with variables to be restored or None if the default variables should be used.

neuralmonkey.run.**main**() → None

neuralmonkey.server module

neuralmonkey.server.**main**() → None

neuralmonkey.server.**post_request**()

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, variable_files: typing.Union[typing.List[str], NoneType] = None, gpu_allow_growth: bool = True, per_process_gpu_memory_fraction: float = 1.0, report_gpu_memory_consumption: bool = False, enable_tf_debug: bool = False*) → None

Bases: object

Inteface between computational graph, data and TF sessions.

sessions
List of active Tensorflow sessions.

execute(*dataset: neuralmonkey.dataset.Dataset*, *execution_scripts*, *train=False*, *compute_losses=True*, *summaries=True*, *batch_size=None*) → typing.List[neuralmonkey.runners.base_runner.ExecutionResult]

init_saving(*vars_prefix: str*) → None

initialize_model_parts(*runners*, *save=False*) → None
Initialize model parts variables from their checkpoints.

restore(*variable_files: typing.Union[str, typing.List[str]]*) → None

restore_best_vars() → None

save(*variable_files: typing.Union[str, typing.List[str]]*) → None

validation_hook(*score: float*, *epoch: int*, *batch: int*) → None

neuralmonkey.tf_utils module

Small helper functions for TensorFlow.

neuralmonkey.tf_utils.**gpu_memusage**() → str
Return '' or a string showing current GPU memory usage.

nvidia-smi result parsing based on https://github.com/wookayin/gpustat

neuralmonkey.tf_utils.**has_gpu**() → bool
Check if TensorFlow can access GPU.

The test is based on https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/platform/test.py

...but we are interested only in CUDA GPU devices.

Returns True, if TF can access the GPU

neuralmonkey.train module

This is a training script for sequence to sequence learning.

neuralmonkey.train.**create_config**() → neuralmonkey.config.configuration.Configuration

neuralmonkey.train.**main**() → None

neuralmonkey.vocabulary 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: typing.List[str] = None*, *unk_sample_prob: float = 0.0*) → None
Bases: collections.abc.Sized

add_tokenized_text(*tokenized_text: typing.List[str]*) → None
Add words from a list to the vocabulary.

Parameters **tokenized_text** – The list of words to add.

add_word(*word: str*) → None
Add a word to the vocablulary.

Parameters **word** – The word to add. If it's already there, increment the count.

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*)
Logs a sample of the vocabulary

Parameters **size** – How many sample words to log.

save_to_file(*path: str*, *overwrite: bool = False*) → None
Save the vocabulary to a file.

Parameters

- **path** – The path to save the file to.
- **overwrite** – Flag whether to overwrite existing file. Defaults to False.

Raises

- FileExistsError if the file exists and overwrite flag is
- disabled.

sentences_to_tensor(*sentences: typing.List[typing.List[str]], max_len: typing.Union[int, NoneType] = None, pad_to_max_len: bool = True, train_mode: bool = False, add_start_symbol: bool = False, add_end_symbol: bool = False*) → typing.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 by discarding infrequent tokens.

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: typing.List[numpy.ndarray]*) → typing.List[typing.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_bpe**(*path: str, encoding: str = 'utf-8'*) → neuralmonkey.vocabulary.Vocabulary
Loads vocabulary from Byte-pair encoding merge list.

NOTE: The frequencies of words in this vocabulary are not computed from data. Instead, they correspond to the number of times the subword units occurred in the BPE merge list. This means that smaller words will tend to have larger frequencies assigned and therefore the truncation of the vocabulary can be somehow performed (but not without a great deal of thought).

Parameters

- **path** – File name to load the vocabulary from.
- **encoding** – The encoding of the merge file (defaults to UTF-8)

neuralmonkey.vocabulary.**from_dataset**(*datasets: typing.List[neuralmonkey.dataset.Dataset], series_ids: typing.List[str], max_size: int, save_file: str = None, overwrite: bool = False, min_freq: typing.Union[int, NoneType] = None, unk_sample_prob: float = 0.5*) → neuralmonkey.vocabulary.Vocabulary
Loads 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**(*path: str*) → neuralmonkey.vocabulary.Vocabulary

Loads vocabulary from a pickled file

Parameters **path** – The path to the pickle file

Returns The newly created vocabulary.

neuralmonkey.vocabulary.**from_wordlist**(*path: str*, *encoding: str = 'utf-8'*) → neuralmonkey.vocabulary.Vocabulary

Loads vocabulary from a wordlist.

Parameters

- **path** – The path to the wordlist file
- **encoding** – The encoding of the merge file (defaults to UTF-8)

Returns The new Vocabulary instance.

neuralmonkey.vocabulary.**initialize_vocabulary**(*directory: str*, *name: str*, *datasets: typing.List[neuralmonkey.dataset.Dataset] = None*, *series_ids: typing.List[str] = None*, *max_size: int = None*) → neuralmonkey.vocabulary.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

Module contents

The neuralmonkey package is the root package of this project.

Visualization

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.

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>
```

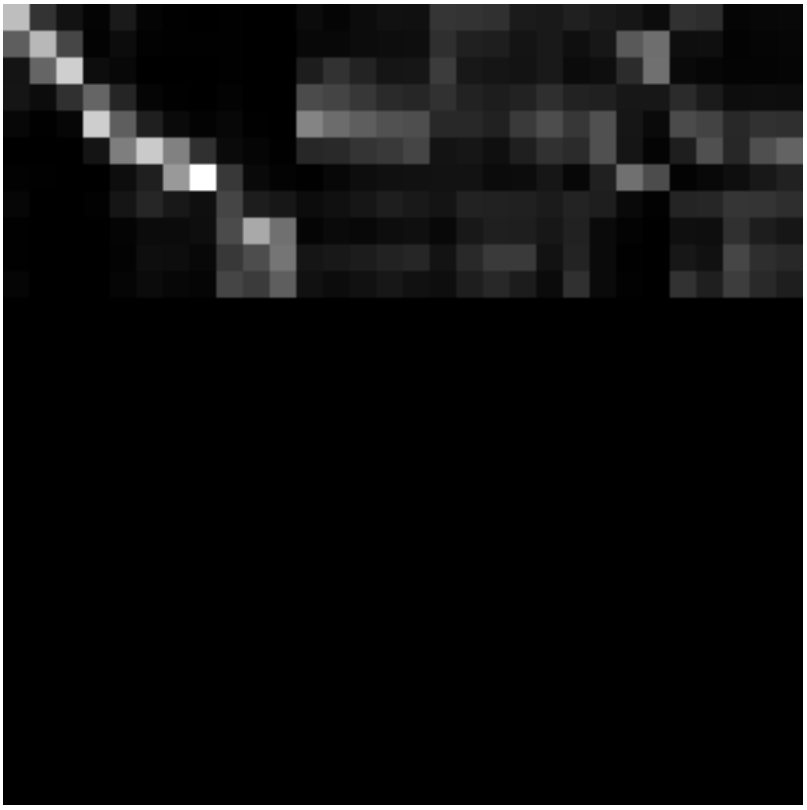
And then you can navigate in your browser to *http://localhost:6006/* (or if the TensorBoard assigns different port) and view all the summaries about your experiment.

How to read TensorBoard

The *step* in the TensorBoard is describing how many inputs (not batches) was processed.

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
```

Advanced Features

Byte Pair Encoding

This is explained in *the machine translation tutorial*.

