Joey NMT Documentation

Release 1.2

Joost Bastings and Julia Kreutzer

Mar 12, 2021

Contents

JoeyNMT is a minimalist neural machine translation toolkit for educational purposes.

It aims to be a clean and minimalistic code base to help novices pursuing the understanding of neural machine translation.

CHAPTER 1

Getting Started

1.1 Installation

1.1.1 Basics

First install Python >= 3.5, PyTorch >=v.0.4.1 and git.

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

```
$ python3 -m venv jnmt
$ source jnmt/bin/activate
```

1.1.2 Cloning

Then clone JoeyNMT from GitHub and switch to its root directory:

```
(jnmt)$ git clone https://github.com/joeynmt/joeynmt.git
(jnmt)$ cd joeynmt
```

1.1.3 Installing JoeyNMT

Install JoeyNMT and it's requirements:

```
(jnmt)$ pip3 install .
```

Run the unit tests to make sure your installation is working:

```
(jnmt)$ python3 -m unittest
```

Warning! When running on *GPU* you need to manually install the suitable PyTorch version for your [CUDA](https://developer.nvidia.com/cuda-zone) version. This is described in the [PyTorch installation instructions](https://pytorch.org/get-started/locally/).

You're ready to go!

1.2 Tutorial

In this tutorial you learn to build a recurrent neural translation system for a toy translation task, how to train, tune and test it.

Instead of following the example here, you might also run the Colab Notebook from the Masakhane project or the scripts compiled in Joey Toy Models, that both walk you through the installation, data preparation, training, evaluation.

1.2.1 1. Data Preparation

For training a translation model, you need parallel data, i.e. a collection of source sentences and reference translations that are aligned sentence-by-sentence and stored in two files, such that each line in the reference file is the translation of the same line in the source file.

Synthetic Data

For the sake of this tutorial, we'll simply generate synthetic data to mimic a real-world translation task. Our machine translation task is here to learn to reverse a given input sequence of integers.

For example, the input would be a source sentence like this:

```
14 46 43 2 36 6 20 8 38 17 3 24 13 49 8 25
```

And the correct "translation" would be:

```
25 8 49 13 24 3 17 38 8 20 6 36 2 43 46 14
```

Why is this an interesting toy task?

Let's generate some data!

```
python3 scripts/generate_reverse_task.py
```

This generates 50k training and 1k dev and test examples for integers between 0 and 50 of maximum length 25 for training and 30 for development and testing. Lets move it to a better directory.

```
mkdir test/data/reverse
mv train* test/data/reverse/
mv test* test/data/reverse/
mv dev* test/data/reverse/
```

Pre-processing

Before training a model on it, parallel data is most commonly filtered by length ratio, tokenized and true- or lowercased. For our reverse task, this is not important, but for real translation data it matters.

The Moses toolkit provides a set of useful scripts for this purpose. For a standard pipeline, follow for example the one described in the Sockeye paper.

In addition, you might want to build the NMT model not on the basis of words, but rather sub-words or characters (the `level` in JoeyNMT configurations). Currently, JoeyNMT supports word-based, character-based models and sub-word models with byte-pair-encodings (BPE) as learned with subword-nmt or sentencepiece.

1.2.2 2. Configuration

Once you have the data, it's time to build the NMT model.

In JoeyNMT, experiments are specified in configuration files, in YAML format. Most importantly, the configuration contains the description of the model architecture (e.g. number of hidden units in the encoder RNN), paths to the training, development and test data, and the training hyperparameters (learning rate, validation frequency etc.).

You can find examples in the configs directory. small.yaml contains a detailed explanation of all configuration options.

For the tutorial we'll use reverse.yaml. We'll go through it section by section.

1. Data Section

Here we give the path to the data (".src" is the source suffix, ".trg" is the target suffix) and indicate which segmentation level we want to train on, here simply on the word level, as opposed to the character level. The training set will be filtered by `max_sent_length`, i.e. only examples where source and target contain not more than 25 tokens are retained for training (that's the full data set for us). Source and target vocabulary are created from the training data, by keeping `src_voc_limit` source tokens that occur at least `src_voc_min_freq` times, and equivalently for the target side. If you want to use a pre-generated vocabulary, you can load it with `src_vocab` and `trg_vocab`. This will be important when loading a trained model for testing.

```
data:
    src: "src"
    trg: "trg"
    train: "test/data/reverse/train"
    dev: "test/data/reverse/dev"
    test: "test/data/reverse/test"
    level: "word"
    lowercase: False
    max_sent_length: 25
    src_voc_min_freq: 0
    src_voc_limit: 100
    trg_voc_min_freq: 0
    trg_voc_limit: 100
    #src_vocab: "reverse_model/src_vocab.txt"
    #trg_vocab: "reverse_model/trg_vocab.txt"
```

2. Training Section

This section describes how the model is trained. Training stops when either the learning rate decreased to `learning_rate_min` (when using a decreasing learning rate schedule) or the maximum number of epochs is reached. For individual schedulers and optimizers, we refer to the PyTorch documentation.

Here we're using the "plateau" scheduler that reduces the initial learning rate by `decrease_factor` whenever the `early_stopping_metric` has not improved for `patience` validations. Validations (with greedy decoding) are performed every `validation_freq` batches and every `logging_freq` batches the training batch loss will be logged.

Checkpoints for the model parameters are saved whenever a new high score in `early_stopping_metric`, here the `eval_metric` BLEU, has been reached. In order to not waste much memory on old checkpoints, we're

only keeping the `keep_last_ckpts` best checkpoints. We can also always keep the latest checkpoint by setting `save_latest_ckpt` to `True`. This will keep the latest checkpoint and delete the previous checkpoint as necessary.

At the beginning of each epoch the training data is shuffled if we set `shuffle` to True (there is actually no good reason for not doing so).

With *use_cuda* we can decide whether to train the model on GPU (True) or CPU (False). Note that for training on GPU you need the appropriate CUDA libraries installed.

