NMT-Keras Documentation

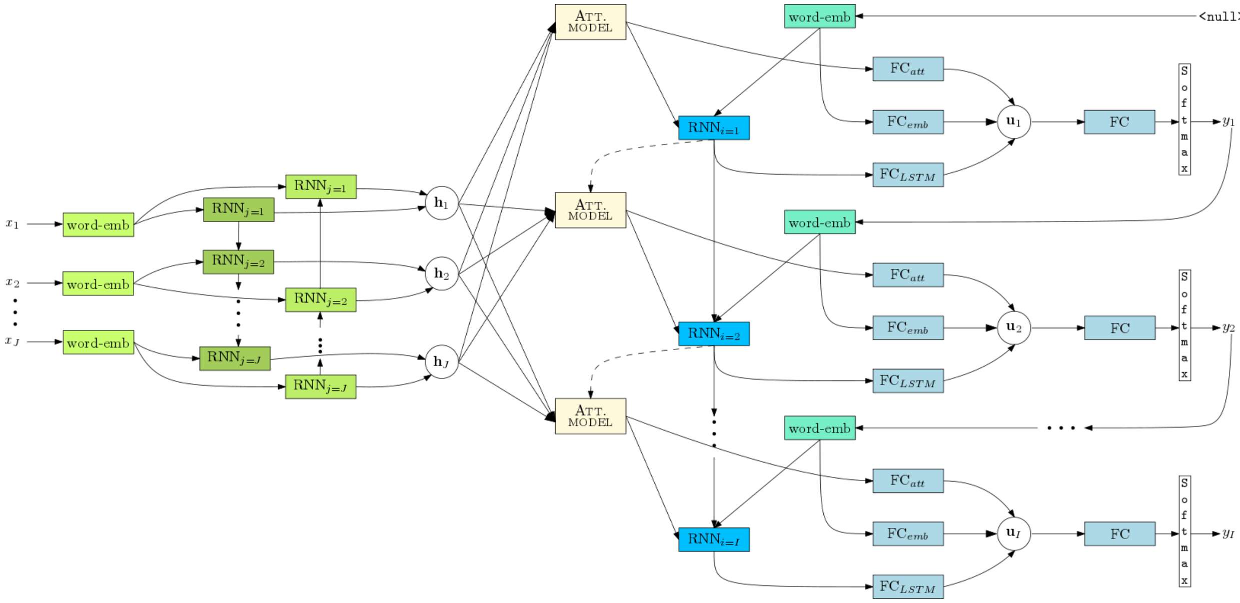
Release 0.2

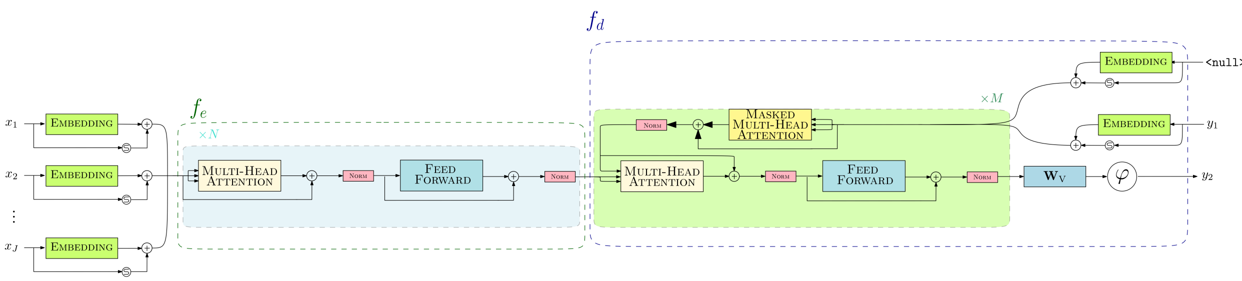
Álvaro Peris

Apr 01, 2019

Contents

Neural Machine Translation with Keras (Theano and Tensorflow).





CHAPTER 1

Features

- Attention RNN and Transformer models.
- Online learning and Interactive neural machine translation (INMT). See the interactive NMT branch.
- Tensorboard integration. Training process, models and word embeddings visualization.
- Attention model over the input sequence of annotations. - Supporting Bahdanau (Add) and Luong (Dot) attention mechanisms. - Also supports double stochastic attention.
- Peeked decoder: The previously generated word is an input of the current timestep.
- Beam search decoding. - Featuring length and source coverage normalization.
- Ensemble decoding.
- Translation scoring.
- N-best list generation (as byproduct of the beam search process).
- Support for GRU/LSTM networks: - Regular GRU/LSTM units. - Conditional GRU/LSTM units in the decoder. - Multilayered residual GRU/LSTM networks.
- Unknown words replacement.
- Use of pretrained (Glove or Word2Vec) word embedding vectors.
- MLPs for initializing the RNN hidden and memory state.
- Spearmint wrapper for hyperparameter optimization.
- Client-server architecture for web demos.

CHAPTER 2

Guide

2.1 Installation

Assuming that you have pip installed, run:

```
git clone https://github.com/lvapeab/nmt-keras
cd nmt-keras
pip install -r requirements.txt
```

for obtaining the required packages for running this library.

Nevertheless, it is highly recommended to install and configure Theano or Tensorflow with the GPU and speed optimizations enabled.

2.1.1 Requirements

- Our version of Keras.
- Multimodal Keras Wrapper. See the documentation and tutorial.
- Coco-caption evaluation package (Only required to perform evaluation).

2.2 Usage

2.2.1 Training

1) Set a training configuration in the config.py script. Each parameter is commented. See the documentation file for further info about each specific hyperparameter. You can also specify the parameters when calling the main.py script following the syntax *Key=Value*
2) Train!:

```
python main.py
```

2.2.2 Decoding

Once we have our model trained, we can translate new text using the sample_ensemble.py script. Please refer to the ensembling tutorial for more details about this script. In short, if we want to use the models from the first three epochs to translate the *examples/EuTrans/test.en* file, just run:

```
python sample_ensemble.py --models trained_models/tutorial_model/epoch_1 \
                                   trained_models/tutorial_model/epoch_2 \
                          --dataset datasets/Dataset_tutorial_dataset.pkl \
                          --text examples/EuTrans/test.en
```

2.2.3 Scoring

The score.py script can be used to obtain the (-log)probabilities of a parallel corpus. Its syntax is the following:

```
python score.py --help
usage: Use several translation models for scoring source--target pairs
   [-h] -ds DATASET [-src SOURCE] [-trg TARGET] [-s SPLITS [SPLITS ...]]
   [-d DEST] [-v] [-c CONFIG] --models MODELS [MODELS ...]
optional arguments:
    -h, --help            show this help message and exit
    -ds DATASET, --dataset DATASET
                        Dataset instance with data
    -src SOURCE, --source SOURCE
                        Text file with source sentences
    -trg TARGET, --target TARGET
                        Text file with target sentences
    -s SPLITS [SPLITS ...], --splits SPLITS [SPLITS ...]
                        Splits to sample. Should be already included into the
                        dataset object.
    -d DEST, --dest DEST  File to save scores in
    -v, --verbose         Be verbose
    -c CONFIG, --config CONFIG
                        Config pkl for loading the model configuration. If not
                        specified, hyperparameters are read from config.py
    --models MODELS [MODELS ...]
                        path to the models
```

2.3 Configuration options

This document describes the available hyperparameters used for training NMT-Keras.

These hyperparameters are set in the config.py script or via command-line-interface.

2.3.1 Naming and experiment setup

- **DATASET_NAME**: Task name. Used for naming and for indexing files.
- **SRC_LAN**: Language of the source text. Used for naming.

- **TRG_LAN**: Language of the target text. Used for naming and for coputing language-dependent metrics (e.g. Meteor)
- **DATA_ROOT_PATH**: Path to the data
- **TEXT_FILES**: Dictionary containing the splits ('train/val/test) and the files corresponding to each one. The source/target languages will be appended to these files.

