Joey NMT Documentation

Release 2.3.0

Jasmijn Bastings and Julia Kreutzer

Jan 25, 2024

GETTING STARTED

CHAPTER

ONE

GOALS & PURPOSES

Joey NMT framework is developed for educational purposes. It aims to be a **clean** and **minimalistic** code base to help novices find fast answers to the following questions.

- How to implement classic NMT architectures (RNN and Transformer) in PyTorch?
- What are the building blocks of these architectures and how do they interact?
- How to modify these blocks (e.g. deeper, wider, . . .)?
- How to modify the training procedure (e.g. add a regularizer)?

In contrast to other NMT frameworks, we will **not** aim for the most recent features or speed through engineering or training tricks since this often goes in hand with an increase in code complexity and a decrease in readability.

However, Joey NMT re-implements baselines from major publications.

CHAPTER

TWO

FEATURES

Joey NMT implements the following features (aka the minimalist toolkit of NMT):

- Recurrent Encoder-Decoder with GRUs or LSTMs
- Transformer Encoder-Decoder
- Attention Types: MLP, Dot, Multi-Head, Bilinear
- Word-, BPE- and character-based tokenization
- BLEU, ChrF evaluation
- Beam search with length penalty and greedy decoding
- Customizable initialization
- Attention visualization
- Learning curve plotting
- Scoring hypotheses and references
- Multilingual translation with language tags

2.1 Installation

2.1.1 Basics

First install Python >= 3.10, PyTorch >=v.2.0.0 and git.

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

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

2.1.2 Cloning

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

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

Note: For Windows users, we recommend to doublecheck whether txt files (i.e. *test/data/toy/**) have utf-8 encoding.

2.1.3 Installing JoeyNMT

Install JoeyNMT and its requirements:

```
(jnmt)$ python -m pip install -e .
```

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

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

Warning: 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.

You're ready to go!

2.2 Command-line Interface

Joey NMT has 3 modes: `train`, `test`, and `translate`, and all of them takes a YAML-style config file as argument. You can find examples in the `configs` directory. transformer_small.yaml contains a detailed explanation of configuration options.

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.).

Note: Note that subword model training and joint vocabulary creation is not included in the 3 modes above, has to be done separately. We provide a script that takes care of it: build_vocab.py.

```
python scripts/build_vocab.py configs/transformer_small.yaml --joint
```

2.2.1 `train` mode

For training, run

```
python -m joeynmt train configs/transformer_small.yaml
```

This will train a model on the training data, validate on validation data, and store model parameters, vocabularies, validation outputs. All needed information should be specified in the `data`, `training` and `model` sections of the config file (here transformer_small.yaml).

```
model_dir/
├── *.ckpt          # checkpoints
├── *.hyps          # translated texts at validation
├── config.yaml     # config file
├── spm.model       # sentencepiece model / subword-nmt codes file
├── src_vocab.txt   # src vocab
├── trg_vocab.txt   # trg vocab
├── train.log       # train log
└── validation.txt  # validation scores
```

Danger: Be careful not to overwrite `model_dir`; set `overwrite: False` in the config file.

2.2.2 `test` mode

This mode will generate translations for validation and test set (as specified in the configuration) in `model_dir/out.[dev|test]`.

```
python -m joeynmt test configs/transformer_small.yaml
```

You can specify the ckpt path explicitly in the config file. If `load_model` is not given in the config, the best model in `model_dir` will be used to generate translations.

You can specify i.e. sacrebleu options in the `test` section of the config file.

Note: average_checkpoints.py will generate averaged checkpoints for you.

```
python scripts/average_checkpoints.py --inputs model_dir/*00.ckpt --output model_dir/avg.
↪ckpt
```

If you want to output the log-probabilities of the hypotheses or references, you can specify `return_score: 'hyp'` or `return_score: 'ref'` in the testing section of the config. And run `test` with `--output-path` and `--save-scores` options.

```
python -m joeynmt test configs/transformer_small.yaml --output-path model_dir/pred --
↪save-scores
```

This will generate `model_dir/pred.{dev|test}.{scores|tokens}` which contains scores and corresponding tokens.

Tip:

- If you set `return_score: 'hyp'` with greedy decoding, then token-wise scores will be returned. The beam search will return sequence-level scores, because the scores are summed up per sequence during beam exploration.
- If you set `return_score: 'ref'`, the model looks up the probabilities of the given ground truth tokens, and both decoding and evaluation will be skipped.
- If you specify `n_best` > 1 in config, the first translation in the nbest list will be used in the evaluation.