Caution: In this example we set `overwrite: True` which you shouldn't do if you're running serious experiments, since it overwrites the existing `model_dir` and all its content if it already exists and you re-start training.

```
training:
    random_seed: 42
    optimizer: "adam"
    learning_rate: 0.001
    learning_rate_min: 0.0002
    weight_decay: 0.0
    clip_grad_norm: 1.0
    batch_size: 10
    scheduling: "plateau"
    patience: 5
    decrease_factor: 0.5
    early_stopping_metric: "eval_metric"
    epochs: 6
    validation_freq: 1000
    logging_freq: 100
    eval_metric: "bleu"
    model_dir: "reverse_model"
    overwrite: True
    shuffle: True
    use_cuda: False
    max_output_length: 30
    print_valid_sents: [0, 3, 6]
    keep_last_ckpts: 2
```

3. Testing Section

Here we only specify which decoding strategy we want to use during testing. If `beam_size: 1` the model greedily decodes, otherwise it uses a beam of `beam_size` to search for the best output. *alpha* is the length penalty for beam search (proposed in Wu et al. 2018).

```
testing:
    beam_size: 10
    alpha: 1.0
```

4. Model Section

Here we describe the model architecture and the initialization of parameters.

In this example we use a one-layer bidirectional LSTM encoder with 64 units, a one-layer LSTM decoder with also 64 units. Source and target embeddings both have the size of 16.

We're not going into details for the initialization, just know that it matters for tuning but that out default configurations should generally work fine. A detailed description for the initialization options is described in initialization.py.

Dropout is applied onto the input of the encoder RNN with dropout probability of 0.1, as well as to the input of the decoder RNN and to the input of the attention vector layer (`hidden_dropout`). Input feeding (Luong et al. 2015) means the attention vector is concatenated to the hidden state before feeding it to the RNN in the next step.

The first decoder state is simply initialized with zeros. For real translation tasks, the options are *last* (taking the last encoder state) or *bridge* (learning a projection of the last encoder state).

Encoder and decoder are connected through global attention, here through *luong* attention, aka the "general" (Luong et al. 2015) or bilinear attention mechanism.

```
model:
    initializer: "xavier"
    embed_initializer: "normal"
    embed_init_weight: 0.1
    bias_initializer: "zeros"
    init_rnn_orthogonal: False
    lstm_forget_gate: 0.
    encoder:
        rnn_type: "lstm"
        embeddings:
            embedding_dim: 16
            scale: False
        hidden_size: 64
        bidirectional: True
        dropout: 0.1
        num_layers: 1
    decoder:
        rnn_type: "lstm"
        embeddings:
            embedding_dim: 16
            scale: False
        hidden_size: 64
        dropout: 0.1
        hidden_dropout: 0.1
        num_layers: 1
        input_feeding: True
        init_hidden: "zero"
        attention: "luong"
```

That's it! We've specified all that we need to train a translation model for the reverse task.

1.2.3 3. Training

Start

For training, run the following command:

```
python3 -m joeynmt train configs/reverse.yaml
```

This will train a model on the reverse data specified in the config, validate on validation data, and store model parameters, vocabularies, validation outputs and a small number of attention plots in the `reverse_model` directory.

Progress Tracking

The Log File

During training the JoeyNMT will print the training log to stdout, and also save it to a log file `reverse_model/train.log`. It reports information about the model, like the total number of parameters, the vocabulary size, the data sizes. You can doublecheck that what you specified in the configuration above is actually matching the model that is now training.

After the reports on the model should see something like this:

```
2019-04-10 23:14:59,056 Epoch 1 Step: 800 Batch Loss: 58.698814 Tokens per Sec: 11418.
↪961022
2019-04-10 23:15:08,522 Epoch 1 Step: 1000 Batch Loss: 71.565094 Tokens per Sec:␣
↪14743.648984
2019-04-10 23:15:17,651 Hooray! New best validation result [eval_metric]!
2019-04-10 23:15:17,655 Example #0
2019-04-10 23:15:17,655          Raw source: ['33', '9', '15', '3', '14', '33', '32',
↪'42', '23', '12', '14', '17', '4', '35', '0', '48', '46', '36', '46', '27', '2', '34
↪', '35', '17', '36', '39', '7', '14', '9', '0']
2019-04-10 23:15:17,655          Source: 33 9 15 3 14 33 32 42 23 12 14 17 4 35 0 48␣
↪46 36 46 27 2 34 35 17 36 39 7 14 9 0
2019-04-10 23:15:17,655          Reference: 0 9 14 7 39 36 17 35 34 2 27 46 36 46 48 0␣
↪35 4 17 14 12 23 42 32 33 14 3 15 9 33
2019-04-10 23:15:17,655          Raw hypothesis: ['0', '9', '14', '7', '39', '36', '17
↪', '40', '35', '2', '26', '47', '22', '12', '46', '46', '42', '42', '42', '24', '24
↪', '24', '24', '24', '24', '24', '24', '24', '24', '24']
2019-04-10 23:15:17,655          Hypothesis: 0 9 14 7 39 36 17 40 35 2 26 47 22 12 46␣
↪46 42 42 42 24 24 24 24 24 24 24 24 24 24 24
...
2019-04-10 23:15:17,656 Validation result at epoch 1, step 1000: bleu: 37.957326,␣
↪loss: 34737.589844, ppl: 8.401401, duration: 9.1334s
```

The training batch loss is logged every 200 mini-batches, as specified in the configuration, and every 1000 batches the model is validated on the dev set. So after 1000 batches the model achieves a BLEU score of 37.96 (which will not be that fast for a real translation task, our reverse task is much easier). You can see that the model prediction is only partially correct, up to the 7th token.

The loss on individual batches might vary and not only decrease, but after every completed epoch, the accumulated training loss for the whole training set is reported. This quantity should decrease if your model is properly learning.

Validation Reports

The scores on the validation set express how well your model is generalizing to unseen data. The `validations.txt` file in the model directory reports the validation results (Loss, evaluation metric (here: BLEU), Perplexity (PPL)) and the current learning rate at every validation point.