2.3.2 Input/output

- **INPUTS_IDS_DATASET**: Name of the inputs of the Dataset class.
- **OUTPUTS_IDS_DATASET**: Name of the outputs of the Dataset class.
- **INPUTS_IDS_MODEL**: Name of the inputs of the Model.
- **OUTPUTS_IDS_MODEL**: Name of the outputs of the Model.

2.3.3 Evaluation

- **METRICS**: List of metric used for evaluating the model. The coco package is recommended.
- **EVAL_ON_SETS**: List of splits ('train', 'val', 'test') to evaluate with the metrics from METRICS. Typically: 'val'
- **EVAL_ON_SETS_KERAS**: List of splits ('train', 'val', 'test') to evaluate with the Keras metrics.
- **START_EVAL_ON_EPOCH**: The evaluation starts at this epoch.
- **EVAL_EACH_EPOCHS**: Whether the evaluation frequency units are epochs or updates.
- **EVAL_EACH**: Evaluation frequency.

2.3.4 Decoding

- **SAMPLING**: Decoding mode. Only 'max_likelihood' tested.
- **TEMPERATURE**: Multinomial sampling temerature.
- **BEAM_SEARCH**: Switches on-off the beam search.
- **BEAM_SIZE**: Beam size.
- **OPTIMIZED_SEARCH**: Encode the source only once per sample (recommended).

2.3.5 Search normalization

- **SEARCH_PRUNING**: Apply pruning strategies to the beam search method. It will likely increase decoding speed, but decrease quality.
- **MAXLEN_GIVEN_X**: Generate translations of similar length to the source sentences.
- **MAXLEN_GIVEN_X_FACTOR**: The hypotheses will have (as maximum) the number of words of the source sentence * LENGTH_Y_GIVEN_X_FACTOR.
- **MINLEN_GIVEN_X**: Generate translations of similar length to the source sentences.
- **MINLEN_GIVEN_X_FACTOR**: The hypotheses will have (as minimum) the number of words of the source sentence / LENGTH_Y_GIVEN_X_FACTOR.

- **LENGTH_PENALTY**: Apply length penalty (Wu et al. (2016)).
- **LENGTH_NORM_FACTOR**: Length penalty factor (Wu et al. (2016)).
- **COVERAGE_PENALTY**: Apply source coverage penalty (Wu et al. (2016)).
- **COVERAGE_NORM_FACTOR**: Coverage penalty factor (Wu et al. (2016)).
- **NORMALIZE_SAMPLING**: Alternative (simple) length normalization. Normalize hypotheses scores according to their length.
- **ALPHA_FACTOR**: Normalization according to |h|**ALPHA_FACTOR (Wu et al. (2016)).

2.3.6 Sampling

- **SAMPLE_ON_SETS**: Splits from where we'll sample.
- **N_SAMPLES**: Number of samples generated
- **START_SAMPLING_ON_EPOCH**: First epoch where to start the sampling counter
- **SAMPLE_EACH_UPDATES**: Sampling frequency (always in #updates)

2.3.7 Unknown words treatment

- **POS_UNK**: Enable unknown words replacement strategy.
- **HEURISTIC**: Heuristic followed for replacing unks:
- 0: Replace the UNK by the correspondingly aligned source.
- 1: Replace the UNK by the translation (given by an external dictionary of the aligned source.
- 2: Replace the UNK by the translation (given by an external dictionary of the aligned source only if it starts with a lowercase. Otherwise, copies the source word.
- **ALIGN_FROM_RAW**: Align using the source files or the short-list model vocabulary.
- **MAPPING**: Mapping dictionary path (for heuristics 1 and 2). Obtained with the build_mapping_file script.

2.3.8 Word representation

- **TOKENIZATION_METHOD**: Tokenization applied to the input and output text.
- **DETOKENIZATION_METHOD**: Detokenization applied to the input and output text.
- **APPLY_DETOKENIZATION**: Wheter we apply the detokenization method
- **TOKENIZE_HYPOTHESES**: Whether we tokenize the hypotheses (for computing metrics).
- **TOKENIZE_REFERENCES**: Whether we tokenize the references (for computing metrics).
- **BPE_CODES_PATH**: If *TOKENIZATION_METHOD == 'tokenize_bpe'*, sets the path to the learned BPE codes.

2.3.9 Text representation

- **FILL**: Padding mode: Insert zeroes at the 'start', 'center' or 'end'.
- **PAD_ON_BATCH**: Make batches of a fixed number of timesteps or pad to the maximum length of the minibatch.

2.3.10 Input text

- **INPUT_VOCABULARY_SIZE**: Input vocabulary size. Set to 0 for using all, otherwise it will be truncated to these most frequent words.
- **MIN_OCCURRENCES_INPUT_VOCAB**: Discard all input words with a frequency below this threshold.
- **MAX_INPUT_TEXT_LEN**: Maximum length of the input sentences.

2.3.11 Output text

- **INPUT_VOCABULARY_SIZE**: Output vocabulary size. Set to 0 for using all, otherwise it will be truncated to these most frequent words.
- **MIN_OCCURRENCES_OUTPUT_VOCAB**: Discard all output words with a frequency below this threshold.
- **MAX_INPUT_TEXT_LEN**: Maximum length of the output sentences.
- **MAX_OUTPUT_TEXT_LEN_TEST**: Maximum length of the output sequence during test time.

2.3.12 Optimization

- **LOSS**: Loss function to optimize.
- **CLASSIFIER_ACTIVATION**: Last layer activation function.
- **SAMPLE_WEIGHTS**: Apply a mask to the output sequence. Should be set to True.
- **LR_DECAY**: Reduce the learning rate each this number of epochs. Set to None if don't want to decay the learning rate
- **LR_GAMMA**: Decay rate.
- **LABEL_SMOOTHING**: Epsilon value for label smoothing. Only valid for 'categorical_crossentropy' loss. See [1512.00567](arxiv.org/abs/1512.00567).

Optimizer setup

- **OPTIMIZER**: Optimizer to use. See the available Keras optimizers.
- **LR**: Learning rate.
- **CLIP_C**: During training, clip L2 norm of gradients to this value.
- **CLIP_V**: During training, clip absolute value of gradients to this value.
- **USE_TF_OPTIMIZER**: Use native Tensorflow's optimizer (only for the Tensorflow backend).

Advanced parameters for optimizers

- **MOMENTUM**: Momentum value (for SGD optimizer).
- **NESTEROV_MOMENTUM**: Use Nesterov momentum (for SGD optimizer).
- **RHO**: Rho value (for Adadelta and RMSprop optimizers).
- **BETA_1**: Beta 1 value (for Adam, Adamax Nadam optimizers).
- **BETA_2**: Beta 2 value (for Adam, Adamax Nadam optimizers).
- **EPSILON**: Oprimizers epsilon value.

2.3.13 Learning rate schedule

- **LR_DECAY**: Frequency (number of epochs or updates) between LR annealings. Set to None for not decay the learning rate.
- **LR_GAMMA**: Multiplier used for decreasing the LR.
- **LR_REDUCE_EACH_EPOCHS**: Reduce each LR_DECAY number of epochs or updates.
- **LR_START_REDUCTION_ON_EPOCH**: Epoch to start the reduction.
- **LR_REDUCER_TYPE**: Function to reduce. 'linear' and 'exponential' implemented.
- **LR_REDUCER_EXP_BASE**: Base for the exponential decay.
- **LR_HALF_LIFE**: Factor/warmup steps for exponenital/noam decay.
- **WARMUP_EXP**: Warmup steps for noam decay.

2.3.14 Training options

- **MAX_EPOCH**: Stop when computed this number of epochs.
- **BATCH_SIZE**: Size of each minibatch.
- **HOMOGENEOUS_BATCHES**: If activated, use batches with similar output lengths, in order to better profit parallel computations.
- **JOINT_BATCHES**: When using homogeneous batches, size of the maxibatch.
- **PARALLEL_LOADERS**: Parallel CPU data batch loaders.
- **EPOCHS_FOR_SAVE**: Save model each this number of epochs.
- **WRITE_VALID_SAMPLES**: Write validation samples in file.
- **SAVE_EACH_EVALUATION**: Save the model each time we evaluate.