2.2.3 `translate` mode

This mode accepts inputs from stdin and generate translations.

File translation

```
python -m joeynmt translate configs/transformer_small.yaml < my_input.txt > output.txt
```

Interactive translation

```
python -m joeynmt translate configs/transformer_small.yaml
```

You'll be prompted to type an input sentence. Joey NMT will then translate with the model specified in the config file.

2.3 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 synthetic example here, you might also run the quick start guide that walks you step-by-step through the installation, data preparation, training, evaluation using "real" translation dataset from Tatoeba.

Torchhub tutorial demonstrates how to generate translation from a pretrained model via Torchhub API.

2.3.1 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!

```
python 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. The generated files are placed under *test/data/reverse/*.

```
wc -l test/data/reverse/*
```

```
 1000 test/data/reverse/dev.src
 1000 test/data/reverse/dev.trg
 1000 test/data/reverse/test.src
 1000 test/data/reverse/test.trg
50000 test/data/reverse/train.src
50000 test/data/reverse/train.trg
```

2.3.2 2. Configuration

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

In Joey NMT, 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. rnn_small.yaml contains a detailed explanation of all configuration options.

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

Top Section

Here we specify general settings applied both in training and prediction. 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.

```yaml
name: "reverse_experiment"
joeynmt_version: "2.3.0"
model_dir: "reverse_model"
use_cuda: False
fp16: False
random_seed: 42
```

Data Section

Here we give the path to the data (".src" is the source suffix, ".trg" is the target suffix of the plain txt files) and for each side separately, 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_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 `voc_limit` source tokens that occur at least `voc_min_freq` times, and equivalently for the target side. If you want to use a pre-generated vocabulary, you can load it in `voc_file` field. This will be important when loading a trained model for testing. `special_symbols` section defines special tokens required to control training and generation.

```
data:
    train: "test/data/reverse/train"
    dev: "test/data/reverse/dev"
    test: "test/data/reverse/test"
    dataset_type: "plain"
    src:
        lang: "src"
        max_length: 25
        level: "word"
        voc_limit: 100
        voc_min_freq: 0
        #voc_file: src_vocab.txt
    trg:
        lang: "trg"
        max_length: 25
        level: "word"
        voc_limit: 100
        voc_min_freq: 0
        #voc_file: trg_vocab.txt
    special_symbols:
        unk_token: "<unk>"
        unk_id: 0
        pad_token: "<pad>"
        pad_id: 1
        bos_token: "<s>"
        bos_id: 2
        eos_token: "</s>"
        eos_id: 3
```

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 not to waste much memory on old checkpoints, we're only keeping the `keep_best_ckpts` best checkpoints. Nevertheless, we always keep the latest checkpoint so that one can resume the training from that point. By setting `keep_best_ckpts = -1`, you can prevent to delete any checkpoints.

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).

```
training:
    #load_model: "reverse_model/best.ckpt"
    optimizer: "adamw"
    learning_rate: 0.001
    learning_rate_min: 0.0002
    weight_decay: 0.0
    clip_grad_norm: 1.0
    batch_size: 12
    batch_type: "sentence"
    batch_multiplier: 2
    scheduling: "plateau"
    patience: 5
    decrease_factor: 0.5
    early_stopping_metric: "bleu"
    epochs: 5
    validation_freq: 1000
    logging_freq: 100
    shuffle: True
    print_valid_sents: [0, 3, 6]
    keep_best_ckpts: 2
    overwrite: True
```

Danger: 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.

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. `beam_alpha` is the length penalty for beam search (proposed in Wu et al. 2018).

```
testing:
    #load_model: "reverse_model/best.ckpt"
    n_best: 1
    beam_size: 1
    beam_alpha: 1.0
    eval_metrics: ["bleu"]
    min_output_length: 1
    max_output_length: 30
    batch_size: 12
    batch_type: "sentence"
    return_prob: "none"
    generate_unk: False
    sacrebleu_cfg:
        tokenize: "13a"
        lowercase: False
```

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 our 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_uniform"
    embed_initializer: "normal"
    embed_init_weight: 0.1
    bias_initializer: "zeros"
    init_rnn_orthogonal: False
    lstm_forget_gate: 0.
    encoder:
        type: "recurrent"
        rnn_type: "lstm"
        embeddings:
            embedding_dim: 16
            scale: False
        hidden_size: 64
        bidirectional: True
        dropout: 0.1
        num_layers: 1
        activation: "tanh"
    decoder:
        type: "recurrent"
        rnn_type: "lstm"
        embeddings:
            embedding_dim: 16
            scale: False
        hidden_size: 64
        dropout: 0.1
        hidden_dropout: 0.1
        num_layers: 1
        activation: "tanh"
        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.

2.3.3 3. Training

Start

For training, run the following command:

```
python -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.