For our example, the first lines should look like this:

```
Steps: 1000     Loss: 34737.58984       PPL: 8.40140    bleu: 37.95733  LR: 0.
↪00100000  *
Steps: 2000     Loss: 14954.59082       PPL: 2.49997    bleu: 74.06024  LR: 0.
↪00100000  *
Steps: 3000     Loss: 12533.76465       PPL: 2.15535    bleu: 83.41361  LR: 0.
↪00100000  *
Steps: 4000     Loss: 12846.20703       PPL: 2.19701    bleu: 80.79483  LR: 0.00100000
```

Models are saved whenever a new best validation score is reached, in `batch_no.ckpt`, where `batch_no` is the number of batches the model has been trained on so far. You can see when a checkpoint was saved by the asterisk at

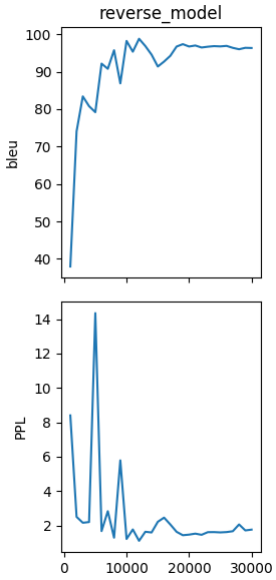
the end of the line in `validations.txt`. `best.ckpt` links to the checkpoint that has so far achieved the best validation score.

Learning Curves

JoeyNMT provides a script to plot validation scores with matplotlib. You can choose several models and metrics to plot. For now, we're interested in BLEU and perplexity and we want to save it as png.

```
python3 scripts/plot_validations.py reverse_model --plot_values bleu PPL  --output_
↪path reverse_model/bleu-ppl.png
```

It should look like this:



Tensorboard

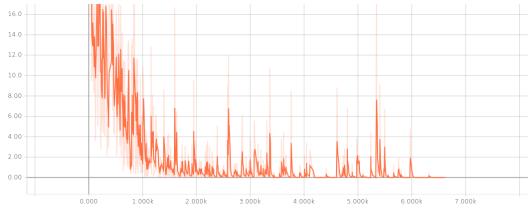
JoeyNMT additionally uses TensorboardX to visualize training and validation curves and attention matrices during training. Launch Tensorboard (requires installation that is not included in JoeyNMTs requirements) like this:

```
tensorboard --logdir reverse_model/tensorboard
```

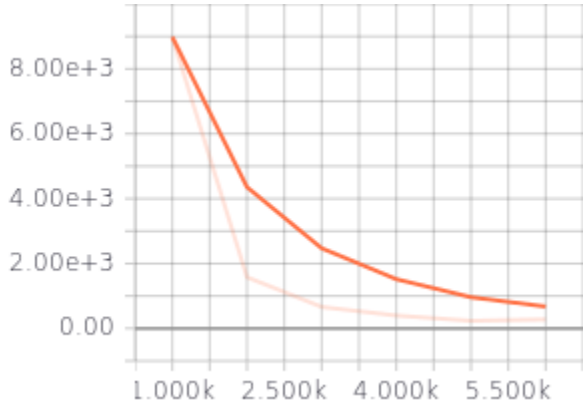
and then open the url (default: `localhost:6006`) with a browser.

You should see something like that:

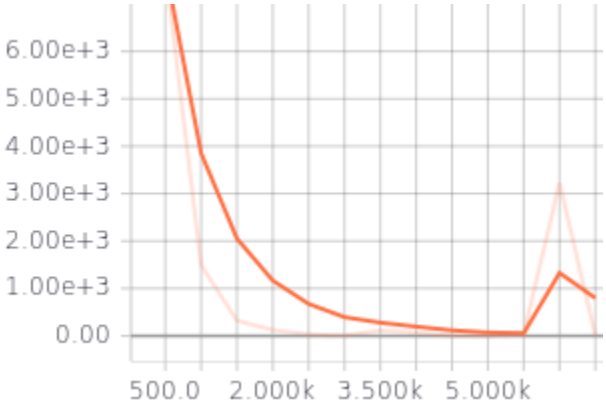
We can now inspect the training loss curves, both for individual batches



and for the whole training set:



and the validation loss:

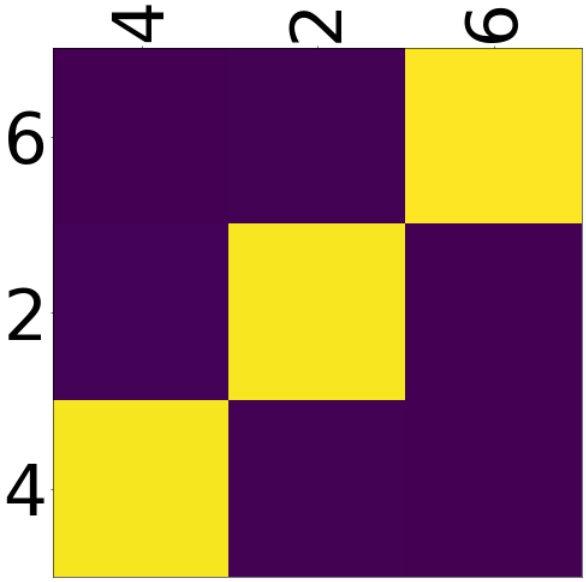


Looks good! Training and validation loss are decreasing, that means the model is doing well.

Attention Visualization

Attention scores often allow us a more visual inspection of what the model has learned. For every pair of source and target token the model computes attention scores, so we can visualize this matrix. JoeyNMT automatically saves plots of attention scores for examples of the validation set (the ones you picked for `print_valid_examples`) and saves them in your model directory.

