# Neural Monkey Documentation

*Release 0.1*

Jindřich Libovický, Jindřich Helcl, Tomáš Musil

Mar 12, 2017

# Contents

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!

## 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
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)]
minimize=True
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
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)]
minimize=False
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>]

[eval_data]
class=dataset.load_dataset_from_files
s_source="exp-nm-mt/data/test/Batch3a_en.txt.gz"
```

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.

## Developers' guidelines

**Warning:** This document is far from finished. For now, it should serve as a collection of hints for new developers.

This is a brief document describing the Neural Monkey development workflow.

### Commiting code

Use pull requests to introduce new changes. 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 for the changes. Use `git checkout -b branch_name` to create a new branch and switch the working copy to that branch.
2. Make your changes to the code, commit them to the new branch.
3. Push the new branch to the repository by typing `git push origin branch_name`, or just `git push`.
4. You should now see the new branch on the Github project page. When you open the branch page, click on "Create Pull request" button.
5. When the pull request is created, the 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.
6. Simultaneously to the test runs, anyone now can look at the code changes by clicking on the "Files changed" tab and comment on individual lines in the diff, and approve or reject the proposed pull request.
7. When all the tests are passing and the pull request is approved, it can be merged.

**Note:** When you are pushing a commit that does not change the functionality of the code (e.g. changes in comments, documentation, etc.), you can push them to master directly (TODO: subject to discussion). Every commit that does not affect the program behavior should be marked with `[ci skip]` inside the commit message.

### Documentation

Documentation related to GitHub is written in *Markdown <https://daringfireball.net/projects/markdown/>* files, Python documentation using *reStructuredText <http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html>*. This concerns both the standalone documents (in */docs/*) and the docstrings in source code.

Style of the Markdown files is automatically checked using *Markdownlint <https://github.com/mivok/markdownlint>*.

### Other

**Todo**

describe other stuff too

## 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*`)`

Bases: `object`

Represents a class (or other callable) in configuration.

**`create`**`()`

`neuralmonkey.config.builder.`**`build_config`**`(`*config_dicts*, *ignore_names*`)`

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.

`neuralmonkey.config.builder.`**`build_object`**`(`*value*, *all_dicts*, *existing_objects*, *depth*`)`

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*, *all_dicts*, *existing_objects*, *depth*)

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*, *arg_type=<class 'object'>*, *required=False*, *default=None*, *cond=None*) → None

**`build_model`**() → None

**`ignore_argument`**(*name: str*) → None

**`load_file`**(*path: str*) → None

**`make_namespace`**(*d_obj*) → argparse.Namespace

## neuralmonkey.config.exceptions module

Module that contains exceptions handled in config parsing and loading

**exception** `neuralmonkey.config.exceptions.`**`ConfigBuildException`**(*object_name*, *original_exception*)

Bases: `Exception`

Exception caused by error in loading the model

**exception** `neuralmonkey.config.exceptions.`**`ConfigInvalidValueException`**(*value*, *message*)

Bases: `Exception`

**exception** `neuralmonkey.config.exceptions.`**`IniError`**(*line*, *message*, *original_exc=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*)

Parses an INI file and creates all values

### neuralmonkey.config.utils module

### Module contents

## neuralmonkey.decoders package

### Submodules

### neuralmonkey.decoders.decoder module

### 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.fra 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.multi_decoder module

## 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_plc=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

### neuralmonkey.decoders.sequence_labeler module

### neuralmonkey.decoders.word_alignment_decoder module

### 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

### neuralmonkey.encoders.factored_encoder module

### neuralmonkey.encoders.imagenet_encoder module

### neuralmonkey.encoders.numpy_encoder module

### neuralmonkey.encoders.sentence_encoder module

### Module contents

### neuralmonkey.evaluators package

#### Submodules

#### neuralmonkey.evaluators.accuracy module

**class** `neuralmonkey.evaluators.accuracy.`**`Accuracy`**(*name='Accuracy'*)

Bases: `object`

**static** **`compare_scores`**(*score1*, *score2*)

#### neuralmonkey.evaluators.bleu module

**class** `neuralmonkey.evaluators.bleu.`**`BLEUEvaluator`**(*n=4*, *deduplicate=False*, *name=None*)

Bases: `object`

**static** **`bleu`**(*hypotheses*, *references*, *ngrams=4*, *case_sensitive=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.
- **`ngram`** – Maximum order of n-grams. Default 4.
- **`case_sensitive`** – Perform case-sensitive computation. Default True.

**static** **`compare_scores`**(*score1*, *score2*)

**static** **`effective_reference_length`**(*hypotheses*, *references_list*)

Computes the effective reference corpus length (based on best match length)

**Parameters**

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

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 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[str]]], n: int, case_sensitive: bool*) → typing.Tuple[float, int]

Computes the modified n-gram precision on a list of sentences

**Parameters**

- **`hypothesis`** – List of output sentences as lists of words

- **references** – 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=' '*) → 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*)

### neuralmonkey.evaluators.edit_distance module

class neuralmonkey.evaluators.edit_distance.**EditDistanceEvaluator**(*name='Edit distance'*)

Bases: object

static **compare_scores**(*score1, score2*)

static **ratio**(*str1, str2*)