Note: If you encounter a file IO error, please consider to use the absolute path in the configuration.

Progress Tracking

The Log File

During training the Joey NMT 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:

```
2024-01-15 12:57:12,987 - INFO - joeynmt.training - Epoch   1, Step:      900, Batch␣
↪Loss:    21.149554, Batch Acc: 0.390395, Tokens per Sec:     9462, Lr: 0.001000
2024-01-15 12:57:16,549 - INFO - joeynmt.training - Epoch   1, Step:     1000, Batch␣
↪Loss:    35.254892, Batch Acc: 0.414826, Tokens per Sec:     9317, Lr: 0.001000
2024-01-15 12:57:16,550 - INFO - joeynmt.prediction - Predicting 1000 example(s)...␣
↪(Greedy decoding with min_output_length=1, max_output_length=30, return_prob='none',␣
↪generate_unk=True, repetition_penalty=-1, no_repeat_ngram_size=-1)
2024-01-15 12:57:29,506 - INFO - joeynmt.prediction - Generation took 12.9554[sec].
2024-01-15 12:57:29,548 - INFO - joeynmt.metrics -␣
↪nrefs:1|case:mixed|eff:no|tok:13a|smooth:exp|version:2.4.0
2024-01-15 12:57:29,549 - INFO - joeynmt.prediction - Evaluation result (greedy): bleu: ␣
↪22.52, loss:  29.77, ppl:   5.88, acc:   0.50, 0.0398[sec]
2024-01-15 12:57:29,549 - INFO - joeynmt.training - Hooray! New best validation result␣
↪[bleu]!
2024-01-15 12:57:29,576 - INFO - joeynmt.training - Checkpoint saved in reverse_model/
↪1000.ckpt.
2024-01-15 12:57:29,578 - INFO - joeynmt.training - Example #0
2024-01-15 12:57:29,578 - INFO - joeynmt.training -     Source:     10 43 37 32 6 9 25␣
↪36 21 29 16 7 18 27 30 46 37 15 7 48 18
2024-01-15 12:57:29,578 - INFO - joeynmt.training -     Reference:  18 48 7 15 37 46 30␣
↪27 18 7 16 29 21 36 25 9 6 32 37 43 10
2024-01-15 12:57:29,578 - INFO - joeynmt.training -     Hypothesis: 18 15 48 7 7 37 37␣
↪30 27 18 18 21 36 29 36 25 9 32 37
...
2024-01-15 13:02:15,428 - INFO - joeynmt.training - Epoch   5, total training loss: 3602.
↪67, num. of seqs: 40000, num. of tokens: 558505, 61.0933[sec]
2024-01-15 13:02:15,429 - INFO - joeynmt.training - Training ended after   5 epochs.
```

(continues on next page)

(continued from previous page)

```
2024-01-15 13:02:15,429 - INFO - joeynmt.training - Best validation result (greedy) at␣
↪step     7000:  95.42 bleu.
```

The training batch loss is logged every 100 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 22.52 (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.

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: 29.77000  acc: 0.50119    ppl: 5.88275    bleu: 22.51791  LR: 0.
↪00100000  *
Steps: 2000     loss: 25.81088  acc: 0.61057    ppl: 5.00362    bleu: 57.30290  LR: 0.
↪00100000  *
Steps: 3000     loss: 25.59565  acc: 0.71042    ppl: 4.86078    bleu: 83.38687  LR: 0.
↪00100000  *
Steps: 4000     loss: 19.88389  acc: 0.79269    ppl: 3.61883    bleu: 89.83186  LR: 0.
↪00100000  *
Steps: 5000     loss: 24.50622  acc: 0.76759    ppl: 4.37760    bleu: 89.38016  LR: 0.
↪00100000
```

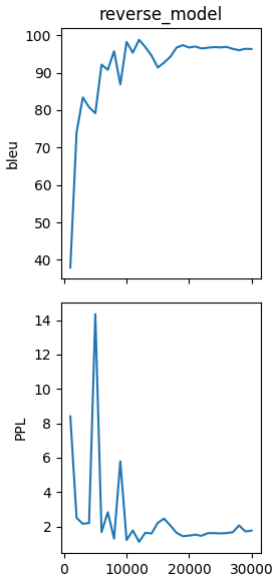
Models are saved whenever a new best validation score is reached, in `batch_no.ckpt`, where `batch_no` is the number of batches the model has been trained on so far. You can see when a checkpoint was saved by the asterisk at the end of the line in `validations.txt`. `best.ckpt` links to the checkpoint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

Learning Curves

Joey NMT provides a script plot_validations.py 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.