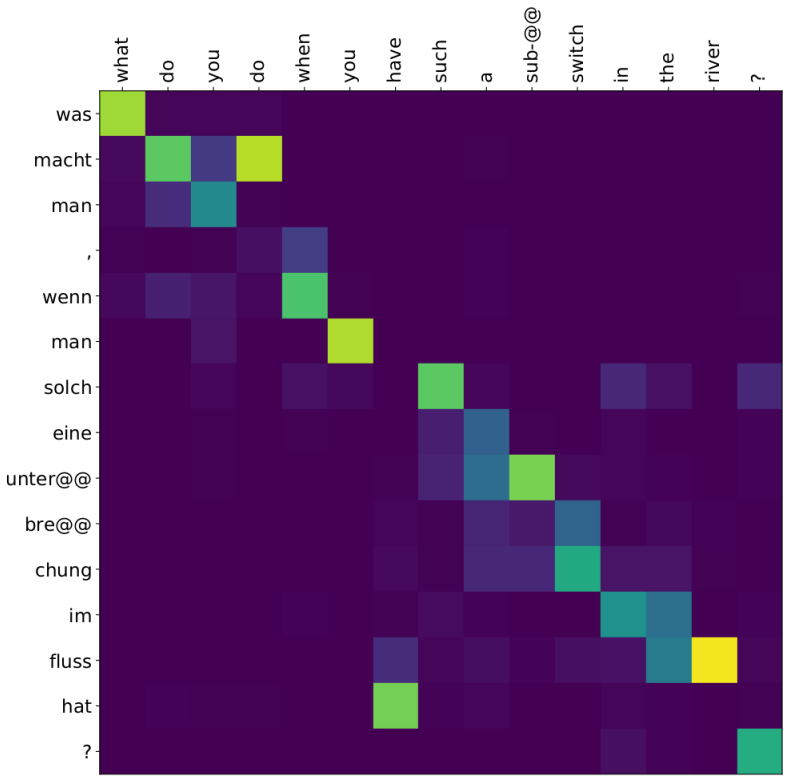
Here's an example, target tokens as columns and source tokens as rows:



The bright colors mean that these positions got high attention, the dark colors mean there was not much attention. We can see here that the model has figured out to give "2" on the source high attention when it has to generate "2" on the target side.

Tensorboard (tab: "images") allows us to inspect how attention develops over time, here's what happened for a relatively short sentence:

For real machine translation tasks, the attention looks less monotonic, for example for an IWSLT de-en model like this:



1.2.4 4. Testing

There are *three* options for testing what the model has learned.

In general, testing works by loading a trained model (`load_model` in the configuration) and feeding it new sources that it will generate predictions for.

1. Test Set Evaluation

For testing and evaluating on the parallel test set specified in the configuration, run

```
python3 -m joeynmt test reverse_model/config.yaml --output_path reverse_model/
↪predictions
```

This will generate beam search translations for dev and test set (as specified in the configuration) in `reverse_model/predictions.[dev|test]` with the latest/best model in the `reverse_model`` directory (or a specific checkpoint set with ``load_model`). It will also evaluate the outputs with `eval_metric` and print the evaluation result. If `--output_path` is not specified, it will not store the translation, and solely do the evaluation and print the results.

The evaluation for our reverse model should look like this:

```
test bleu[13a]: 98.48812178559285 [Beam search decoding with beam size = 10 and alpha␣
↪= 1.0]
Translations saved to: reverse_model/test_predictions.test
dev  bleu[13a]: 98.80524689263555 [Beam search decoding with beam size = 10 and alpha␣
↪= 1.0]
Translations saved to: reverse_model/test_predictions.dev
```

Once again you can see that the reverse task is relatively easy to learn, while for translation high BLEU scores like this would be miraculous/suspicious.

2. File Translation

In order to translate the contents of any file (one source sentence per line) not contained in the configuration (here `my_input.txt`), simply run

```
echo $'2 34 43 21 2 \n3 4 5 6 7 8 9 10 11 12' > my_input.txt
python3 -m joeynmt translate reverse_model/config.yaml < my_input.txt
```

The translations will be written to stdout or alternatively `--output_path` if specified.

For this example the output (all correct!) will be

```
2 21 43 34 2
12 11 10 9 8 7 6 5 4 3
```

3. Interactive

If you just want try a few examples, run

```
python3 -m joeynmt translate reverse_model/config.yaml
```

and you'll be prompted to type input sentences that JoeyNMT will then translate with the model specified in the configuration.

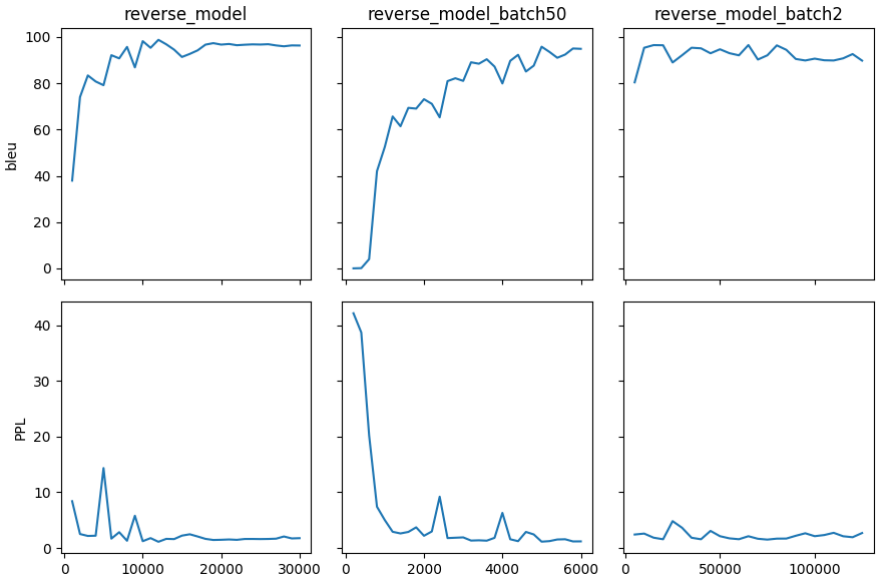
Let's try a challenging long one:

```
Please enter a source sentence (pre-processed):
1 23 23 43 34 2 2 2 2 2 4 5 32 47 47 47 21 20 0 10 10 10 10 10 8 7 33 36 37
JoeyNMT: 37 36 33 7 8 10 10 10 10 10 0 20 21 47 47 47 32 5 4 2 2 2 2 2 34 43 23 10 1
```

1.2.5 5. Tuning

Trying out different combinations of hyperparameters to improve the model is called "tuning". Improving the model could mean in terms of generalization performance at the end of training, faster convergence or making it more efficient or smaller while achieving the same quality. For our case that means going back to the configuration and changing a few of the hyperparameters.