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.sentence_encoder.SentenceEncoder
dropout_keep_prob=0.8
...
```

or:

```
[decoder]
class=decoders.decoder.Decoder
dropout_keep_prob=0.8
...
```

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.sentence_encoder.SentenceEncoder
dropout_keep_prob=0.8
use_pervasive_dropout=True
...
```

Attention Seeded by GIZA++ Word Alignments

todo: OC to reference the paper and describe how to use this in NM

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!)

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

Internal Development 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 on Github.

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.

Python Module Index

n

neuralmonkey, 78
neuralmonkey.checking, 67
neuralmonkey.config, 22
neuralmonkey.config.builder, 20
neuralmonkey.config.configuration, 21
neuralmonkey.config.exceptions, 21
neuralmonkey.config.parsing, 22
neuralmonkey.config.utils, 22
neuralmonkey.dataset, 67
neuralmonkey.decoders, 31
neuralmonkey.decoders.beam_search_decoder, 23
neuralmonkey.decoders.ctc_decoder, 24
neuralmonkey.decoders.decoder, 25
neuralmonkey.decoders.encoder_projection, 25
neuralmonkey.decoders.output_projection, 27
neuralmonkey.decoders.sequence_classifier, 28
neuralmonkey.decoders.sequence_labeler, 29
neuralmonkey.decoders.sequence_regressor, 30
neuralmonkey.decoders.word_alignment_decoder, 31
neuralmonkey.decoding_function, 69
neuralmonkey.decorators, 70
neuralmonkey.encoders, 41
neuralmonkey.encoders.attentive, 31
neuralmonkey.encoders.cnn_encoder, 31
neuralmonkey.encoders.encoder_wrapper, 32
neuralmonkey.encoders.factored_encoder, 34
neuralmonkey.encoders.imagenet_encoder, 34
neuralmonkey.encoders.numpy_encoder, 35
neuralmonkey.encoders.raw_rnn_encoder, 35
neuralmonkey.encoders.sentence_cnn_encoder, 36
neuralmonkey.encoders.sentence_encoder, 39
neuralmonkey.encoders.sequence_cnn_encoder, 40
neuralmonkey.evaluators, 45
neuralmonkey.evaluators.accuracy, 41
neuralmonkey.evaluators.average, 41
neuralmonkey.evaluators.beer, 41
neuralmonkey.evaluators.bleu, 42
neuralmonkey.evaluators.bleu_ref, 43
neuralmonkey.evaluators.edit_distance, 43
neuralmonkey.evaluators.f1_bio, 43
neuralmonkey.evaluators.gleu, 43
neuralmonkey.evaluators.mse, 44
neuralmonkey.evaluators.multeval, 44
neuralmonkey.evaluators.ter, 45
neuralmonkey.evaluators.wer, 45
neuralmonkey.functions, 70
neuralmonkey.learning_utils, 71
neuralmonkey.logging, 73
neuralmonkey.model, 45
neuralmonkey.model.model_part, 45
neuralmonkey.nn, 49
neuralmonkey.nn.highway, 46
neuralmonkey.nn.mlp, 46
neuralmonkey.nn.noisy_gru_cell, 46
neuralmonkey.nn.ortho_gru_cell, 47
neuralmonkey.nn.pervasive_dropout_wrapper, 47
neuralmonkey.nn.projection, 47
neuralmonkey.nn.utils, 48
neuralmonkey.processors, 50
neuralmonkey.processors.alignment, 49
neuralmonkey.processors.bpe, 49
neuralmonkey.processors.editops, 50

neuralmonkey.processors.german, 50
neuralmonkey.processors.helpers, 50
neuralmonkey.readers, 52
neuralmonkey.readers.image_reader, 51
neuralmonkey.readers.numpy_reader, 51
neuralmonkey.readers.plain_text_reader, 51
neuralmonkey.readers.string_vector_reader, 51
neuralmonkey.readers.utils, 52
neuralmonkey.run, 74
neuralmonkey.runners, 60
neuralmonkey.runners.base_runner, 52
neuralmonkey.runners.beamsearch_runner, 53
neuralmonkey.runners.label_runner, 54
neuralmonkey.runners.logits_runner, 54
neuralmonkey.runners.perplexity_runner, 55
neuralmonkey.runners.plain_runner, 56
neuralmonkey.runners.regression_runner, 56
neuralmonkey.runners.representation_runner, 57
neuralmonkey.runners.rnn_runner, 58
neuralmonkey.runners.runner, 59
neuralmonkey.runners.word_alignment_runner, 60
neuralmonkey.server, 74
neuralmonkey.tests, 63
neuralmonkey.tests.test_bleu, 60
neuralmonkey.tests.test_config, 60
neuralmonkey.tests.test_dataset, 61
neuralmonkey.tests.test_decoder, 61
neuralmonkey.tests.test_encoders_init, 61
neuralmonkey.tests.test_eval_wrappers, 61
neuralmonkey.tests.test_functions, 62
neuralmonkey.tests.test_model_part, 62
neuralmonkey.tests.test_nn_utils, 62
neuralmonkey.tests.test_readers, 62
neuralmonkey.tests.test_ter, 63
neuralmonkey.tests.test_vocabulary, 63
neuralmonkey.tf_manager, 74
neuralmonkey.tf_utils, 75
neuralmonkey.train, 75
neuralmonkey.trainers, 66
neuralmonkey.trainers.cross_entropy_trainer, 64
neuralmonkey.trainers.generic_trainer, 64
neuralmonkey.trainers.self_critical_objective, 65
neuralmonkey.vocabulary, 75

Index

A

AccuracyEvaluator (class in neuralmonkey.evaluators.accuracy), 41
AccuracySeqLevelEvaluator (class in neuralmonkey.evaluators.accuracy), 41
adadelta_optimizer() (in module neuralmonkey.config.utils), 22
adam_optimizer() (in module neuralmonkey.config.utils), 22
add_argument() (neuralmonkey.config.configuration.Configuration method), 21
add_series() (neuralmonkey.dataset.Dataset method), 67
add_series() (neuralmonkey.dataset.LazyDataset method), 68
add_tokenized_text() (neuralmonkey.vocabulary.Vocabulary method), 75
add_word() (neuralmonkey.vocabulary.Vocabulary method), 75
assert_same_shape() (in module neuralmonkey.checking), 67
assert_shape() (in module neuralmonkey.checking), 67
Attention (class in neuralmonkey.decoding_function), 69
attention() (neuralmonkey.decoding_function.Attention method), 69
attention() (neuralmonkey.decoding_function.BaseAttention method), 69
attention() (neuralmonkey.decoding_function.RecurrentAttention method), 70
attention() (neuralmonkey.encoders.encoder_wrapper.FlatMultiAttention method), 33
attention() (neuralmonkey.encoders.encoder_wrapper.HierarchicalMultiAttention method), 33
attention() (neuralmonkey.encoders.encoder_wrapper.MultiAttention method), 33
Attentive (class in neuralmonkey.encoders.attentive), 31
attn_size (neuralmonkey.encoders.encoder_wrapper.MultiAttention attribute), 34
AverageEvaluator (class in neuralmonkey.evaluators.average), 41