2.3.15 Early stop

- **EARLY_STOP** = Turns on/off the early stop regularizer.
- **PATIENCE**: We'll stop if we don't improve after this number of evaluations
- **STOP_METRIC**: Stopping metric.

2.3.16 Model main hyperparameters

- **MODEL_TYPE**: Model to train. See the model zoo for the supported architectures.
- **RNN_TYPE**: RNN unit type ('LSTM' and 'GRU' supported).
- **INIT_FUNCTION**: Initialization function for matrices (see keras/initializations).
- **INNER_INIT**: Initialization function for inner RNN matrices.
- **INIT_ATT**: Initialization function for attention mechism matrices.

Source word embeddings

- **SOURCE_TEXT_EMBEDDING_SIZE**: Source language word embedding size.
- **SRC_PRETRAINED_VECTORS**: Path to source pretrained vectors. See the utils folder for preprocessing scripts. Set to None if you don't want to use source pretrained vectors. When using pretrained word embeddings. this parameter must match with the source word embeddings size
- **SRC_PRETRAINED_VECTORS_TRAINABLE**: Finetune or not the target word embedding vectors.
- **SCALE_SOURCE_WORD_EMBEDDINGS**: Scale source word embeddings by Sqrt(SOURCE_TEXT_EMBEDDING_SIZE).

Target word embedding

- **TARGET_TEXT_EMBEDDING_SIZE**: Source language word embedding size.
- **TRG_PRETRAINED_VECTORS**: Path to target pretrained vectors. See the utils folder for preprocessing scripts. Set to None if you don't want to use source pretrained vectors. When using pretrained word embeddings. this parameter must match with the target word embeddings size
- **TRG_PRETRAINED_VECTORS_TRAINABLE**: Finetune or not the target word embedding vectors.
- **SCALE_TARGET_WORD_EMBEDDINGS**: Scale target word embeddings by Sqrt(TARGET_TEXT_EMBEDDING_SIZE).

Deepness

- **N_LAYERS_DECODER**: Stack this number of decoding layers.
- **DEEP_OUTPUT_LAYERS**: Additional Fully-Connected layers applied before softmax.

2.3.17 AttentionRNNEncoderDecoder model

- **ENCODER_RNN_TYPE**: Encoder's RNN unit type ('LSTM' and 'GRU' supported).
- **DECODER_RNN_TYPE**: Decoder's RNN unit type ('LSTM', 'GRU', 'ConditionalLSTM' and 'ConditionalGRU' supported).
- **ATTENTION_MODE**: Attention mode. 'add' (Bahdanau-style) or 'dot' (Luong-style).

Encoder configuration

- **ENCODER_HIDDEN_SIZE**: Encoder RNN size.
- **BIDIRECTIONAL_ENCODER**: Use a bidirectional encoder.
- **BIDIRECTIONAL_DEEP_ENCODER**: Use bidirectional encoder in all stacked encoding layers

Decoder configuration

- **DECODER_HIDDEN_SIZE**: Decoder RNN size.
- **ADDITIONAL_OUTPUT_MERGE_MODE**: Merge mode for the deep output layer.
- **SKIP_VECTORS_HIDDEN_SIZE**: Deep output layer size
- **INIT_LAYERS**: Initialize the first decoder state with these layers (from the encoder).
- **SKIP_VECTORS_SHARED_ACTIVATION**: Activation for the skip vectors.

2.3.18 Transformer model

- **MODEL_SIZE**: Transformer model size (dmodel in de paper).
- **MULTIHEAD_ATTENTION_ACTIVATION**: Activation the input projections in the Multi-Head Attention blocks.
- **FF_SIZE**: Size of the feed-forward layers of the Transformer model.
- **N_HEADS**: Number of parallel attention layers of the Transformer model.

2.3.19 Regularizers

Regularization functions

- **REGULARIZATION_FN**: Regularization function. 'L1', 'L2' and 'L1_L2' supported.
- **WEIGHT_DECAY**: L2 regularization in non-recurrent layers.
- **RECURRENT_WEIGHT_DECAY**: L2 regularization in recurrent layers
- **DOUBLE_STOCHASTIC_ATTENTION_REG**: Doubly stochastic attention (Eq. 14 from arXiv:1502.03044).

Dropout

- **DROPOUT_P**: Percentage of units to drop in non-recurrent layers (0 means no dropout).
- **RECURRENT_DROPOUT_P**: Percentage of units to drop in recurrent layers(0 means no dropout).
- **ATTENTION_DROPOUT_P**: Percentage of units to drop in attention layers (0 means no dropout).

Gaussian noise

- **USE_NOISE**: Apply gaussian noise during training.
- **NOISE_AMOUNT**: Amount of noise.

Batch normalization

- **USE_BATCH_NORMALIZATION**: Batch normalization regularization in non-recurrent layers and recurrent inputs. If True it is recommended to deactivate Dropout.
- **BATCH_NORMALIZATION_MODE**: Sample-wise or feature-wise BN mode.

Additional normalization layers

- **USE_PRELU**: Apply use PReLU activations as regularizer.
- **USE_L1**: L1 normalization on the features.
- **USE_L2**: Apply L2 function on the features.

2.3.20 Tensorboard

- **TENSORBOARD**: Switches On/Off the tensorboard callback.
- **LOG_DIR**: irectory to store teh model. Will be created inside STORE_PATH.
- **EMBEDDINGS_FREQ**: Frequency (in epochs) at which selected embedding layers will be saved.
- **EMBEDDINGS_LAYER_NAMES**: A list of names of layers to keep eye on. If None or empty list all the embedding layer will be watched.
- **EMBEDDINGS_METADATA**: Dictionary which maps layer name to a file name in which metadata for this embedding layer is saved.
- **LABEL_WORD_EMBEDDINGS_WITH_VOCAB**: Whether to use vocabularies as word embeddings labels (will overwrite EMBEDDINGS_METADATA).
- **WORD_EMBEDDINGS_LABELS**: Vocabularies for labeling. Must match EMBEDDINGS_LAYER_NAMES.

2.3.21 Storage and plotting

- **MODEL_NAME**: Name for the model.
- **EXTRA_NAME**: MODEL_NAME suffix
- **STORE_PATH**: Models and evaluation results will be stored here.
- **DATASET_STORE_PATH**: Dataset instance will be stored here.
- **SAMPLING_SAVE_MODE**: Save evaluation outputs in this format. Set to 'list' for a raw file.
- **VERBOSE**: Verbosity level.
- **RELOAD**: Reload a stored model. If 0 start training from scratch, otherwise use the model from this epoch/update.
- **REBUILD_DATASET**: Build dataset again or use a stored instance.
- **MODE**: 'training' or 'sampling' (if 'sampling' then RELOAD must be greater than 0 and EVAL_ON_SETS will be used). For 'sampling' mode, is recommended to use the sample_ensemble script.

2.4 Resources

2.4.1 Theoretical NMT

Before using an NMT, you should read and understand the theoretical basis of attentional NMT systems.

2.4.2 NMT-Keras Step-by-step

- NMT-Keras step-by-step guide (iPython and html versions): Tutorials for running this library. They are expected to be followed in order:
  1. Dataset setup: Shows how to invoke and configure a Dataset instance for a translation problem.
  2. Training tutorial: Shows how to call a translation model, link it with the dataset object and build callbacks for monitorizing the training.
  3. Decoding tutorial: Shows how to call a trained translation model and use it to translate new text.
  4. NMT model tutorial: Shows how to build a state-of-the-art NMT model with Keras in few (~50) lines.