For example, let's try out what happens if we increase the batch size to 50 or reduce it to 2 (and change the "model_dir"!). For a one-to-one comparison we consequently need to divide or multiply the validation frequency by 5, respectively, since the "steps" are counted in terms of mini-batches. In the plot below we can see that we reach approximately the same quality after 6 epochs, but that the shape of the curves looks quite different. In this case, a small mini-batch size leads to the fastest progress but also takes noticeabl longer to complete the full 6 epochs in terms of wall-clock time.



You might have noticed that there are lots hyperparameters and that you can't possible try out all combinations to find the best model. What is commonly done instead of an exhaustive search is grid search over a small subset of hyperparameters, or random search (Bergstra & Bengio 2012), which is usually the more efficient solution.

1.2.6 6. What's next?

If you want to implement something new in JoeyNMT or dive a bit deeper, you should take a look at the architecture *Overview* and explore the API documentation of modules.

Other than that, we hope that you found this tutorial helpful. Please leave an issue on Github if you had trouble with anything or have ideas for improvement.

1.3 Overview

This page gives an overview of the particular organization of the code. If you want to modify or contribute to the code, this is a must-read, so you know where to enter your code.

For a detailed documentation of the API, go to modules.

1.3.1 Modes

When JoeyNMT is called from the command line, the mode ("train/test/translate") determines what happens next.

The **"train"** mode leads to `training.py`, where executes the following steps:

1. load the configuration file
2. load the data and build the vocabularies
3. build the model
4. create a training manager
5. train and validate the model (includes saving checkpoints)
6. test the model with the best checkpoint (if test data given)

"test" and **"translate"** mode are handled by `prediction.py`. In **"test"** mode, JoeyNMT does the following:

1. load the configuration file
2. load the data and vocabulary files
3. load the model from checkpoint
4. predict hypotheses for the test set
5. evaluate hypotheses against references (if given)

The **"translate"** mode is similar, but it loads source sentences either from an *external* file or prompts lines of *inputs from the user* and does not perform an evaluation.

1.3.2 Training Management

The training process is managed by the TrainManager. The manager receives a model and then performs the following steps: parses the input configuration, sets up the logger, schedules the learning rate, sets up the optimizer and counters for update steps. It then keeps track of the current best checkpoint to determine when to stop training. Most of the hyperparameters in the "training" section of the configuration file are turned into attributes of the TrainManager.

1.3.3 Encoder-Decoder Model

The encoder-decoder model architecture is defined in model.py. This is where encoder and decoder get connected. The forward pass as well as the computation of the training loss and the generation of predictions of the combined encoder-decoder model are defined here.

Individual encoders and decoders are defined with their forward functions in encoders.py and decoders.py.

1.3.4 Data Handling

Mini-Batching

The **training** data is split into buckets of similar source and target length and then split into batches (data.py) to reduce the amount of padding, i.e. waste of computation time. The samples within each mini-batch are sorted, so that we can make use of PyTorch's efficient RNN sequence padding and packing functions.

For **inference**, we sort the data as well (when creating batches with batch.py), but we keep track of the original order so that we can revert the order of the model outputs. This trick speeds up validation and also testing.

Vocabulary

For the creation of the vocabulary (vocabulary.py), all tokens occuring in the training set are collected, sorted and optionally filtered by frequency and then cut off as specified in the configuration. The vocabularies are stored in the model directory. The vocabulary files contain one token per line, where the line number corresponds to the index of the token in the vocabulary.

Data Loading

At the current state, we use Torchtext for data loading and the transformation of files of strings to PyTorch tensors. Most importantly, the code (*data.py*) works with the Dataset and Field objects: one field for source and one for target, creating a TranslationDataset.

1.3.5 Inference

For inference we run either beam search or greedy decoding, both implemented in search.py. We chose to largely adopt the implementation of beam search in OpenNMT-py for the neat solution of dropping hypotheses from the batch when they are finished.

1.3.6 Checkpoints

The TrainManager takes care of saving checkpoints whenever the model has reached a new validation highscore (keeping a configurable number of checkpoints in total). The checkpoints do not only contain the model parameters (`model_state`), but also the cumulative count of training tokens and steps, the highscore and iteration count for that highscore, the state of the optimizer, the scheduler and the data iterator. This ensures a seamless continuation of training when training is interrupted.

From `_save_checkpoint`:

```
model_state_dict = self.model.module.state_dict() if \
isinstance(self.model, torch.nn.DataParallel) else self.model.state_dict()
state = {
    "steps": self.steps,
    "total_tokens": self.total_tokens,
    "best_ckpt_score": self.best_ckpt_score,
    "best_ckpt_iteration": self.best_ckpt_iteration,
    "model_state": model_state_dict,
    "optimizer_state": self.optimizer.state_dict(),
    "scheduler_state": self.scheduler.state_dict() if \
    self.scheduler is not None else None,
    'amp_state': amp.state_dict() if self.fp16 else None
    'train_iter_state': train_iter.state_dict()
}
```

1.4 API Documentation

1.4.1 Indices and tables

- genindex
- modindex
- search

1.5 Frequently Asked Questions

1.5.1 Documentation

- **Are there any Notebooks for Joey?** Check out the Colab Notebook from the Masakhane project that walks you through the installation, data preparation, training, evaluation.
- **Is there a bunch of scripts to run all those Joey commands?** Check out the scripts compiled in Joey Toy Models, that also walk you through the installation, data preparation, training, evaluation, and even data download and pre-processing.