### neuralmonkey.evaluators.perplexity module

class neuralmonkey.evaluators.perplexity.**Perplexity**(*name='Perplexity'*)

Bases: object

### neuralmonkey.evaluators.ter module

### Module contents

### neuralmonkey.model package

### Submodules

### neuralmonkey.model.model_part module

#### Module contents

### neuralmonkey.nn package

#### Submodules

#### neuralmonkey.nn.bidirectional_rnn_layer module

**class** `neuralmonkey.nn.bidirectional_rnn_layer.`**`BidirectionalRNNLayer`**(*forward_cell*, *backward_cell*, *inputs*, *sentence_lengths_placeholder*)

Bases: `object`

Bidirectional RNN Layer class - forward and backward RNN layers in one.

**`encoded`**
Last state of the bidirectional layer

**`outputs_bidi`**
Outputs of the bidirectional layer

#### neuralmonkey.nn.init_ops module

`neuralmonkey.nn.init_ops.`**`orthogonal_initializer`**(*gain=1.0*, *dtype=tf.float32*, *seed=None*)

Returns an initializer that generates an orthogonal matrix or a reshaped orthogonal matrix.

If the shape of the tensor to initialize is two-dimensional, i is initialized with an orthogonal matrix obtained from the singular value decomposition of a matrix of uniform random numbers.

If the shape of the tensor to initialize is more than two-dimensional, a matrix of shape *(shape[0] * ... * shape[n - 2], shape[n - 1])* is initialized, where *n* is the length of the shape vector. The matrix is subsequently reshaped to give a tensor of the desired shape.

**Parameters**

- **`gain`** – multiplicative factor to apply to the orthogonal matrix
- **`dtype`** – The type of the output.
- **`seed`** – A Python integer. Used to create random seeds.

**Returns** An initializer that generates orthogonal tensors

**Raises**

- `ValueError` – if *dtype* is not a floating point type or if *shape* has
- fewer than two entries.

### neuralmonkey.nn.mlp module

**class** `neuralmonkey.nn.mlp.`**`MultilayerPerceptron`**(*mlp_input*, *layer_configuration*, *dropout_plc*, *output_size*, *name='multilayer_perceptron'*)

Bases: `object`

General implementation of the multilayer perceptron.

**`classification`**

**`softmax`**

`neuralmonkey.nn.mlp.`**`multilayer_perceptron`**(*input_*, *layer_sizes*, *activation=<function tanh>*, *dropout_plc=None*, *scope='mlp'*)

### neuralmonkey.nn.noisy_gru_cell module

**class** `neuralmonkey.nn.noisy_gru_cell.`**`NoisyGRUCell`**(*num_units*, *training*)

Bases: `tensorflow.python.ops.rnn_cell.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=1.1*, *c=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>*)

Bases: `tensorflow.python.ops.rnn_cell.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*)

Bases: `tensorflow.python.ops.rnn_cell.RNNCell`

**`output_size`**

**`state_size`**

### neuralmonkey.nn.projection module

This module implements various types of projections.

`neuralmonkey.nn.projection.`**`linear`**(*inputs*, *size*, *scope='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*, *size*, *scope='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.`**`nonlinear`**(*inputs*, *size*, *activation=<function tanh>*, *scope='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`**(***kwargs*)

Bases: `object`

**`decode`**(*sentence*)

**class** `neuralmonkey.processors.bpe.`**`BPEPreprocessor`**(***kwargs*)

Bases: `object`

Wrapper class for Byte-Pair-Encoding from Edinburgh

### neuralmonkey.processors.editops module

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

### Module contents

### neuralmonkey.readers package

### Submodules

### 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.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.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.rnn_runner module

## neuralmonkey.runners.runner module

**class** `neuralmonkey.runners.runner.`**GreedyRunExecutable**(*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.runner.`**`GreedyRunner`**(*output_series: str, decoder, 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`**

### 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_decoder module

### 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

#### neuralmonkey.tests.test_ter module

#### neuralmonkey.tests.test_vocabulary module

#### Module contents

### neuralmonkey.trainers package

#### Submodules

#### neuralmonkey.trainers.cross_entropy_trainer module

#### 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]]]

### 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.`**assert_type**(*obj*, *name*, *value*, *expected_type*, *can_be_none=False*)

`neuralmonkey.checking.`**check_dataset_and_coders**(*dataset*, *runners*)

`neuralmonkey.checking.`**missing_attributes**(*obj*, *attributes*)

`neuralmonkey.checking.`**type_to_str**(*type_obj*)

### neuralmonkey.dataset module

### 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*, *scope*, *input_weights=None*, *attention_fertility=None*)
Bases: `object`

**attention**(*query_state*)
Put attention masks on att_states_reshaped using hidden_features and query.

**get_logits**(*y*)

**class** `neuralmonkey.decoding_function.`**`CoverageAttention`**(*attention_states*, *scope*, *input_weights=None*, *attention_fertility=5*)

Bases: `neuralmonkey.decoding_function.Attention`

**`get_logits`**(*y*)

## neuralmonkey.decorators module

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

## neuralmonkey.functions module

`neuralmonkey.functions.`**`inverse_sigmoid_decay`**(*param*, *rate*, *min_value=0.0*, *max_value=1.0*, *name=None*, *dtype=tf.float32*)

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.logging module

**class** `neuralmonkey.logging.`**`Logging`**

Bases: `object`

**static** **`debug`**(*message*, *label=None*)

**`debug_disabled`** **= ['‘]**

**`debug_enabled`** **= ['none']**

**static** **`log`**(*message*, *color='yellow'*)

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** `print_header`(*title*)
Prints the title of the experiment and the set of arguments it uses.

**static** `set_log_file`(*path*)
Sets up the file where the logging will be done.

`neuralmonkey.logging.debug`(*message*, *label=None*)

`neuralmonkey.logging.log`(*message*, *color='yellow'*)
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.run module

### neuralmonkey.server module

### neuralmonkey.tf_manager module

### 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

### neuralmonkey.vocabulary module

### 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>
```