2.4.3 NMT-Keras Output

This is a brief explanation about the typical output produced by the training pipeline of NMT-Keras. Assuming that we launched NMT-Keras for the example from tutorials, we'll have the following tree of folders (after 1 epoch):

```
├── trained_models
│   ├── EuTrans_GroundHogModel_src_emb_420_bidir_True_enc_600_dec_600_deepout_maxout_
↪trg_emb_420_Adam_0.001
│   │   ├── config.pkl
│   │   ├── epoch_1_Model_Wrapper.pkl
│   │   ├── epoch_1_structure_init.json
│   │   ├── epoch_1_structure.json
│   │   ├── epoch_1_structure_next.json
│   │   ├── epoch_1_weights.h5
│   │   ├── epoch_1_weights_init.h5
│   │   ├── epoch_1_weights_next.h5
│   │   ├── val.coco
│   │   ├── val_epoch_1.pred
```

Let's have a look to these files.

- *config.pkl*: Pickle containing the training parameters.
- *epoch_1_Model_Wrapper.pkl*: Pickle containing the Model_Wrapper object that we have trained.
- *epoch_1_structure.json*: Keras json specifying the layer and connections of the model.
- *epoch_1_structure_init.json*: Keras json specifying the layer and connections of the model_init (see tutorial 4 for more info about the model).
- *epoch_1_structure_next.json*: Keras json specifying the layer and connections of the model_next (see tutorial 4 for more info about the model).
- *epoch_1_weights.h5*: Model parameters (weight matrices).
- *epoch_1_weights_init.h5*: Model init parameters (weight matrices).
- *epoch_1_weights_next.h5*: Model next parameters (weight matrices).

- *val.coco*: Metrics dump. This file is name as [tested_split].[metrics_name]. It contains a header with the metrics name and the value of all evaluations (epoch/updates). For instance:

  epoch,Bleu_1, Bleu_2, Bleu_3, Bleu_4, CIDEr, METEOR, ROUGE_L, 1,0.906982874122, 0.875873151361, 0.850383597611, 0.824070996966, 8.084477458, 0.550547408997, 0.931523374569, 2,0.932937494321, 0.90923787501, 0.889965151506, 0.871819102335, 8.53565391657, 0.586377788443, 0.947634196936, 3,0.965579088172, 0.947927460597, 0.934090548706, 0.920166838768, 9.0864109399, 0.63234570058, 0.971618921459,

- *val_epoch_1.pred*: Raw file with the output of the NMT system at the evaluation performed at the end of epoch 1.

We can modify the save and evaluation frequencies from the config file.

2.4.4 Tensorboard integration

TensorBoard is a visualization tool provided with TensorFlow.

It can be accessed by NMT-Keras and provide visualization of the learning process, dynamic graphs of our training and metrics, as well representation of different layers (such as word embeddings). Of course, this tool is only available with the Tensorflow backend.

In this document, we'll set some parameters and explore some of the options that Tensorboard provides. We'll:

- Configure Tensorboard and NMT-Keras.
- Visualize the learning process (loss curves).
- Visualize the computation graphs built by NMT Keras.
- Visualize the words embeddings obtained during the training stage.

In the [configuration file] we have available the following tensorboard-related options:

```
TENSORBOARD = True                          # Switches On/Off the tensorboard callback
LOG_DIR = 'tensorboard_logs'                # Directory to store teh model. Will be␣
↪created inside STORE_PATH
EMBEDDINGS_FREQ = 1                         # Frequency (in epochs) at which selected␣
↪embedding layers will be saved.
EMBEDDINGS_LAYER_NAMES = [                  # A list of names of layers to keep eye on.␣
↪If None or empty list all the embedding layer will be watched.
'source_word_embedding',
'target_word_embedding']
EMBEDDINGS_METADATA = None                  # Dictionary which maps layer name to a file␣
↪name in which metadata for this embedding layer is saved.
LABEL_WORD_EMBEDDINGS_WITH_VOCAB = True   # Whether to use vocabularies as word␣
↪embeddings labels (will overwrite EMBEDDINGS_METADATA)
WORD_EMBEDDINGS_LABELS = [                  # Vocabularies for labeling. Must match␣
↪EMBEDDINGS_LAYER_NAMES
                            'source_text',
                            'target_text']
```

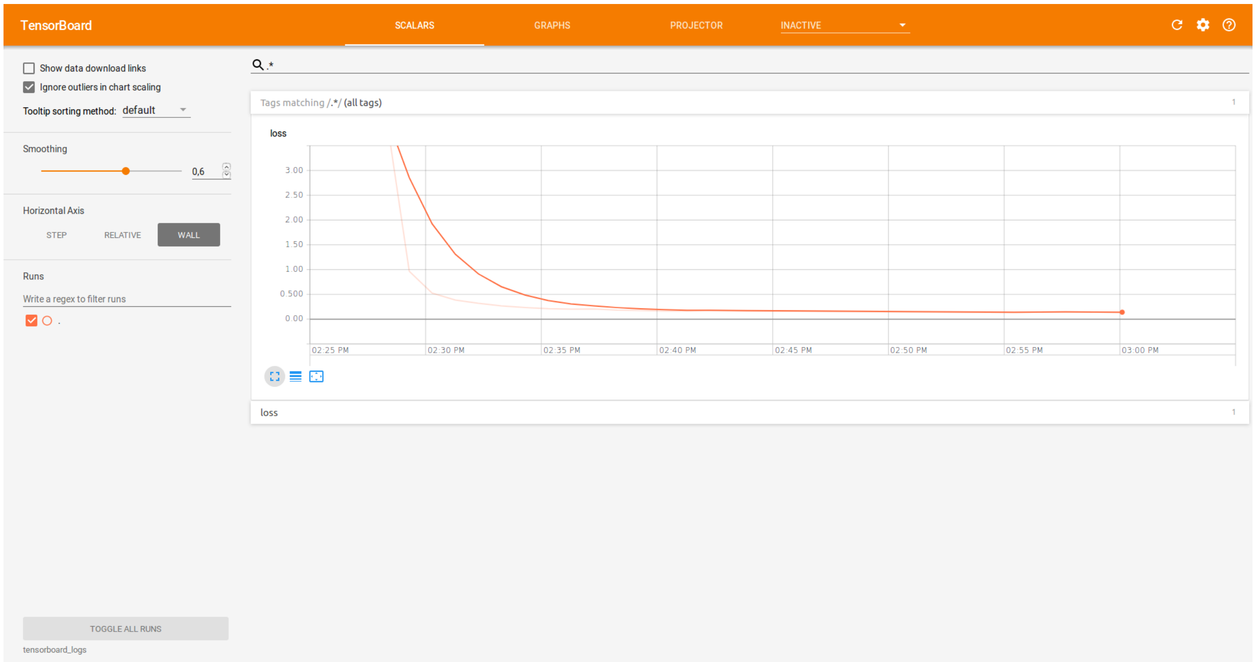
With these options, we are telling Tensorboard where to store the data we want to visualize: loss curve, computation graph and word embeddings. Moreover, we are specifying the word embedding layers that we want to visualize. By setting the *WORD_EMBEDDINGS_LABELS* to the corresponding *Dataset* ids, we can print labels in the word embedding visualization.