B

BaseAttention (class in neuralmonkey.decoding_function), 69
BaseRunner (class in neuralmonkey.runners.base_runner), 52
batch_dataset() (neuralmonkey.dataset.Dataset method), 67
batch_serie() (neuralmonkey.dataset.Dataset method), 67
beam_batch (neuralmonkey.runners.rnn_runner.ExpandedBeamBatch attribute), 58
beam_search_runner_range() (in module neuralmonkey.runners.beamsearch_runner), 53
beam_size (neuralmonkey.decoders.beam_search_decoder.BeamSearchDec attribute), 23
BeamBatch (class in neuralmonkey.runners.rnn_runner), 58
BeamSearchDecoder (class in neuralmonkey.decoders.beam_search_decoder), 23
BeamSearchExecutable (class in neuralmonkey.runners.beamsearch_runner), 53
BeamSearchRunner (class in neuralmonkey.runners.beamsearch_runner), 53
BeerWrapper (class in neuralmonkey.evaluators.beer), 41
bidirectional_rnn (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 37
bleu() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 42
BLEUEvaluator (class in neuralmonkey.evaluators.bleu), 42
BLEUReferenceImplWrapper (class in neuralmonkey.evaluators.bleu_ref), 43
BPEPostprocessor (class in neuralmonkey.processors.bpe), 49
BPEPreprocessor (class in neuralmonkey.processors.bpe), 49

build_config() (in module neuralmonkey.config.builder), 20
build_model() (neuralmonkey.config.configuration.Configuration method), 21
build_object() (in module neuralmonkey.config.builder), 21

C

cell_type (neuralmonkey.encoders.raw_rnn_encoder.RNNSpec attribute), 35
check_dataset_and_coders() (in module neuralmonkey.checking), 67
CheckingException, 67
chunk2set() (neuralmonkey.evaluators.f1_bio.F1Evaluator static method), 43
classification (neuralmonkey.nn.mlp.MultilayerPerceptron attribute), 46
ClassSymbol (class in neuralmonkey.config.builder), 20
cnn_encoded (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 37
cnn_states (neuralmonkey.encoders.imagenet_encoder.ImageNet attribute), 35
CNNEncoder (class in neuralmonkey.encoders.cnn_encoder), 31
collect_encoders() (in module neuralmonkey.runners.base_runner), 53
collect_results() (neuralmonkey.runners.base_runner.Executable method), 52
collect_results() (neuralmonkey.runners.beamsearch_runner.BeamSearchExecutable method), 53
collect_results() (neuralmonkey.runners.label_runner.LabelRunExecutable method), 54
collect_results() (neuralmonkey.runners.logits_runner.LogitsExecutable method), 55
collect_results() (neuralmonkey.runners.perplexity_runner.PerplexityExecutable method), 55
collect_results() (neuralmonkey.runners.plain_runner.PlainExecutable method), 56
collect_results() (neuralmonkey.runners.regression_runner.RegressionRunExecutable method), 56
collect_results() (neuralmonkey.runners.representation_runner.RepresentationExecutable method), 57
collect_results() (neuralmonkey.runners.rnn_runner.RuntimeRnnExecutable method), 58
collect_results() (neuralmonkey.runners.runner.GreedyRunExecutable method), 59
collect_results() (neuralmonkey.runners.word_alignment_runner.WordAlignmentRunnerE method), 60
collect_results() (neuralmonkey.trainers.generic_trainer.TrainExecutable method), 65
compare_scores() (neuralmonkey.evaluators.accuracy.AccuracyEvaluator static method), 41
compare_scores() (neuralmonkey.evaluators.accuracy.AccuracySeqLevelEvaluator static method), 41
compare_scores() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 42
compare_scores() (neuralmonkey.evaluators.edit_distance.EditDistanceEvaluator static method), 43
compare_scores() (neuralmonkey.evaluators.mse.MeanSquaredErrorEvaluator static method), 44
concat_encoder_projection() (in module neuralmonkey.decoders.encoder_projection), 25
ConfigBuildException, 21
ConfigInvalidValueException, 21
Configuration (class in neuralmonkey.config.configuration), 21
convert_to_edits() (in module neuralmonkey.processors.editops), 50
cost (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 24
cost (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 28
cost (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 29
cost (neuralmonkey.decoders.sequence_regressor.SequenceRegressor attribute), 30
cost (neuralmonkey.decoders.word_alignment_decoder.WordAlignmentDec attribute), 31
CoverageAttention (class in neuralmonkey.decoding_function), 70
create() (neuralmonkey.config.builder.ClassSymbol method), 20
create_attention_object() (neuralmonkey.encoders.attentive.Attentive method), 34
create_attention_object() (neuralmonkey.encoders.encoder_wrapper.EncoderWrapper method), 33
create_config() (in module neuralmonkey.train), 75
CrossEntropyTrainer (class in neural-

monkey.trainers.cross_entropy_trainer), 64
CTCDecoder (class in neuralmonkey.decoders.ctc_decoder), 24

D

Dataset (class in neuralmonkey.dataset), 67
dataset_from_files() (in module neuralmonkey.config.utils), 22
debug() (in module neuralmonkey.logging), 74
debug() (neuralmonkey.logging.Logging static method), 73
debug_disabled (neuralmonkey.logging.Logging attribute), 73
debug_enabled (neuralmonkey.logging.Logging attribute), 73
decode() (neuralmonkey.processors.bpe.BPEPostprocessor method), 49
decode() (neuralmonkey.processors.german.GermanPostprocessor method), 50
decoded (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 24
decoded (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 28
decoded (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 29
decoded (neuralmonkey.decoders.sequence_regressor.SequenceRegressor attribute), 30
decoded (neuralmonkey.runners.rnn_runner.BeamBatch attribute), 58
decoded_logits (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 28
decoded_seq (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 28
Decoder (class in neuralmonkey.decoders.decoder), 25
decoder (neuralmonkey.trainers.generic_trainer.Objective attribute), 64
decoder_data_id (neuralmonkey.runners.base_runner.BaseRunner attribute), 52
decoder_data_id (neuralmonkey.runners.beamsearch_runner.BeamSearchRunner attribute), 53
deduplicate_sentences() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 42
default_variable_file() (in module neuralmonkey.run), 74
deprecated() (in module neuralmonkey.config.utils), 22
direction (neuralmonkey.encoders.raw_rnn_encoder.RNNSpec attribute), 35
dropout() (in module neuralmonkey.nn.utils), 48