- **I can't find the information I'm looking for. What now?** Open an issue on GitHub or post a question on gitter.

1.5.2 Usage

Training

- **How can I train the model on GPU/CPU?** First of all, make sure you have the correct version of pytorch installed. When running on *GPU* you need to manually install the suitable PyTorch version for your CUDA version. This is described in the PyTorch installation instructions. Then set the `use_cuda` flag in the configuration to True for training on GPU (requires CUDA) or to False for training on CPU.
- **Does Joey NMT support multi-GPU processing?** ~~At the current stage, the code does not support multi-GPU processing. Contributions welcome :)~~ [UPDATE] In version 1.0, we integrated multi-gpu and half-precision support.
- **How can I stop training?** Simply press Control+C.
- **My training data is huge and I actually don't want to train on it all. What can I do?** You could use the `random_train_subset` parameter in the data section of the configuration to load only a random subset of the training data. If you change the random seed, this selection changes too. So you could train on multiple random subsets and then ensemble the models with `scripts/average_checkpoints.py`.
- **How can I see how well my model is doing?**

  1. *Training log*: Validation results and training loss (after each epoch and batch) are reported in the training log file `train.log` in your model directory.
  2. *Validation reports*: `validations.txt` contains the validation results, learning rates and indicators when a checkpoint was saved. You can easily plot the validation results with this script, e.g.

     ```
     python3 scripts/plot_validation.py model_dir --plot_values bleu PPL --
     ↪output_path my_plot.pdf
     ```

  3. *Tensorboard*: Validation results, training losses and attention scores are also stored in summaries for Tensorboard. Launch Tensorboard with

     ```
     tensorboard --logdir model_dir/tensorboard
     ```

     and then open the url (default: `localhost:6006`) with a browser.

  See *Tutorial*, section "Progress Tracking", for a detailed description of the quantities being logged.
- **How often should I validate?** Depends on the size of your data. For most use-cases you want to validate at least once per epoch. Say you have 100k training examples and train with mini-batches of size 20, then you should set `validation_freq` to 5000 (100k/20) to validate once per epoch.
- **How can I perform domain adaptation or fine-tuning?** Both approaches are similar, so we call the fine-tuning data *in-domain* data in the following.

  1. First train your model on one dataset (the *out-of-domain* data).
  2. Modify the original configuration file (or better a copy of it) in the data section to point to the new *in-domain* data. Specify which vocabularies to use: `src_vocab: out-of-domain-model/src_vocab.txt` and likewise for `trg_vocab`. You have to specify this, otherwise JoeyNMT will try to build a new vocabulary from the new in-domain data, which the out-of-domain model wasn't built with. In the training section, specify which checkpoint of the out-of-domain model you want to start adapting: `load_model: out-of-domain-model/best.ckpt`. If you set "reset_best_ckpt": True", previously stored high scores under your metric will be ignored, and if you set

"reset_scheduler" and "reset_optimizer" you can also overwrite the stored scheduler and optimizer with the new ones in your configuration. Use this if the scores on your new dev set are lower than on the old dev set, or if you use a different metric or schedule for fine-tuning.

3. Train the in-domain model.

- **What if training is interrupted and I need to resume it?** Modify the configuration to load the latest checkpoint (`load_model`) and the vocabularies (`src_vocab`, `trg_vocab`) and to write the model into a new directory (`model_dir`). Then train with this configuration. Joey can be configured to save the checkpoint after every validation run, ensuring that you don't have to resume training from an old checkpoint. This can be enabled by setting `save_latest_ckpt` to `True` in your config file.

Tuning

- **Which default hyperparameters should I use?** There is no universal answer to this question. We recommend you to check publications that used the same data as you're using (or at least the same language pair and data size) and find out how large their models where, how long they trained them etc. You might also get inspiration from the benchmarks that we report. Their configuration files can be found in the `configs` directory.
- **Which hyperparameters should I change first?** As above, there is no universal answer. Some things to consider:
  - The *learning rate* determines how fast you can possibly learn. If you use a learning rate scheduler, make sure to configure it in a way that it doesn't reduce the learning rate too fast. Different optimizers need individually tuned learning rates as well.
  - The *model size and depth* matters. Check the benchmarks and their model and data sizes to get an estimate what might work.

Tensorboard

- **How can I start Tensorboard for a model that I trained on a remote server?** Start jupyter notebook in the Joey NMT directory, remote_port_number should be a free port, e.g. 8889.

  Create an SSH tunnel on the local machine (with free ports yyyy (local) and xxxx (remote)):

  ```
  ssh -N -L localhost:yyyy:localhost:xxxx <remote_user@remote_user>
  ```

  On the remote machine, launch tensorboard and pass it the path to the tensorboard logs of your model:

  ```
  tensorboard --logdir model_dir/tensorboard --host=localhost --port=xxxx
  ```

  Then navigate to *localhost:yyyy* in a browser on your local machine.

Configurations

- **Where can I find the default values for the settings in the configuration file?** Either check the configuration file or *API Documentation* for individual modules. Please note that there is no guarantee that the default setting is a good setting.
- **What happens if I made a mistake when configuring my model?** JoeyNMT will complain by raising a `ConfigurationError`.
- **How many parameters has my model?** The number of parameters is logged in the training log file. You can find it in the model directory in `train.log`. Search for the line containing "Total params:".