Now, we run a regular training: *python main.py*. If we *cd* to the model directory, we'll see a directiory named *tensorboard_logs*. Now, we launch Tensorboard on this directory:

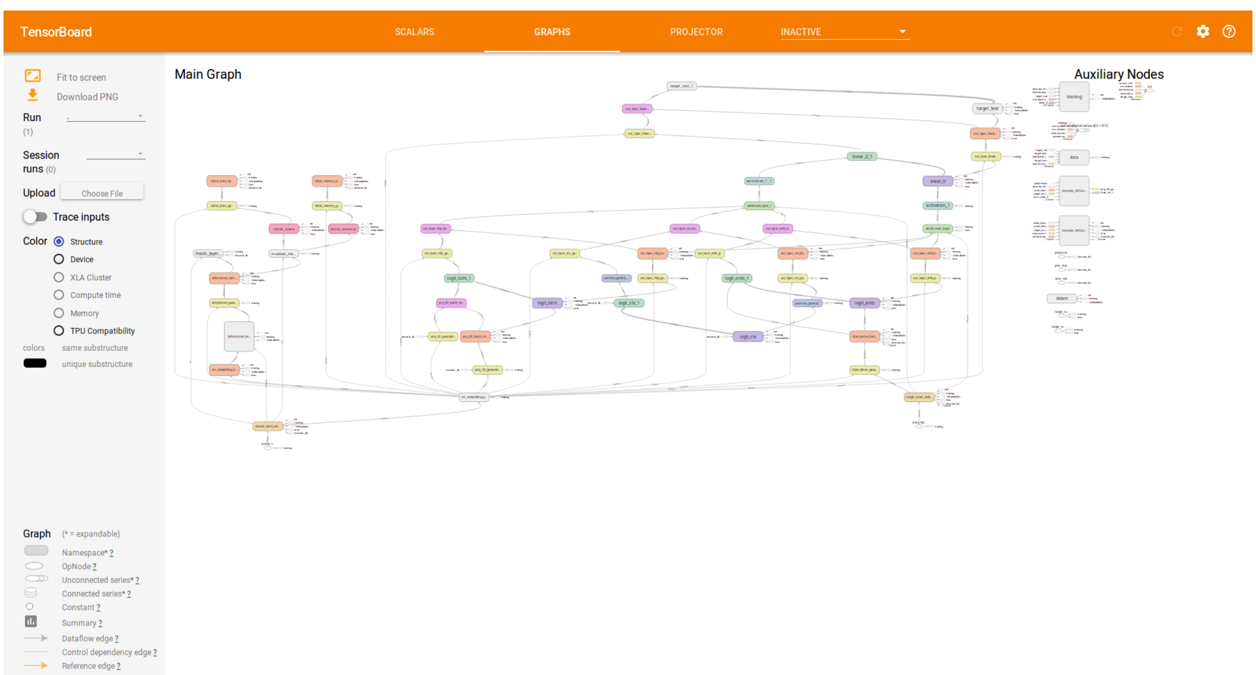
```
$ tensorboard --logdir=tensorboard_logs
TensorBoard 0.1.5 at http://localhost:6006 (Press CTRL+C to quit)
```

We can open Tensorboard in our browser (http://localhost:6006) with the NMT-Keras information:

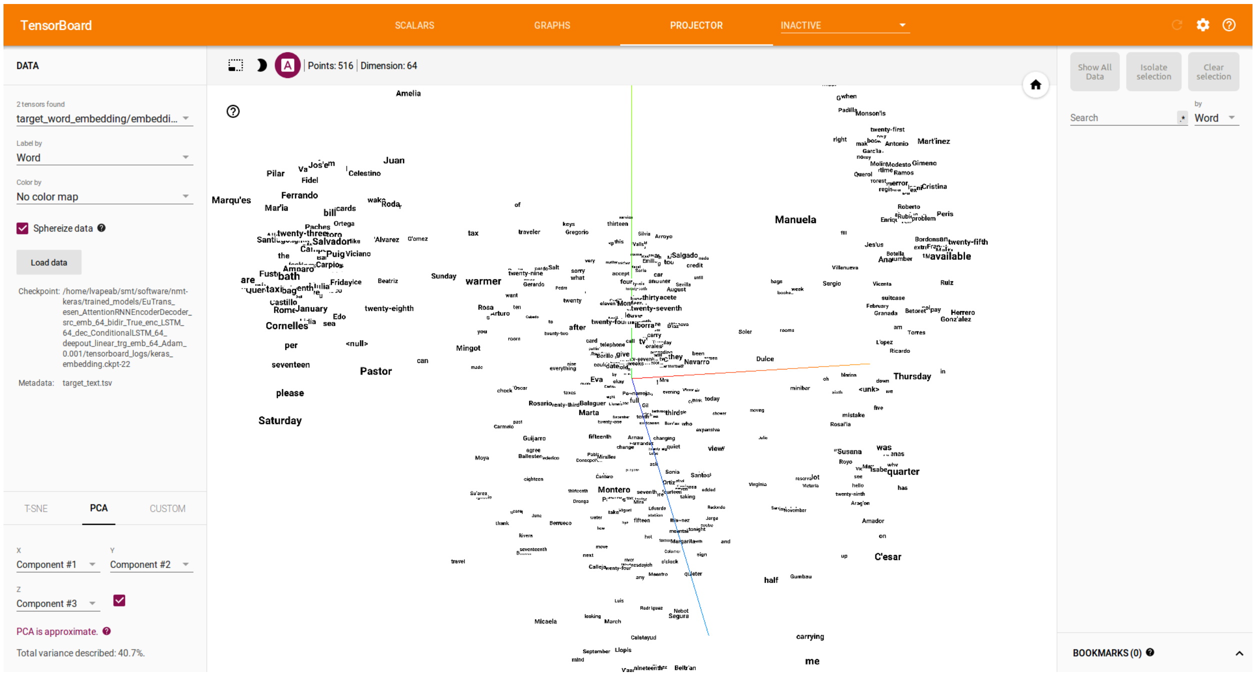
Loss curve



Model graphs



Embedding visualization



2.5 Tutorials

This page contains some examples and tutorials showing how the library works. All tutorials have an iPython notebook version.

Almost every variable tutorials representing model hyperparameters have been intentionally hardcoded in the tutorials, aiming to facilitate readability. On a real execution, these values are taken from the *config.py* file.

All tutorials have been executed from the root *nmt-keras* folder. These tutorials basically are a split version of the execution pipeline of the library. If you run *python main.py*, you'll execute almost the same as tutorials 1, 2 and 4.

The translation task is *EuTrans* (Amengual et al.), a toy-task mainly used for debugging purposes.

2.5.1 Dataset tutorial

First, we'll create a Dataset instance, in order to properly manage the data. First, we are creating a Dataset object (from the Multimodal Keras Wrapper library). Let's make some imports and create an empty Dataset instance:

```
from keras_wrapper.dataset import Dataset, saveDataset
from data_engine.prepare_data import keep_n_captions
ds = Dataset('tutorial_dataset', 'tutorial', silence=False)
```

Now that we have the empty Dataset, we must indicate its inputs and outputs. In our case, we'll have two different inputs and one single output:

1. **Outputs:: target_text**: Sentences in the target language.
2. **Inputs:: source_text**: Sentences in the source language.

   state_below: Sentences in the target language, but shifted one position to the right (for teacher-forcing training of the model).

For setting up the outputs, we use the setOutputs function, with the appropriate parameters. Note that, when we are building the dataset for the training split, we build the vocabulary (up to 30000 words):

```
ds.setOutput('examples/EuTrans/training.en',
             'train',
             type='text',
             id='target_text',
             tokenization='tokenize_none',
             build_vocabulary=True,
             pad_on_batch=True,
             sample_weights=True,
             max_text_len=30,
             max_words=30000,
             min_occ=0)

ds.setOutput('examples/EuTrans/dev.en',
             'val',
             type='text',
             id='target_text',
             pad_on_batch=True,
             tokenization='tokenize_none',
             sample_weights=True,
             max_text_len=30,
             max_words=0)
```

Similarly, we introduce the source text data, with the setInputs function. Again, when building the training split, we must construct the vocabulary:

```
ds.setInput('examples/EuTrans/training.es',
            'train',
            type='text',
            id='source_text',
            pad_on_batch=True,
            tokenization='tokenize_none',
            build_vocabulary=True,
            fill='end',
            max_text_len=30,
            max_words=30000,
            min_occ=0)
ds.setInput('examples/EuTrans/dev.es',
            'val',
            type='text',
            id='source_text',
            pad_on_batch=True,
            tokenization='tokenize_none',
            fill='end',
            max_text_len=30,
            min_occ=0)
```

…and for the *state_below* data. Note that: 1) The offset flat is set to 1, which means that the text will be shifted to the right 1 position. 2) During sampling time, we won't have this input. Hence, we 'hack' the dataset model by inserting an artificial input, of type 'ghost' for the validation split:

```
ds.setInput('examples/EuTrans/training.en',
            'train',
            type='text',
            id='state_below',
            required=False,
```

(continues on next page)

(continued from previous page)

```
                tokenization='tokenize_none',
                pad_on_batch=True,
                build_vocabulary='target_text',
                offset=1,
                fill='end',
                max_text_len=30,
                max_words=30000)
ds.setInput(None,
                'val',
                type='ghost',
                id='state_below',
                required=False)
```

Next, we match the references with the inputs, in order to evaluate against the raw references:

```
keep_n_captions(ds, repeat=1, n=1, set_names=['val'])
```

Finally, we can save our dataset instance for using it in other experiments:

```
saveDataset(ds, 'datasets')
```

2.5.2 Training tutorial

Now, we'll create and train a Neural Machine Translation (NMT) model. We'll build the so-called *GroundHogModel*. It is defined at the *model_zoo.py* file. If you followed prior tutorial, you should have a dataset instance. Otherwise, you should follow that notebook first.