E

EditDistanceEvaluator (class in neuralmonkey.evaluators.edit_distance), 43
effective_reference_length() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 42
embed_and_dropout() (neuralmonkey.decoders.decoder.Decoder method), 25
embedded_inputs (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 38
embedded_inputs (neuralmonkey.encoders.sequence_cnn_encoder.SequenceCNNEncoder attribute), 40
empty_initial_state() (in module neuralmonkey.decoders.encoder_projection), 26
encoded (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 32
encoded (neuralmonkey.encoders.imagenet_encoder.ImageNet attribute), 35
encoded (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEnco attribute), 38
encoded (neuralmonkey.encoders.sequence_cnn_encoder.SequenceCNNEnc attribute), 40
EncoderWrapper (class in neuralmonkey.encoders.encoder_wrapper), 32
evaluation() (in module neuralmonkey.learning_utils), 71
Executable (class in neuralmonkey.runners.base_runner), 52
execute() (neuralmonkey.tf_manager.TensorFlowManager method), 75
ExecutionResult (class in neuralmonkey.runners.base_runner), 52
ExpandedBeamBatch (class in neuralmonkey.runners.rnn_runner), 58

F

f1_score() (neuralmonkey.evaluators.f1_bio.F1Evaluator static method), 43
F1Evaluator (class in neuralmonkey.evaluators.f1_bio), 43
FactoredEncoder (class in neuralmonkey.encoders.factored_encoder), 34
feed_dict() (neuralmonkey.decoders.beam_search_decoder.BeamSearchDec method), 23
feed_dict() (neuralmonkey.decoders.ctc_decoder.CTCDecoder method), 24
feed_dict() (neuralmonkey.decoders.decoder.Decoder method), 25
feed_dict() (neuralmonkey.decoders.sequence_classifier.SequenceClassifier method), 28
feed_dict() (neuralmonkey.decoders.sequence_labeler.SequenceLabeler method), 29
feed_dict() (neuralmonkey.decoders.sequence_regressor.SequenceRegressor method), 30

feed_dict() (neuralmonkey.decoders.word_alignment_decoder.WordAlignmentDecoder method), 31
feed_dict() (neuralmonkey.encoders.cnn_encoder.CNNEncoder method), 32
feed_dict() (neuralmonkey.encoders.encoder_wrapper.EncoderWrapper method), 33
feed_dict() (neuralmonkey.encoders.factored_encoder.FactoredEncoder method), 34
feed_dict() (neuralmonkey.encoders.imagenet_encoder.ImageNet method), 35
feed_dict() (neuralmonkey.encoders.numpy_encoder.PostCNNImageEncoder method), 35
feed_dict() (neuralmonkey.encoders.numpy_encoder.VectorEncoder method), 35
feed_dict() (neuralmonkey.encoders.raw_rnn_encoder.RawRNNEncoder method), 36
feed_dict() (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder method), 38
feed_dict() (neuralmonkey.encoders.sentence_encoder.SentenceEncoder method), 39
feed_dict() (neuralmonkey.encoders.sequence_cnn_encoder.SequenceCNNEncoder method), 40
feed_dict() (neuralmonkey.model.model_part.ModelPart method), 45
finished (neuralmonkey.decoders.beam_search_decoder.SearchState attribute), 23
FlatMultiAttention (class in neuralmonkey.encoders.encoder_wrapper), 33
FloatVectorReader() (in module neuralmonkey.readers.string_vector_reader), 51
from_bpe() (in module neuralmonkey.vocabulary), 77
from_dataset() (in module neuralmonkey.vocabulary), 77
from_file() (in module neuralmonkey.vocabulary), 78
from_wordlist() (in module neuralmonkey.vocabulary), 78

G

GenericTrainer (class in neuralmonkey.trainers.generic_trainer), 64
GermanPostprocessor (class in neuralmonkey.processors.german), 50
GermanPreprocessor (class in neuralmonkey.processors.german), 50
get_attention_object() (neuralmonkey.decoders.decoder.Decoder method), 25
get_encoder_projections() (neuralmonkey.encoders.encoder_wrapper.FlatMultiAttention method), 33
get_executable() (neuralmonkey.runners.base_runner.BaseRunner method), 52
get_executable() (neuralmonkey.runners.beamsearch_runner.BeamSearchRunner method), 53
get_executable() (neuralmonkey.runners.label_runner.LabelRunner method), 54
get_executable() (neuralmonkey.runners.logits_runner.LogitsRunner method), 55
get_executable() (neuralmonkey.runners.perplexity_runner.PerplexityRunner method), 55
get_executable() (neuralmonkey.runners.plain_runner.PlainRunner method), 56
get_executable() (neuralmonkey.runners.regression_runner.RegressionRunner method), 57
get_executable() (neuralmonkey.runners.representation_runner.RepresentationRunner method), 57
get_executable() (neuralmonkey.runners.rnn_runner.RuntimeRnnRunner method), 58
get_executable() (neuralmonkey.runners.runner.GreedyRunner method), 59
get_executable() (neuralmonkey.runners.word_alignment_runner.WordAlignmentRunner method), 60
get_executable() (neuralmonkey.trainers.generic_trainer.GenericTrainer method), 64
get_logits() (neuralmonkey.decoding_function.Attention method), 69
get_logits() (neuralmonkey.decoding_function.CoverageAttention method), 70
get_plain_text_reader() (in module neuralmonkey.readers.plain_text_reader), 51
get_series() (neuralmonkey.dataset.Dataset method), 67
get_series() (neuralmonkey.dataset.LazyDataset method), 68
get_string_vector_reader() (in module neuralmonkey.readers.string_vector_reader), 52
get_unk_sampled_word_index() (neuralmonkey.vocabulary.Vocabulary method), 76
get_word_index() (neuralmonkey.vocabulary.Vocabulary method), 76
gleu() (neuralmonkey.evaluators.gleu.GLEUEvaluator static method), 44
GLEUEvaluator (class in neuralmonkey.evaluators.gleu), 43
gpu_memusage() (in module neuralmonkey.tf_utils), 75
gradients (neuralmonkey.trainers.generic_trainer.Objective attribute), 64

GreedyRunExecutable (class in neuralmonkey.runners.runner), 59
GreedyRunner (class in neuralmonkey.runners.runner), 59
gt_inputs (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 28

H

has_gpu() (in module neuralmonkey.tf_utils), 75
has_series() (neuralmonkey.dataset.Dataset method), 68
has_series() (neuralmonkey.dataset.LazyDataset method), 68
HEIGHT (neuralmonkey.encoders.imagenet_encoder.ImageNet attribute), 35
HierarchicalMultiAttention (class in neuralmonkey.encoders.encoder_wrapper), 33
highway() (in module neuralmonkey.nn.highway), 46
highway_layer (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 38
histogram_summaries (neuralmonkey.runners.base_runner.ExecutionResult attribute), 52