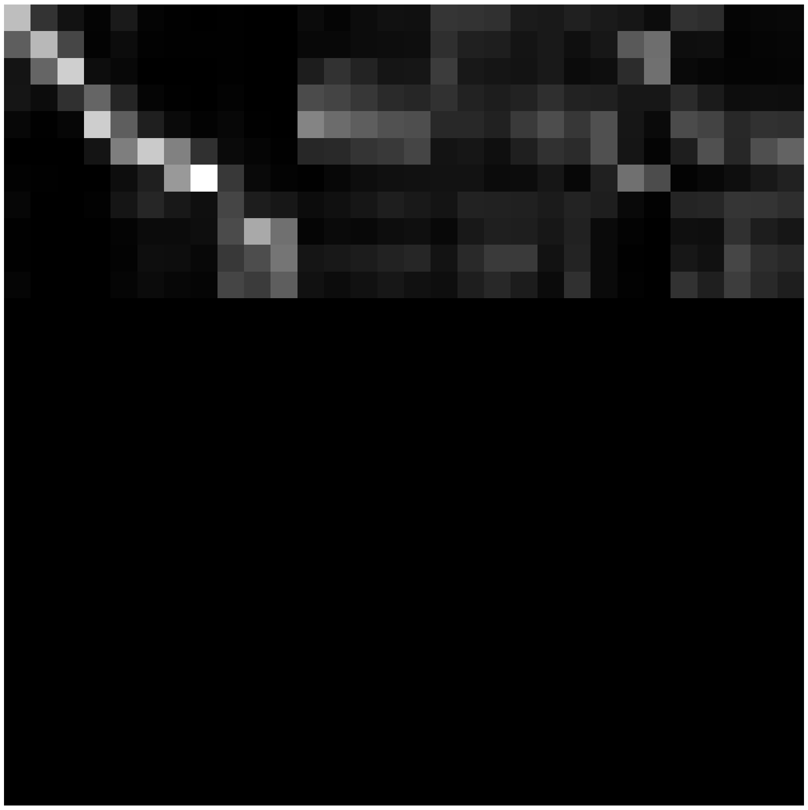
And then you can navigate in your browser to *http://localhost:6006/* (or if the TensorBoard assigns different port) and view all the summaries about your experiment.

### 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

# Python Module Index

# Index