So, let's go! First, we make some imports, load the default parameters and the dataset:

```
from config import load_parameters
from model_zoo import TranslationModel
import utils
from keras_wrapper.cnn_model import loadModel
from keras_wrapper.dataset import loadDataset
params = load_parameters()
dataset = loadDataset('datasets/Dataset_tutorial_dataset.pkl')
```

Since the number of words in the dataset may be unknown beforehand, we must update the params information according to the dataset instance:

```
params['INPUT_VOCABULARY_SIZE'] = dataset.vocabulary_len['source_text']
params['OUTPUT_VOCABULARY_SIZE'] = dataset.vocabulary_len['target_text']
```

Now, we create a *TranslationModel* object: An instance of a Model_Wrapper object from Multimodal Keras Wrapper. We specify the type of the model (*GroundHogModel*) and the vocabularies from the Dataset:

```
nmt_model = TranslationModel(params,
                             model_type='GroundHogModel',
                             model_name='tutorial_model',
                             vocabularies=dataset.vocabulary,
                             store_path='trained_models/tutorial_model/',
                             verbose=True)
```

Now, we must define the inputs and outputs mapping from our Dataset instance to our model:

```
inputMapping = dict()
for i, id_in in enumerate(params['INPUTS_IDS_DATASET']):
    pos_source = dataset.ids_inputs.index(id_in)
    id_dest = nmt_model.ids_inputs[i]
    inputMapping[id_dest] = pos_source
nmt_model.setInputsMapping(inputMapping)

outputMapping = dict()
for i, id_out in enumerate(params['OUTPUTS_IDS_DATASET']):
    pos_target = dataset.ids_outputs.index(id_out)
    id_dest = nmt_model.ids_outputs[i]
    outputMapping[id_dest] = pos_target
nmt_model.setOutputsMapping(outputMapping)
```

We can add some callbacks for controlling the training (e.g. Sampling each N updates, early stop, learning rate annealing…). For instance, let's build a *PrintPerformanceMetricOnEpochEndOrEachNUpdates* callback. Each 2 epochs, it will compute the 'coco' scores on the development set. We need to pass some variables to the callback (in the extra_vars dictionary):

```
from keras_wrapper.extra.callbacks import *
extra_vars = {'language': 'en',
              'n_parallel_loaders': 8,
              'tokenize_f': eval('dataset.' + 'tokenize_none'),
              'beam_size': 12,
              'maxlen': 50,
              'model_inputs': ['source_text', 'state_below'],
              'model_outputs': ['target_text'],
              'dataset_inputs': ['source_text', 'state_below'],
              'dataset_outputs': ['target_text'],
              'normalize': True,
              'alpha_factor': 0.6,
              'val':{'references': dataset.extra_variables['val']['target_text']}
              }
vocab = dataset.vocabulary['target_text']['idx2words']
callbacks = []
callbacks.append(PrintPerformanceMetricOnEpochEnd(nmt_model,
                                                  dataset,
                                                  gt_id='target_text',
                                                  metric_name=['coco'],
                                                  set_name=['val'],
                                                  batch_size=50,
                                                  each_n_epochs=2,
                                                  extra_vars=extra_vars,
                                                  reload_epoch=0,
                                                  is_text=True,
                                                  index2word_y=vocab,
                                                  sampling_type='max_likelihood',
                                                  beam_search=True,
                                                  save_path=nmt_model.model_path,
                                                  start_eval_on_epoch=0,
                                                  write_samples=True,
                                                  write_type='list',
                                                  save_each_evaluation=True,
                                                  verbose=True))
```

Now we are almost ready to train. We set up some training parameters…:

```
training_params = {'n_epochs': 100,
                   'batch_size': 40,
                   'maxlen': 30,
                   'epochs_for_save': 1,
                   'verbose': 0,
                   'eval_on_sets': [],
                   'n_parallel_loaders': 8,
                   'extra_callbacks': callbacks,
                   'reload_epoch': 0,
                   'epoch_offset': 0}
```

And train!:

```
nmt_model.trainNet(dataset, training_params)
```

For a description of the training output, refer to the typical output document.

2.5.3 Decoding tutorial

Now, we'll load from disk a trained Neural Machine Translation (NMT) model and we'll apply it for translating new text. This is done by the sample_ensemble script.

This tutorial assumes that you followed both previous tutorials. In this case, we want to translate the 'test' split of our dataset.

As before, let's import some stuff and load the dataset instance:

```
from config import load_parameters
from data_engine.prepare_data import keep_n_captions
from keras_wrapper.cnn_model import loadModel
from keras_wrapper.dataset import loadDataset
params = load_parameters()
dataset = loadDataset('datasets/Dataset_tutorial_dataset.pkl')
```

Since we want to translate a new data split ('test') we must add it to the dataset instance, just as we did before (at the first tutorial). In case we also had the refences of the test split and we wanted to evaluate it, we can add it to the dataset. Note that this is not mandatory and we could just predict without evaluating.:

```
dataset.setInput('examples/EuTrans/test.es',
            'test',
            type='text',
            id='source_text',
            pad_on_batch=True,
            tokenization='tokenize_none',
            fill='end',
            max_text_len=100,
            min_occ=0)

dataset.setInput(None,
'test',
type='ghost',
id='state_below',
required=False)
```

Now, let's load the translation model. Suppose we want to load the model saved at the end of the epoch 4:

```
params['INPUT_VOCABULARY_SIZE'] = dataset.vocabulary_len[params['INPUTS_IDS_DATASET
↪'][0]]
params['OUTPUT_VOCABULARY_SIZE'] = dataset.vocabulary_len[params['OUTPUTS_IDS_DATASET
↪'][0]]
# Load model
nmt_model = loadModel('trained_models/tutorial_model', 4)
```

Once we loaded the model, we just have to invoke the sampling method (in this case, the Beam Search algorithm) for the 'test' split:

```
params_prediction = {'max_batch_size': 50,
                     'n_parallel_loaders': 8,
                     'predict_on_sets': ['test'],
                     'beam_size': 12,
                     'maxlen': 50,
                     'model_inputs': ['source_text', 'state_below'],
                     'model_outputs': ['target_text'],
                     'dataset_inputs': ['source_text', 'state_below'],
                     'dataset_outputs': ['target_text'],
                     'normalize': True,
                     'alpha_factor': 0.6
                     }
predictions = nmt_model.predictBeamSearchNet(dataset, params_prediction)['test']
```

Up to this moment, in the variable 'predictions', we have the indices of the words of the hypotheses. We must decode them into words. For doing this, we'll use the dictionary stored in the dataset object:

```
from keras_wrapper.utils import decode_predictions_beam_search
vocab = dataset.vocabulary['target_text']['idx2words']
predictions = decode_predictions_beam_search(predictions,
                                             vocab,
                                             verbose=params['VERBOSE'])
```