I

ignore_argument() (neuralmonkey.config.configuration.Configuration method), 21
image_input (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 32
image_processing_layers (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 32
image_reader() (in module neuralmonkey.readers.image_reader), 51
image_summaries (neuralmonkey.runners.base_runner.ExecutionResult attribute), 52
ImageNet (class in neuralmonkey.encoders.imagenet_encoder), 34
imagenet_reader() (in module neuralmonkey.readers.image_reader), 51
IniError, 22
init_saving() (neuralmonkey.tf_manager.TensorFlowManager method), 75
initialize_for_running() (in module neuralmonkey.run), 74
initialize_model_parts() (neuralmonkey.tf_manager.TensorFlowManager method), 75
initialize_vocabulary() (in module neuralmonkey.vocabulary), 78
input_image (neuralmonkey.encoders.imagenet_encoder.ImageNet attribute), 35
input_lengths (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 24
input_mask (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNE attribute), 38
input_mask (neuralmonkey.encoders.sequence_cnn_encoder.SequenceCNN attribute), 41
inputs (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncod attribute), 38
inputs (neuralmonkey.encoders.sequence_cnn_encoder.SequenceCNNEnco attribute), 41
instantiate_class() (in module neuralmonkey.config.builder), 21
IntVectorReader() (in module neuralmonkey.readers.string_vector_reader), 51
inverse_sigmoid_decay() (in module neuralmonkey.functions), 70

L

LabelRunExecutable (class in neuralmonkey.runners.label_runner), 54
LabelRunner (class in neuralmonkey.runners.label_runner), 54
last_attns (neuralmonkey.decoders.beam_search_decoder.SearchState attribute), 23
last_state (neuralmonkey.decoders.beam_search_decoder.SearchState attribute), 23
last_word_ids (neuralmonkey.decoders.beam_search_decoder.SearchState attribute), 23
LazyDataset (class in neuralmonkey.dataset), 68
lengths (neuralmonkey.decoders.beam_search_decoder.SearchState attribute), 23
likelihood_beam_score() (in module neuralmonkey.runners.rnn_runner), 59
linear() (in module neuralmonkey.nn.projection), 47
linear_encoder_projection() (in module neuralmonkey.decoders.encoder_projection), 26
load() (neuralmonkey.model.model_part.ModelPart method), 45
load_dataset_from_files() (in module neuralmonkey.dataset), 69
load_file() (neuralmonkey.config.configuration.Configuration method), 21
log() (in module neuralmonkey.logging), 74
log() (neuralmonkey.logging.Logging static method), 73
log_file (neuralmonkey.logging.Logging attribute), 73
log_print() (in module neuralmonkey.logging), 74
log_print() (neuralmonkey.logging.Logging static method), 73
log_sample() (neuralmonkey.vocabulary.Vocabulary method), 76
Logging (class in neuralmonkey.logging), 73
logits (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 24

logits (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 29
LogitsExecutable (class in neuralmonkey.runners.logits_runner), 54
LogitsRunner (class in neuralmonkey.runners.logits_runner), 55
logprob_sum (neuralmonkey.decoders.beam_search_decoder.SearchState attribute), 23
logprobs (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 29
logprobs (neuralmonkey.runners.rnn_runner.BeamBatch attribute), 58
loss (neuralmonkey.trainers.generic_trainer.Objective attribute), 64
loss_names (neuralmonkey.runners.base_runner.BaseRunner attribute), 52
loss_names (neuralmonkey.runners.beamsearch_runner.BeamSearchRunner attribute), 53
loss_names (neuralmonkey.runners.label_runner.LabelRunner attribute), 54
loss_names (neuralmonkey.runners.logits_runner.LogitsRunner attribute), 55
loss_names (neuralmonkey.runners.perplexity_runner.PerplexityRunner attribute), 56
loss_names (neuralmonkey.runners.plain_runner.PlainRunner attribute), 56
loss_names (neuralmonkey.runners.regression_runner.RegressionRunner attribute), 57
loss_names (neuralmonkey.runners.representation_runner.RepresentationRunner attribute), 57
loss_names (neuralmonkey.runners.rnn_runner.RuntimeRnnRunner attribute), 59
loss_names (neuralmonkey.runners.runner.GreedyRunner attribute), 59
loss_names (neuralmonkey.runners.word_alignment_runner.WordAlignmentRunner attribute), 60
loss_with_decoded_ins (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 28
loss_with_gt_ins (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 29
losses (neuralmonkey.runners.base_runner.ExecutionResult attribute), 52

M

main() (in module neuralmonkey.run), 74
main() (in module neuralmonkey.server), 74
main() (in module neuralmonkey.train), 75
make_namespace() (neuralmonkey.config.configuration.Configuration method), 21
maxout() (in module neuralmonkey.nn.projection), 48
maxout_output() (in module neuralmonkey.decoders.output_projection), 27
MeanSquaredErrorEvaluator (class in neuralmonkey.evaluators.mse), 44
merge_max_counters() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 42
minimum_reference_length() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 42
mlp_output() (in module neuralmonkey.decoders.output_projection), 27
ModelPart (class in neuralmonkey.model.model_part), 45
modified_ngram_precision() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 42
MultevalWrapper (class in neuralmonkey.evaluators.multeval), 44
MultiAttention (class in neuralmonkey.encoders.encoder_wrapper), 33
multilayer_projection() (in module neuralmonkey.nn.projection), 48
MultiLayerPerceptron (class in neuralmonkey.nn.mlp), 46

N

n_best() (in module neuralmonkey.runners.rnn_runner), 59
name (neuralmonkey.model.model_part.ModelPart attribute), 45
name (neuralmonkey.trainers.generic_trainer.Objective attribute), 65
neuralmonkey (module), 20, 78
neuralmonkey.checking (module), 67
neuralmonkey.config (module), 22
neuralmonkey.config.builder (module), 20
neuralmonkey.config.configuration (module), 21
neuralmonkey.config.exceptions (module), 21
neuralmonkey.config.parsing (module), 22
neuralmonkey.config.utils (module), 22
neuralmonkey.dataset (module), 67
neuralmonkey.decoders (module), 31
neuralmonkey.decoders.beam_search_decoder (module), 23
neuralmonkey.decoders.ctc_decoder (module), 24
neuralmonkey.decoders.decoder (module), 25
neuralmonkey.decoders.encoder_projection (module), 25
neuralmonkey.decoders.output_projection (module), 27
neuralmonkey.decoders.sequence_classifier (module), 28
neuralmonkey.decoders.sequence_labeler (module), 29
neuralmonkey.decoders.sequence_regressor (module), 30
neuralmonkey.decoders.word_alignment_decoder (module), 31
neuralmonkey.decoding_function (module), 69
neuralmonkey.decorators (module), 70