- **What's the influence of the random seed?** The random seed is used for all random factors in NMT training, such as the initialization of model parameters and the order of training samples. If you train two identical models with the same random seed, they should behave exactly the same.
- **How do you count the number of hidden units for bi-directional RNNs?** A bi-directional RNN with k hidden units will have k hidden units in the forward RNN plus k for the backward RNN. This might be different in other toolkits where the number of hidden units is divided by two to use half of them each for backward and forward RNN.
- **My model with configs/small.yaml doesn't perform well.'** No surprise! This configuration is created for the purpose of documentation: it contains all parameter settings with a description. It does not perform well on the actual task that it uses. Try the reverse or copy task instead!
- **What does batch_type mean?** The code operates on mini-batches, i.e., blocks of inputs instead of single inputs. Several inputs are grouped into one mini-batch. This grouping can either be done by defining a maximum number of sentences to be in one mini-batch (*batch_type: "sentence"*), or by a maximum number of tokens (*batch_type: "token"*). For Transformer models, mini-batching is usually done by tokens.
- **Do I need a warm-up scheduler with the Transformer architecture?** No. The 'Noam scheduler' that was introduced with the original Transformer architecture works well for the data sets (several millions) described in the paper (Vaswani et al. 2017). However, on different data it might require a careful tuning of the warm-up schedule. We experienced good performance with the plateau scheduler as well, which is usally easier to tune. Popel and Bojar (2018) give further tips on how to tune the hyper-parameters for the Transformer.

Data

- **Does JoeyNMT pre-process my data?** JoeyNMT does *not* include any pre-processing like tokenization, filtering by length ratio, normalization or learning/applying of BPEs. For that purpose, you might find the tools provided by the Moses decoder useful, as well as the subwordnmt or sentencepiece library for BPEs. An example of a pre-processing pipeline is show in the data preparation script for IWLST 2014. However, the training data gets *filtered* by the `max_sent_length` (keeping all training instances where source and target are up to that length) that you specify in the data section of the configuration file. You can find an example of a data pre-processing pipeline here.
- **Does JoeyNMT post-process your data?** JoeyNMT does generally *not* perform any post-processing like detokenization, recasing or the like. The only exception is when you run it with ´level='bpe'´ – then it *merges* the BPEs for your convenience. This holds for computing validation BLEU and test BLEU scores, so that they're not computed on subwords, but the previously split tokens.

Debugging

- **My model doesn't work. What can I do?** First of all, invest in diagnostics: what exactly is not working? Is the training loss going down? Is the validation loss going down? Are there any patterns in the weirdness of the model outputs? Answers to these questions will help you locate the source of the problem. Andrej Karpathy wrote this wonderful recipe for training neural nets by - it has lots of advice on how to find out what's going wrong and how to fix it. Specifically for NMT, here's three things we can recommend: - *Synthetic data*: If you modified the code, it might help to inspect tensors and outputs manually for a synthetic task like the reverse task presented in the *Tutorial*. - *Data*: If you're working with a standard model, doublecheck whether your data is properly aligned, properly pre-processed, properly filtered and whether the vocabularies cover a reasonable amount of tokens. - *Hyperparameters*: Try a smaller/larger/deeper/shallower model architecture with smaller/larger learning rates, different optimizers and turn off schedulers. It might be worth to try different initialization options. Train longer and validate less frequently, maybe training just takes longer than you'd expect.

- **My model takes too much memory. What can I do?** Consider reducing `batch_size`. The mini-batch size can be virtually increased by a factor of *k* by setting `batch_multiplier` to *k*. Tensor operations are still performed with `batch_size` instances each, but model updates are done after *k* of these mini-batches.
- **My model performs well on the validation set, but terrible on the test set. What's wrong?** Make sure that your validation set is similar to the data you want to test on, that it's large enough and that you're not "over-tuning" your model.
- **My model produces translations that are generally too short. What's wrong?** Make sure that `max_sent_length` for the filtering of the data (data section in configuration) is set sufficiently high. The training log reports how many training sentences remain after filtering. `max_output_length` (training section) limits the length of the outputs during inference, so make sure this one is also set correctly.
- **Evaluation breaks because I get an empty iterator. What's wrong?** If you're using *batch_type: token*, try increasing the *eval_batch_size*.

1.5.3 Features

- **Which models does Joey NMT implement?** For the exact description of the RNN and Transformer model, check out the paper.
- **Why is there no convolutional model?** We might add it in the future, but from our experience, the most popular models are recurrent and self-attentional.
- **How are the parameters initialized?** Check the description in initialization.py.
- **Is there the option to ensemble multiple models?** You can do checkpoint averaging to combine multiple models. Use the average_checkpoints script.
- **What is a bridge?** We call the connection between recurrent encoder and decoder states the *bridge*. This can either mean that the decoder states are initialized by copying the last (forward) encoder state (`init_hidden: "last"`), by learning a projection of the last encoder state (`init_hidden: "bridge"`) or simply zeros (`init_hidden: "zero"`).
- **Does learning rate scheduling matter?** Yes. There's a whole branch of research on how to find and modify a good learning rate so that your model ends up in a good place. For JoeyNMT it's most important that you don't decrease your learning rate too quickly, which might happen if you train with very frequent validations (`validation_freq`) and low `patience` for a plateau-based scheduler. So if you change the validation frequency, adapt the patience as well. We recommend to start by finding a good constant learning rate and then add a scheduler that decays this initial rate at a point where the constant learning rate does not further improve the model.
- **What is early stopping?** Early stopping means that training should be stopped when the model's generalization starts to degrade. Jason Brownlee wrote a neat blogpost describing intuition and techniques for early stopping. In JoeyNMT, model checkpoints are stored whenever a new high score is achieved on the validation set, so when training ends, the latest checkpoint automatically captures the model parameters at the early stopping point. There's three options for measuring the high score on the validation set: the evaluation metric (`eval_metric`), perplexity (`ppl`), and the loss (`loss`). Set `early_stopping_metric` in the training configuration to either of those.
- **Is validation performed with greedy decoding or beam search?** Greedy decoding, since it's faster and usually aligns with model selection by beam search validation.
- **What's the difference between "max_sent_length" and and "max_output_length"?** `max_sent_length` determines the maximum source and target length of the training data, `max_output_length` is the maximum length of the translations that your model will be asked to produce.

- **How is the vocabulary generated?** See the *Tutorial*, section "Configuration - Data Section".
- **What does freezing mean?** *Freezing* means that you don't update a subset of your parameters. If you freeze all parts of your model, it won't get updated (which doesn't make much sense). It might, however, might sense to update only a subset of the parameters in the case where you have a pre-trained model and want to carefully fine-tune it to e.g. a new domain. For the modules you want to freeze, set `freeze: True` in the corresponding configuration section.