Finally, we store the system hypotheses:

```
filepath = nmt_model.model_path+'/' + 'test' + '_sampling.pred'  # results file
from keras_wrapper.extra.read_write import list2file
list2file(filepath, predictions)
```

If we have the references of this split, we can also evaluate the performance of our system on it. First, we must add them to the dataset object:

```
# In case we had the references of this split, we could also load the split and␣
↪evaluate on it
dataset.setOutput('examples/EuTrans/test.en',
             'test',
             type='text',
             id='target_text',
             pad_on_batch=True,
             tokenization='tokenize_none',
             sample_weights=True,
             max_text_len=30,
             max_words=0)
keep_n_captions(dataset, repeat=1, n=1, set_names=['test'])
```

Next, we call the evaluation system: The Coco-caption package. Although its main usage is for multimodal captioning, we can use it in machine translation:

```
from keras_wrapper.extra import evaluation
metric = 'coco'
# Apply sampling
extra_vars = dict()
extra_vars['tokenize_f'] = eval('dataset.' + 'tokenize_none')
extra_vars['language'] = params['TRG_LAN']
extra_vars['test'] = dict()
extra_vars['test']['references'] = dataset.extra_variables['test']['target_text']
metrics = evaluation.select[metric](pred_list=predictions,
                                    verbose=1,
                                    extra_vars=extra_vars,
                                    split='test')
```

2.5.4 NMT model tutorial

In this module, we are going to create an encoder-decoder model with:

- A bidirectional GRU encoder and a GRU decoder
- An attention model
- The previously generated word feeds back de decoder
- MLPs for initializing the initial RNN state
- Skip connections from inputs to outputs
- Beam search.

As usual, first we import the necessary stuff:

```
from keras.layers import *
from keras.models import model_from_json, Model
from keras.optimizers import Adam, RMSprop, Nadam, Adadelta, SGD, Adagrad, Adamax
from keras.regularizers import l2
from keras_wrapper.cnn_model import Model_Wrapper
from keras_wrapper.extra.regularize import Regularize
```

And define the dimesnions of our model. For instance, a word embedding size of 50 and 100 units in RNNs. The inputs/outpus are defined as in previous tutorials.:

```
ids_inputs = ['source_text', 'state_below']
ids_outputs = ['target_text']
word_embedding_size = 50
hidden_state_size = 100
input_vocabulary_size=686  # Autoset in the library
output_vocabulary_size=513  # Autoset in the library
```

Now, let's define our encoder. First, we have to create an Input layer to connect the input text to our model. Next, we'll apply a word embedding to the sequence of input indices. This word embedding will feed a Bidirectional GRU network, which will produce our sequence of annotations:

```
# 1. Source text input
src_text = Input(name=ids_inputs[0],
                 batch_shape=tuple([None, None]), # Since the input sequences have␣
↪variable-length, we do not retrict the Input shape
                 dtype='int32')
# 2. Encoder
```

(continues on next page)

(continued from previous page)

```
# 2.1. Source word embedding
src_embedding = Embedding(input_vocabulary_size, word_embedding_size,
                          name='source_word_embedding', mask_zero=True # Zeroes as␣
↪mask
                          )(src_text)
# 2.2. BRNN encoder (GRU/LSTM)
annotations = Bidirectional(GRU(hidden_state_size,
                                return_sequences=True  # Return the full sequence
                                ),
                            name='bidirectional_encoder',
                            merge_mode='concat')(src_embedding)
```

Once we have built the encoder, let's build our decoder. First, we have an additional input: The previously generated word (the so-called state_below). We introduce it by means of an Input layer and a (target language) word embedding:

```
# 3. Decoder
# 3.1.1. Previously generated words as inputs for training -> Teacher forcing
next_words = Input(name=ids_inputs[1], batch_shape=tuple([None, None]), dtype='int32')
# 3.1.2. Target word embedding
state_below = Embedding(output_vocabulary_size, word_embedding_size,
                        name='target_word_embedding',
                        mask_zero=True)(next_words)
```

The initial hidden state of the decoder's GRU is initialized by means of a MLP (in this case, single-layered) from the average of the annotations. We also aplly the mask to the annotations:

```
ctx_mean = MaskedMean()(annotations)
annotations = MaskLayer()(annotations)  # We may want the padded annotations
initial_state = Dense(hidden_state_size, name='initial_state',
                      activation='tanh')(ctx_mean)
```

So, we have the input of our decoder:

```
input_attentional_decoder = [state_below, annotations, initial_state]
```

Note that, for a sample, the sequence of annotations and initial state is the same, independently of the decoding time-step. In order to avoid computation time, we build two models, one for training and the other one for sampling. They will share weights, but the sampling model will be made up of two different models. One (model_init) will compute the sequence of annotations and initial_state. The other model (model_next) will compute a single recurrent step, given the sequence of annotations, the previous hidden state and the generated words up to this moment.

Therefore, now we slightly change the form of declaring layers. We must share layers between the decoding models.

So, let's start by building the attentional-conditional GRU:

```
# Define the AttGRUCond function
sharedAttGRUCond = AttGRUCond(hidden_state_size,
                              return_sequences=True,
                              return_extra_variables=True, # Return attended input␣
↪and attenton weights
                              return_states=True # Returns the sequence of hidden␣
↪states (see discussion above)
                              )
[proj_h, x_att, alphas, h_state] = sharedAttGRUCond(input_attentional_decoder) #␣
↪Apply shared_AttnGRUCond to our input
```

Now, we set skip connections between input and output layer. Note that, since we have a temporal dimension because of the RNN decoder, we must apply the layers in a TimeDistributed way. Finally, we will merge all skip-connections and apply a ‘tanh’ no-linearlity:

```
# Define layer function
shared_FC_mlp = TimeDistributed(Dense(word_embedding_size, activation='linear',),
                                name='logit_lstm')
# Apply layer function
out_layer_mlp = shared_FC_mlp(proj_h)

# Define layer function
shared_FC_ctx = TimeDistributed(Dense(word_embedding_size, activation='linear'),
                                name='logit_ctx')
# Apply layer function
out_layer_ctx = shared_FC_ctx(x_att)
shared_Lambda_Permute = PermuteGeneral((1, 0, 2))
out_layer_ctx = shared_Lambda_Permute(out_layer_ctx)

# Define layer function
shared_FC_emb = TimeDistributed(Dense(word_embedding_size, activation='linear'),
                                name='logit_emb')
# Apply layer function
out_layer_emb = shared_FC_emb(state_below)

additional_output = merge([out_layer_mlp, out_layer_ctx, out_layer_emb], mode='sum',
↪name='additional_input')
shared_activation_tanh = Activation('tanh')
out_layer = shared_activation_tanh(additional_output)
```

Now, we’ll’ apply a deep output layer, with Maxout activation:

```
shared_maxout = TimeDistributed(MaxoutDense(word_embedding_size), name='maxout_layer')
out_layer = shared_maxout(out_layer)
```

Finally, we apply a softmax function for obtaining a probability distribution over the target vocabulary words at each timestep:

```
shared_FC_soft = TimeDistributed(Dense(output_vocabulary_size,
                                               activation='softmax',
                                               name='softmax_layer'),
                                         name=ids_outputs[0])
softout = shared_FC_soft(out_layer)
```

That’s all! We built a NMT Model!

NMT models for decoding

Now, let’s build the models required for sampling. Recall that we are building two models, one for encoding the inputs and the other one for advancing steps in the decoding stage.

Let’s start with model_init. It will take the usual inputs (src_text and state_below) and will output:

1. The vector probabilities (for timestep 1).
2. The sequence of annotations (from encoder).
3. The current decoder’s hidden state.