neuralmonkey.encoders (module), 41
neuralmonkey.encoders.attentive (module), 31
neuralmonkey.encoders.cnn_encoder (module), 31
neuralmonkey.encoders.encoder_wrapper (module), 32
neuralmonkey.encoders.factored_encoder (module), 34
neuralmonkey.encoders.imagenet_encoder (module), 34
neuralmonkey.encoders.numpy_encoder (module), 35
neuralmonkey.encoders.raw_rnn_encoder (module), 35
neuralmonkey.encoders.sentence_cnn_encoder (module), 36
neuralmonkey.encoders.sentence_encoder (module), 39
neuralmonkey.encoders.sequence_cnn_encoder (module), 40
neuralmonkey.evaluators (module), 45
neuralmonkey.evaluators.accuracy (module), 41
neuralmonkey.evaluators.average (module), 41
neuralmonkey.evaluators.beer (module), 41
neuralmonkey.evaluators.bleu (module), 42
neuralmonkey.evaluators.bleu_ref (module), 43
neuralmonkey.evaluators.edit_distance (module), 43
neuralmonkey.evaluators.f1_bio (module), 43
neuralmonkey.evaluators.gleu (module), 43
neuralmonkey.evaluators.mse (module), 44
neuralmonkey.evaluators.multeval (module), 44
neuralmonkey.evaluators.ter (module), 45
neuralmonkey.evaluators.wer (module), 45
neuralmonkey.functions (module), 70
neuralmonkey.learning_utils (module), 71
neuralmonkey.logging (module), 73
neuralmonkey.model (module), 45
neuralmonkey.model.model_part (module), 45
neuralmonkey.nn (module), 49
neuralmonkey.nn.highway (module), 46
neuralmonkey.nn.mlp (module), 46
neuralmonkey.nn.noisy_gru_cell (module), 46
neuralmonkey.nn.ortho_gru_cell (module), 47
neuralmonkey.nn.pervasive_dropout_wrapper (module), 47
neuralmonkey.nn.projection (module), 47
neuralmonkey.nn.utils (module), 48
neuralmonkey.processors (module), 50
neuralmonkey.processors.alignment (module), 49
neuralmonkey.processors.bpe (module), 49
neuralmonkey.processors.editops (module), 50
neuralmonkey.processors.german (module), 50
neuralmonkey.processors.helpers (module), 50
neuralmonkey.readers (module), 52
neuralmonkey.readers.image_reader (module), 51
neuralmonkey.readers.numpy_reader (module), 51
neuralmonkey.readers.plain_text_reader (module), 51
neuralmonkey.readers.string_vector_reader (module), 51
neuralmonkey.readers.utils (module), 52
neuralmonkey.run (module), 74
neuralmonkey.runners (module), 60
neuralmonkey.runners.base_runner (module), 52
neuralmonkey.runners.beamsearch_runner (module), 53
neuralmonkey.runners.label_runner (module), 54
neuralmonkey.runners.logits_runner (module), 54
neuralmonkey.runners.perplexity_runner (module), 55
neuralmonkey.runners.plain_runner (module), 56
neuralmonkey.runners.regression_runner (module), 56
neuralmonkey.runners.representation_runner (module), 57
neuralmonkey.runners.rnn_runner (module), 58
neuralmonkey.runners.runner (module), 59
neuralmonkey.runners.word_alignment_runner (module), 60
neuralmonkey.server (module), 74
neuralmonkey.tests (module), 63
neuralmonkey.tests.test_bleu (module), 60
neuralmonkey.tests.test_config (module), 60
neuralmonkey.tests.test_dataset (module), 61
neuralmonkey.tests.test_decoder (module), 61
neuralmonkey.tests.test_encoders_init (module), 61
neuralmonkey.tests.test_eval_wrappers (module), 61
neuralmonkey.tests.test_functions (module), 62
neuralmonkey.tests.test_model_part (module), 62
neuralmonkey.tests.test_nn_utils (module), 62
neuralmonkey.tests.test_readers (module), 62
neuralmonkey.tests.test_ter (module), 63
neuralmonkey.tests.test_vocabulary (module), 63
neuralmonkey.tf_manager (module), 74
neuralmonkey.tf_utils (module), 75
neuralmonkey.train (module), 75
neuralmonkey.trainers (module), 66
neuralmonkey.trainers.cross_entropy_trainer (module), 64
neuralmonkey.trainers.generic_trainer (module), 64
neuralmonkey.trainers.self_critical_objective (module), 65
neuralmonkey.vocabulary (module), 75
next_logprobs (neuralmonkey.runners.rnn_runner.ExpandedBeamBatch attribute), 58
next_to_execute() (neuralmonkey.runners.base_runner.Executable method), 52
next_to_execute() (neuralmonkey.runners.beamsearch_runner.BeamSearchExecutable method), 53
next_to_execute() (neuralmonkey.runners.label_runner.LabelRunExecutable method), 54
next_to_execute() (neuralmonkey.runners.logits_runner.LogitsExecutable method), 55
next_to_execute() (neuralmonkey.runners.perplexity_runner.PerplexityExecutable method), 55

next_to_execute() (neuralmonkey.runners.plain_runner.PlainExecutable method), 56
next_to_execute() (neuralmonkey.runners.regression_runner.RegressionRunExecutable method), 56
next_to_execute() (neuralmonkey.runners.representation_runner.RepresentationExecutable method), 57
next_to_execute() (neuralmonkey.runners.rnn_runner.RuntimeRnnExecutable method), 58
next_to_execute() (neuralmonkey.runners.runner.GreedyRunExecutable method), 59
next_to_execute() (neuralmonkey.runners.word_alignment_runner.WordAlignmentRunnerExecutable method), 60
next_to_execute() (neuralmonkey.trainers.generic_trainer.TrainExecutable method), 65
ngram_counts() (neuralmonkey.evaluators.bleu.BLEUEvaluator static method), 42
no_deep_output() (in module neuralmonkey.decoders.output_projection), 27
noisy_activation() (in module neuralmonkey.nn.noisy_gru_cell), 46
noisy_sigmoid() (in module neuralmonkey.nn.noisy_gru_cell), 47
noisy_tanh() (in module neuralmonkey.nn.noisy_gru_cell), 47
NoisyGRUCell (class in neuralmonkey.nn.noisy_gru_cell), 46
nonlinear() (in module neuralmonkey.nn.projection), 48
notice() (in module neuralmonkey.logging), 74
notice() (neuralmonkey.logging.Logging static method), 73
numpy_reader() (in module neuralmonkey.readers.numpy_reader), 51

O

Objective (class in neuralmonkey.trainers.generic_trainer), 64
OrthoGRUCell (class in neuralmonkey.nn.ortho_gru_cell), 47
output_size (neuralmonkey.nn.noisy_gru_cell.NoisyGRUCell attribute), 46
output_size (neuralmonkey.nn.pervasive_dropout_wrapper.PervasiveDropoutWrapper attribute), 47
outputs (neuralmonkey.runners.base_runner.ExecutionResult attribute), 52

P

parent_ids (neuralmonkey.decoders.beam_search_decoder.SearchStepOutput attribute), 24
parse_file() (in module neuralmonkey.config.parsing), 22
PerplexityExecutable (class in neuralmonkey.runners.perplexity_runner), 55
PerplexityRunner (class in neuralmonkey.runners.perplexity_runner), 55
PervasiveDropoutWrapper (class in neuralmonkey.nn.pervasive_dropout_wrapper), 47
piecewise_function() (in module neuralmonkey.functions), 71
pipeline() (in module neuralmonkey.processors.helpers), 50
PlainExecutable (class in neuralmonkey.runners.plain_runner), 56
PlainRunner (class in neuralmonkey.runners.plain_runner), 56
post_request() (in module neuralmonkey.server), 74
PostCNNImageEncoder (class in neuralmonkey.encoders.numpy_encoder), 35
Postprocess (class in neuralmonkey.processors.editops), 50
postprocess_char_based() (in module neuralmonkey.processors.helpers), 50
predictions (neuralmonkey.decoders.sequence_regressor.SequenceRegressor attribute), 30
Preprocess (class in neuralmonkey.processors.editops), 50
preprocess_char_based() (in module neuralmonkey.processors.helpers), 50
print_final_evaluation() (in module neuralmonkey.learning_utils), 71
print_header() (neuralmonkey.logging.Logging static method), 73