1.5.4 Model Extensions

- **I want to extend Joey NMT – where do I start? Where do I have to modify the code?** Depends on the scope of your extension. In general, we can recommend describing the desired behavior in the config (e.g. 'use_my_feature:True') and then passing this value along the forward pass and modify the model according to it. If your just loading more/richer inputs, you will only have to modify the part from the corpus reading to the encoder input. If you want to modify the training objective, you will naturally work in 'loss.py'. Logging and unit tests are very useful tools for tracking the changes of your implementation as well.
- **How do I integrate a new learning rate scheduler?**
  1. Check out the existing schedulers in builders.py, some of them are imported from PyTorch. The "Noam" scheduler is implemented here directly, you can use its code as a template how to implement a new scheduler.
  2. You basically need to implement the `step` function that implements whatever happens when the scheduler is asked to make a step (either after every validation (`scheduler_step_at="validation"`) or every batch (`scheduler_step_at="step"`)). In that step, the learning rate can be modified just as you like (`rate = self._compute_rate()`). In order to make an effective update of the learning rate, the learning rate for the optimizer's parameter groups have to be set to the new value (`for p in self.optimizer.param_groups: p['lr'] = rate`).
  3. The last thing that is missing is the parsing of configuration parameters to build the scheduler object. Once again, follow the example of existing schedulers and integrate the code for constructing your new scheduler in the `build_scheduler` function.
  4. Give the new scheduler a try! Integrate it in a basic configuration file and check in the training log and the validation reports whether the learning rate is behaving as desired.

1.5.5 Miscellaneous

- **Why should I use JoeyNMT rather than other NMT toolkits?** It's easy to use, it is well documented, and it works just as well as other toolkits out-of-the-box. It does and will not implement all latest features, but rather the core features that make up for 99% of the quality. That means for you, once you know how to work with it, we guarantee you the code won't completely change from one day to the next.
- **I found a bug in your code, what should I do?** Make a Pull Request on GitHub and describe what it did and how you fixed it.
- **How can I check whether my model is significantly better than my baseline model?** Repeat your experiment with multiple random seeds (`random_seed`) to measure the variance. You can use techniques like approximate randomization or bootstrap sampling to test the significance of the difference in evaluation score between the baseline's output and your model's output, e.g. with multeval.
- **Where can I find training data?** See *Resources*, section "Data".

1.5.6 Contributing

- **How can I contribute?** Check out the current issues and look for "beginner-friendly" tags and grab one of these.
- **What's in a Pull Request?** Opening a pull request means that you have written code that you want to contribute to Joey NMT. In order to communicate what your code does, please write a description of new features, defaults etc. Your new code should also pass tests and adher to style guidelines, this will be tested automatically. The code will only be pushed when all issues raised by reviewers have been addressed. See also here.

1.5.7 Evaluation

- **Which quality metrics does JoeyNMT report?** JoeyNMT reports BLEU, chrF, sentence- and token-level accuracy. You can choose which of those to report with setting *eval_metric* accordingly. As a default, we recommend BLEU since it is a standard metric. However, not all BLEU implementations compute the score in the same way, as discussed in this paper by Matt Post. So the scores that you obtain might not be comparable to those published in a paper, *even* if the data is identical!
- **Which library is JoeyNMT using to compute BLEU scores?** JoeyNMT uses sacrebleu to compute BLEU and chrF scores. It uses the raw_corpus_bleu scoring function that excludes special de/tokenization or smoothing. This is done to respect the tokenization that is inherent in the provided input data. However, that means that the BLEU score you get out of Joey is *dependent on your input tokenization*, so be careful when comparing it to scores you find in literature.
- **Can I publish the BLEU scores JoeyNMT reports on my test set?** As described in the two preceding questions, BLEU reporting has to be handled with care, since it depends on tokenizers and implementations. Generally, whenever you report BLEU scores, report as well how you computed them. This is essential for reproducability of results and future comparisons. If you compare to previous benchmarks or scores, first find out how these were computed. Our recommendation is as follows:
  1. Use the scores that Joey reports on your validation set for tuning and selecting the best model.
  2. Then translate your test set once (in "translate" mode), and post-process the produced translations accordingly, e.g., detokenize it, restore casing.
  3. Use the BLEU scoring library of your choice, this is the one that is reported in previous benchmarks, or e.g. sacrebleu (see above). Make sure to set tokenization flags correctly.
  4. Report these scores together with a description of how you computed them, ideally provide a script with your code.

1.6 Resources

1.6.1 Neural Machine Translation

If you want to learn more about neural machine translation, check out the following resources.

Toolkits

A comprehensive list of NMT toolkits, ordered by deep learning backend can be found here.

Tutorials

- The Annotated Transformer by Alexander Rush
- The Annotated Encoder-Decoder by Jasmijn Bastings
- Graham Neubig: Neural Machine Translation and Sequence-to-sequence Models: A Tutorial.
- Philipp Koehn: Neural Machine Translation.
- Video recording of Chris Manning's lecture on "NMT and Models with Attention" at Stanford (2017)

Publications

- NMT papers in the ACL anthology
- statmt.org survey of NMT publications
- THUNLP-MT MT reading list

Data

- WMT: The shared tasks of the yearly Conference on Machine Translation (WMT) provide lots of parallel data
- OPUS: The OPUS project collects publicly available parallel data and provides it to everyone on their website.

1.6.2 PyTorch

Here's a collection of links that should help you get started or improve your coding skills with PyTorch:

- Intro to PyTorch from Udacity's Deep Learning course (Jupyter notebooks)
- 60 min Blitz tutorial by Soumith Chintala
- Fast AI's MOOC "Practical Deep Learning for Coders" for a practical introduction to Deep Learning and PyTorch

1.6.3 Git Versioning

Never worked with Git before? For the basics, check out this tutorial by Roger Dudler and for more advanced usage this one by Lars Vogel.