The only restriction here is that the first output must be the output layer (probabilities) of the model.:

```
model_init = Model(input=[src_text, next_words], output=[softout, annotations, h_
↪state])
# Store inputs and outputs names for model_init
ids_inputs_init = ids_inputs

# first output must be the output probs.
ids_outputs_init = ids_outputs + ['preprocessed_input', 'next_state']
```

Next, we will be the model_next. It will have the following inputs:

- Preprocessed input
- Previously generated word
- Previous hidden state

And the following outputs:

- Model probabilities
- Current hidden state

First, we define the inputs:

```
preprocessed_size = hidden_state_size*2 # Because we have a bidirectional encoder
preprocessed_annotations = Input(name='preprocessed_input', shape=tuple([None,␣
↪preprocessed_size]))
prev_h_state = Input(name='prev_state', shape=tuple([hidden_state_size]))
input_attentional_decoder = [state_below, preprocessed_annotations, prev_h_state]
```

And now, we build the model, using the functions stored in the 'shared*' variables declared before:

```
# Apply decoder
[proj_h, x_att, alphas, h_state] = sharedAttGRUCond(input_attentional_decoder)
out_layer_mlp = shared_FC_mlp(proj_h)
out_layer_ctx = shared_FC_ctx(x_att)
out_layer_ctx = shared_Lambda_Permute(out_layer_ctx)
out_layer_emb = shared_FC_emb(state_below)
additional_output = merge([out_layer_mlp, out_layer_ctx, out_layer_emb], mode='sum',␣
↪name='additional_input')
out_layer = shared_activation_tanh(additional_output)
out_layer = shared_maxout(out_layer)
softout = shared_FC_soft(out_layer)
model_next = Model(input=[next_words, preprocessed_annotations, prev_h_state],
                   output=[softout, preprocessed_annotations, h_state])
```

Finally, we store inputs/outputs for model_next. In addition, we create a couple of dictionaries, matching inputs/outputs from the different models (model_init->model_next, model_nex->model_next):

```
# Store inputs and outputs names for model_next
# first input must be previous word
ids_inputs_next = [ids_inputs[1]] + ['preprocessed_input', 'prev_state']
# first output must be the output probs.
ids_outputs_next = ids_outputs + ['preprocessed_input', 'next_state']

# Input -> Output matchings from model_init to model_next and from model_next to␣
↪model_nextxt
matchings_init_to_next = {'preprocessed_input': 'preprocessed_input', 'next_state':
↪'prev_state'}
matchings_next_to_next = {'preprocessed_input': 'preprocessed_input', 'next_state':
↪'prev_state'}
```

(continues on next page)

(continued from previous page)

And that's all! For using this model together with the facilities provided by the staged_model_wrapper library, we should declare the model as a method of a Model_Wrapper class. A complete example of this with additional features can be found at model_zoo.py.

2.6 Modules

2.6.1 nmt_keras package

Submodules

model_zoo

training

apply_model

`nmt_keras.apply_model.`**`sample_ensemble`**(*args*, *params*)

Use several translation models for obtaining predictions from a source text file.

Parameters

- **args** (*argparse.Namespace*) – Arguments given to the method:
  - dataset: Dataset instance with data.
  - text: Text file with source sentences.
  - splits: Splits to sample. Should be already included in the dataset object.
  - dest: Output file to save scores.
  - weights: Weight given to each model in the ensemble. You should provide the same number of weights than models. By default, it applies the same weight to each model (1/N).
  - n_best: Write n-best list (n = beam size).
  - config: Config .pkl for loading the model configuration. If not specified, hyperparameters are read from config.py.
  - models: Path to the models.
  - verbose: Be verbose or not.
- **params** – parameters of the translation model.

`nmt_keras.apply_model.`**`score_corpus`**(*args*, *params*)

Use one or several translation models for scoring source–target pairs-

Parameters

- **args** (*argparse.Namespace*) – Arguments given to the method:
  - dataset: Dataset instance with data.
  - source: Text file with source sentences.
  - target: Text file with target sentences.

  - splits: Splits to sample. Should be already included in the dataset object.
  - dest: Output file to save scores.
  - weights: Weight given to each model in the ensemble. You should provide the same number of weights than models. By default, it applies the same weight to each model (1/N).
  - verbose: Be verbose or not.
  - config: Config .pkl for loading the model configuration. If not specified, hyperparameters are read from config.py.
  - models: Path to the models.
- **params** (*dict*) – parameters of the translation model.

build_callbacks

Module contents

2.6.2 data_engine package

Submodules

prepare_data module

`data_engine.prepare_data.`**`build_dataset`**(*params*)

Builds (or loads) a Dataset instance. :param params: Parameters specifying Dataset options :return: Dataset object

`data_engine.prepare_data.`**`keep_n_captions`**(*ds*, *repeat*, *n=1*, *set_names=None*)

Keeps only n captions per image and stores the rest in dictionaries for a later evaluation :param ds: Dataset object :param repeat: Number of input samples per output :param n: Number of outputs to keep. :param set_names: Set name. :return:

`data_engine.prepare_data.`**`update_dataset_from_file`**(*ds*, *input_text_filename*, *params*, *splits=None*, *output_text_filename=None*, *remove_outputs=False*, *compute_state_below=False*, *recompute_references=False*)

Updates the dataset instance from a text file according to the given params. Used for sampling

Parameters

- **ds** – Dataset instance
- **input_text_filename** – Source language sentences
- **params** – Parameters for building the dataset
- **splits** – Splits to sample
- **output_text_filename** – Target language sentences
- **remove_outputs** – Remove outputs from dataset (if True, will ignore the output_text_filename parameter)
- **compute_state_below** – Compute state below input (shifted target text for professor teaching)

- **recompute_references** – Whether we should rebuild the references of the dataset or not.

Returns Dataset object with the processed data

Module contents

2.6.3 utils package

Submodules

evaluate_from_file module

`utils.evaluate_from_file.CocoScore(ref, hyp, metrics_list=None, language='en')`

Obtains the COCO scores from the references and hypotheses.

Parameters

- **ref** – Dictionary of reference sentences (id, sentence)
- **hyp** – Dictionary of hypothesis sentences (id, sentence)
- **metrics_list** – List of metrics to evaluate on
- **language** – Language of the sentences (for METEOR)

Returns dictionary of scores

`utils.evaluate_from_file.evaluate_from_file(args)`

Evaluate translation hypotheses from a file or a list of files of references. :param args: Evaluation parameters :return: None

`utils.evaluate_from_file.load_textfiles(references, hypotheses)`

Loads the references and hypothesis text files.

Parameters

- **references** – References files.
- **hypotheses** – Hypotheses file.

Returns

preprocess_binary_word_vectors module

`utils.preprocess_binary_word_vectors.parse_args()`

`utils.preprocess_binary_word_vectors.word2vec2npy(v_path, base_path_save, dest_filename)`

Preprocess pretrained binary vectors and stores them in a suitable format. :param v_path: Path to the binary vectors file. :param base_path_save: Path where the formatted vectors will be stored. :param dest_filename: Filename of the formatted vectors.

preprocess_text_word_vectors module

`utils.preprocess_text_word_vectors.parse_args()`

`utils.preprocess_text_word_vectors.`**`txtvec2npy`**(*v_path*, *base_path_save*, *dest_filename*)

Preprocess pretrained text vectors and stores them in a suitable format :param v_path: Path to the text vectors file. :param base_path_save: Path where the formatted vectors will be stored. :param dest_filename: Filename of the formatted vectors.

utils module

`utils.utils.`**`update_parameters`**(*params*, *updates*, *restrict=False*)

Updates the parameters from params with the ones specified in updates :param params: Parameters dictionary to update :param updates: Updater dictionary :param restrict: If True, parameters from the original dict are not overwritten. :return:

Module contents

2.7 Contact

If you have any trouble using NMT-Keras, please post a GitHub issue or drop an email to: lvapeab@prhlt.upv.es

2.7.1 Acknowledgement

Much of this library has been developed together with Marc Bolaños for other multimodal projects.

Related projects

To see other projects following the philosophy of NMT-Keras, take a look to:

- TMA: for egocentric captioning based on temporally-linked sequences.
- VIBIKNet: for visual question answering.
- ABiViRNet: for video description.
- Sentence SelectioNN for sentence classification and selection.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index