R

ratio() (neuralmonkey.evaluators.edit_distance.EditDistanceEvaluator static method), 43
RawRNNEncoder (class in neuralmonkey.encoders.raw_rnn_encoder), 36
reconstruct() (in module neuralmonkey.processors.editops), 50
RecurrentAttention (class in neuralmonkey.decoding_function), 70
reduce_execution_results() (in module neuralmonkey.runners.base_runner), 53
RegressionRunExecutable (class in neuralmonkey.runners.regression_runner), 56
RegressionRunner (class in neuralmonkey.runners.regression_runner), 56
reinforce_score() (in module neuralmonkey.trainers.self_critical_objective), 65

RepresentationExecutable (class in neuralmonkey.runners.representation_runner), 57
RepresentationRunner (class in neuralmonkey.runners.representation_runner), 57
restore() (neuralmonkey.tf_manager.TensorFlowManager method), 75
restore_best_vars() (neuralmonkey.tf_manager.TensorFlowManager method), 75
rnn_cells() (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder method), 38
rnn_cells() (neuralmonkey.encoders.sentence_encoder.SentenceEncoder method), 39
RNNSpec (class in neuralmonkey.encoders.raw_rnn_encoder), 35
run_on_dataset() (in module neuralmonkey.learning_utils), 71
runtime_logprobs (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 29
runtime_loss (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 24
runtime_loss (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 29
runtime_loss (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 29
runtime_loss (neuralmonkey.decoders.sequence_regressor.SequenceRegressor attribute), 30
RuntimeRnnExecutable (class in neuralmonkey.runners.rnn_runner), 58
RuntimeRnnRunner (class in neuralmonkey.runners.rnn_runner), 58

S

save() (neuralmonkey.model.model_part.ModelPart method), 45
save() (neuralmonkey.tf_manager.TensorFlowManager method), 75
save_file() (neuralmonkey.config.configuration.Configuration method), 21
save_to_file() (neuralmonkey.vocabulary.Vocabulary method), 76
scalar_summaries (neuralmonkey.runners.base_runner.ExecutionResult attribute), 53
scores (neuralmonkey.decoders.beam_search_decoder.SearchStepOutput attribute), 24
SearchState (class in neuralmonkey.decoders.beam_search_decoder), 23
SearchStepOutput (class in neuralmonkey.decoders.beam_search_decoder), 24
self_critical_objective() (in module neuralmonkey.trainers.self_critical_objective), 65
sentence_bleu() (in module neuralmonkey.trainers.self_critical_objective), 66
sentence_gleu() (in module neuralmonkey.trainers.self_critical_objective), 66
sentence_lengths (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 38
SentenceCNNEncoder (class in neuralmonkey.encoders.sentence_cnn_encoder), 36
SentenceEncoder (class in neuralmonkey.encoders.sentence_encoder), 39
sentences_to_tensor() (neuralmonkey.vocabulary.Vocabulary method), 76
SequenceClassifier (class in neuralmonkey.decoders.sequence_classifier), 28
SequenceCNNEncoder (class in neuralmonkey.encoders.sequence_cnn_encoder), 40
SequenceLabeler (class in neuralmonkey.decoders.sequence_labeler), 29
SequenceRegressor (class in neuralmonkey.decoders.sequence_regressor), 30
serialize_to_bytes() (neuralmonkey.evaluators.beer.BeerWrapper method), 41
serialize_to_bytes() (neuralmonkey.evaluators.bleu_ref.BLEUReferenceImplWrapper method), 43
serialize_to_bytes() (neuralmonkey.evaluators.multeval.MultEvalWrapper method), 44
series_ids (neuralmonkey.dataset.Dataset attribute), 68
series_ids (neuralmonkey.dataset.LazyDataset attribute), 68
sessions (neuralmonkey.tf_manager.TensorFlowManager attribute), 75
set_log_file() (neuralmonkey.logging.Logging static method), 73
setUp() (neuralmonkey.tests.test_eval_wrappers.TestAccuracyEvaluator method), 61
setUp() (neuralmonkey.tests.test_readers.TestStringVectorReader method), 62
shuffle() (neuralmonkey.dataset.Dataset method), 68
shuffle() (neuralmonkey.dataset.LazyDataset method), 68
size (neuralmonkey.encoders.raw_rnn_encoder.RNNSpec attribute), 36

softmax (neuralmonkey.nn.mlp.MultilayerPerceptron attribute), 46
state_size (neuralmonkey.nn.noisy_gru_cell.NoisyGRUCell attribute), 46
state_size (neuralmonkey.nn.pervasive_dropout_wrapper.PervasiveDropoutWrapper attribute), 47
states (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 32
states (neuralmonkey.encoders.imagenet_encoder.ImageNet attribute), 35
states (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 38
states_mask (neuralmonkey.encoders.sentence_encoder.SentenceEncoder attribute), 40
step() (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder method), 23
step() (neuralmonkey.decoders.decoder.Decoder method), 25
strict_mode (neuralmonkey.logging.Logging attribute), 73
subset() (neuralmonkey.dataset.Dataset method), 68
subset() (neuralmonkey.dataset.LazyDataset method), 68

T

tearDown() (neuralmonkey.tests.test_readers.TestStringVectorReader method), 62
tensor() (in module neuralmonkey.decorators), 70
TensorFlowManager (class in neuralmonkey.tf_manager), 74
TEREvalutator (class in neuralmonkey.evaluators.ter), 45
Test (class in neuralmonkey.tests.test_model_part), 62
test_all_words_in() (neuralmonkey.tests.test_vocabulary.TestVocabulary method), 63
test_beer() (neuralmonkey.tests.test_eval_wrappers.TestExternalEvaluators method), 61
test_bleu() (neuralmonkey.tests.test_bleu.TestBLEU method), 60
test_columns() (neuralmonkey.tests.test_readers.TestStringVectorReader method), 62
test_empty_decoded() (neuralmonkey.tests.test_bleu.TestBLEU method), 60
test_empty_decoded() (neuralmonkey.tests.test_ter.TestBLEU method), 63
test_empty_reference() (neuralmonkey.tests.test_bleu.TestBLEU method), 60
test_empty_reference() (neuralmonkey.tests.test_ter.TestBLEU method), 63
test_empty_sentence() (neuralmonkey.tests.test_bleu.TestBLEU method), 60
test_empty_sentence() (neuralmonkey.tests.test_ter.TestBLEU method), 63
test_f1() (neuralmonkey.tests.test_eval_wrappers.TestExternalEvaluators method), 61
test_gleu() (neuralmonkey.tests.test_eval_wrappers.TestExternalEvaluators method), 62
test_identical() (neuralmonkey.tests.test_bleu.TestBLEU method), 60
test_identical() (neuralmonkey.tests.test_ter.TestBLEU method), 63
test_init() (neuralmonkey.tests.test_decoder.TestDecoder method), 61
test_invalid_keep_prob() (neuralmonkey.tests.test_nn_utils.TestDropout method), 62
test_keep_prob() (neuralmonkey.tests.test_nn_utils.TestDropout method), 62
test_min_freq() (neuralmonkey.tests.test_vocabulary.TestVocabulary method), 63
test_multeval_bleu() (neuralmonkey.tests.test_eval_wrappers.TestExternalEvaluators method), 62
test_multeval_meteor() (neuralmonkey.tests.test_eval_wrappers.TestExternalEvaluators method), 62
test_multeval_ter() (neuralmonkey.tests.test_eval_wrappers.TestExternalEvaluators method), 62
test_nonexistent_file() (neuralmonkey.tests.test_dataset.TestDataset method), 61
test_padding() (neuralmonkey.tests.test_vocabulary.TestVocabulary method), 63
test_piecewise_constant() (neuralmonkey.tests.test_functions.TestPiecewiseFunction method), 62
test_post_cnn_encoder() (neuralmonkey.tests.test_encoders_init.TestEncodersInit method), 61
test_reader() (neuralmonkey.tests.test_readers.TestStringVectorReader method), 62
test_save_and_load() (neuralmonkey.tests.test_model_part.Test method), 62
test_sentence_cnn_encoder() (neuralmonkey.tests.test_encoders_init.TestEncodersInit method), 61
test_sentence_encoder() (neural-

monkey.tests.test_encoders_init.TestEncodersInit method), 61
test_seq_level_acc() (neuralmonkey.tests.test_eval_wrappers.TestAccuracyEvaluator method), 61
test_splitter_bad_brackets() (neuralmonkey.tests.test_config.TestParsing method), 60
test_splitter_gen() (in module neuralmonkey.tests.test_config), 60
test_ter() (neuralmonkey.tests.test_ter.TestBLEU method), 63
test_there_and_back_self() (neuralmonkey.tests.test_vocabulary.TestVocabulary method), 63
test_train_false() (neuralmonkey.tests.test_nn_utils.TestDropout method), 62
test_unknown_word() (neuralmonkey.tests.test_vocabulary.TestVocabulary method), 63
test_vector_encoder() (neuralmonkey.tests.test_encoders_init.TestEncodersInit method), 61
test_weights() (neuralmonkey.tests.test_vocabulary.TestVocabulary method), 63
test_word_level_acc() (neuralmonkey.tests.test_eval_wrappers.TestAccuracyEvaluator method), 61
TestAccuracyEvaluator (class in neuralmonkey.tests.test_eval_wrappers), 61
TestBLEU (class in neuralmonkey.tests.test_bleu), 60
TestBLEU (class in neuralmonkey.tests.test_ter), 63
TestDataset (class in neuralmonkey.tests.test_dataset), 61
TestDecoder (class in neuralmonkey.tests.test_decoder), 61
TestDropout (class in neuralmonkey.tests.test_nn_utils), 62
TestEncodersInit (class in neuralmonkey.tests.test_encoders_init), 61
TestExternalEvaluators (class in neuralmonkey.tests.test_eval_wrappers), 61
TestParsing (class in neuralmonkey.tests.test_config), 60
TestPiecewiseFunction (class in neuralmonkey.tests.test_functions), 62
TestStringVectorReader (class in neuralmonkey.tests.test_readers), 62
TestVocabulary (class in neuralmonkey.tests.test_vocabulary), 63
token_ids (neuralmonkey.decoders.beam_search_decoder.SearchStepOutput attribute), 24
total_precision_recall() (neuralmonkey.evaluators.gleu.GLEUEvaluator static method), 44
train_inputs (neuralmonkey.decoders.sequence_regressor.SequenceRegresso attribute), 30
train_loss (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 24
train_loss (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 29
train_loss (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 29
train_loss (neuralmonkey.decoders.sequence_regressor.SequenceRegressor attribute), 30
train_mode (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 24
train_mode (neuralmonkey.decoders.sequence_classifier.SequenceClassifier attribute), 29
train_mode (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 29
train_mode (neuralmonkey.decoders.sequence_regressor.SequenceRegresso attribute), 30
train_mode (neuralmonkey.encoders.cnn_encoder.CNNEncoder attribute), 32
train_mode (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNE attribute), 38
train_mode (neuralmonkey.encoders.sequence_cnn_encoder.SequenceCNN attribute), 41
train_targets (neuralmonkey.decoders.ctc_decoder.CTCDecoder attribute), 24
train_targets (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 29
train_weights (neuralmonkey.decoders.sequence_labeler.SequenceLabeler attribute), 29
TrainExecutable (class in neuralmonkey.trainers.generic_trainer), 65
training_loop() (in module neuralmonkey.learning_utils), 72
traverse_combinations() (in module neuralmonkey.tests.test_encoders_init), 61
truncate() (neuralmonkey.vocabulary.Vocabulary method), 77
truncate_by_min_freq() (neuralmonkey.vocabulary.Vocabulary method), 77

U

untruecase() (in module neuralmonkey.processors.helpers), 50
use_scope() (neuralmonkey.model.model_part.ModelPart method), 45
UtfPlainTextReader() (in module neuralmonkey.readers.plain_text_reader), 51

V

validation_hook() (neuralmonkey.tf_manager.TensorFlowManager method), 75

variable() (in module neuralmonkey.config.utils), 22
VectorEncoder (class in neuralmonkey.encoders.numpy_encoder), 35
vectors_to_sentences() (neuralmonkey.vocabulary.Vocabulary method), 77
Vocabulary (class in neuralmonkey.vocabulary), 75
vocabulary (neuralmonkey.decoders.beam_search_decoder.BeamSearchDecoder attribute), 23
vocabulary_from_bpe() (in module neuralmonkey.config.utils), 22
vocabulary_from_dataset() (in module neuralmonkey.config.utils), 22
vocabulary_from_file() (in module neuralmonkey.config.utils), 22
vocabulary_size (neuralmonkey.encoders.sentence_cnn_encoder.SentenceCNNEncoder attribute), 38
vocabulary_size (neuralmonkey.encoders.sentence_encoder.SentenceEncoder attribute), 40

W

warn() (in module neuralmonkey.logging), 74
warn() (neuralmonkey.logging.Logging static method), 73
weight (neuralmonkey.trainers.generic_trainer.Objective attribute), 65
WEREvaluator (class in neuralmonkey.evaluators.wer), 45
WIDTH (neuralmonkey.encoders.imagenet_encoder.ImageNet attribute), 35
WordAlignmentDecoder (class in neuralmonkey.decoders.word_alignment_decoder), 31
WordAlignmentPreprocessor (class in neuralmonkey.processors.alignment), 49
WordAlignmentRunner (class in neuralmonkey.runners.word_alignment_runner), 60
WordAlignmentRunnerExecutable (class in neuralmonkey.runners.word_alignment_runner), 60
write_file() (in module neuralmonkey.config.parsing), 22

X

xent_objective() (in module neuralmonkey.trainers.cross_entropy_trainer), 64