```
python scripts/plot_validations.py reverse_model --plot-values bleu ppl  --output-path␣
↪reverse_model/bleu-ppl.png
```

It should look like this:



Tensorboard

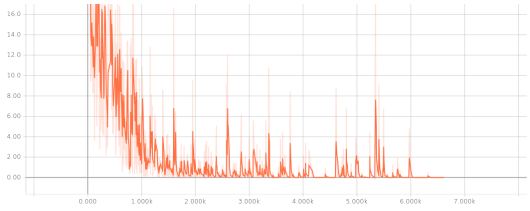
Joey NMT additionally uses Tensorboard to visualize training and validation curves and attention matrices during training. Launch Tensorboardlike this:

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

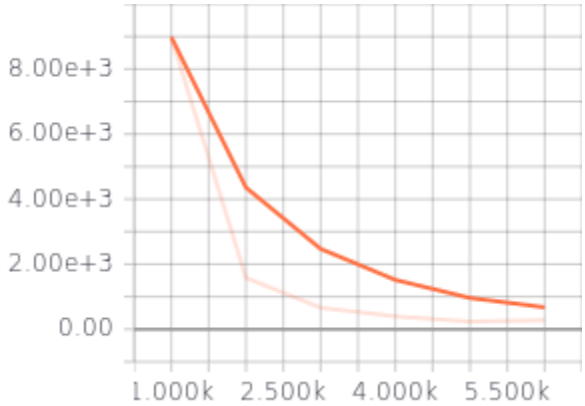
and then open the url (default: `localhost:6006`) with a browser.

You should see something like that:

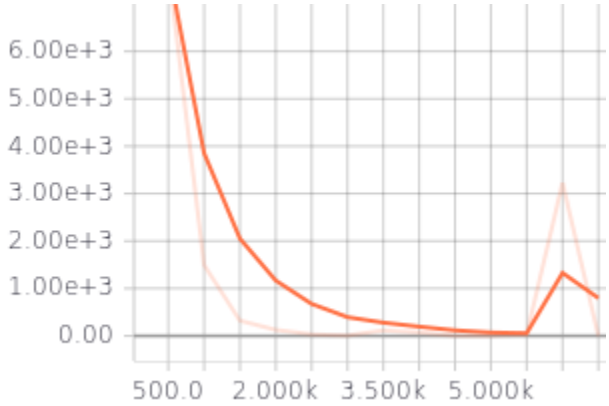
We can now inspect the training loss curves, both for individual batches



and for the whole training set:



and the validation loss:

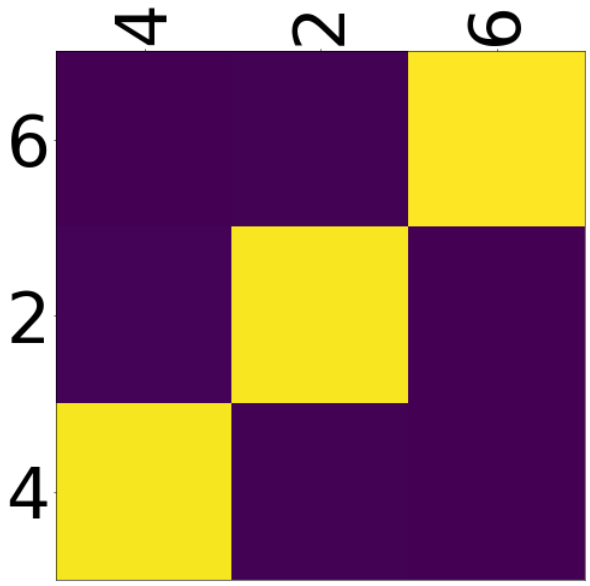


Looks good! Training and validation loss are decreasing, that means the model is doing well.

Attention Visualization

Attention scores often allow us a more visual inspection of what the model has learned. For every pair of source and target tokens, the model computes attention scores, so we can visualize this matrix. Joey NMT 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.

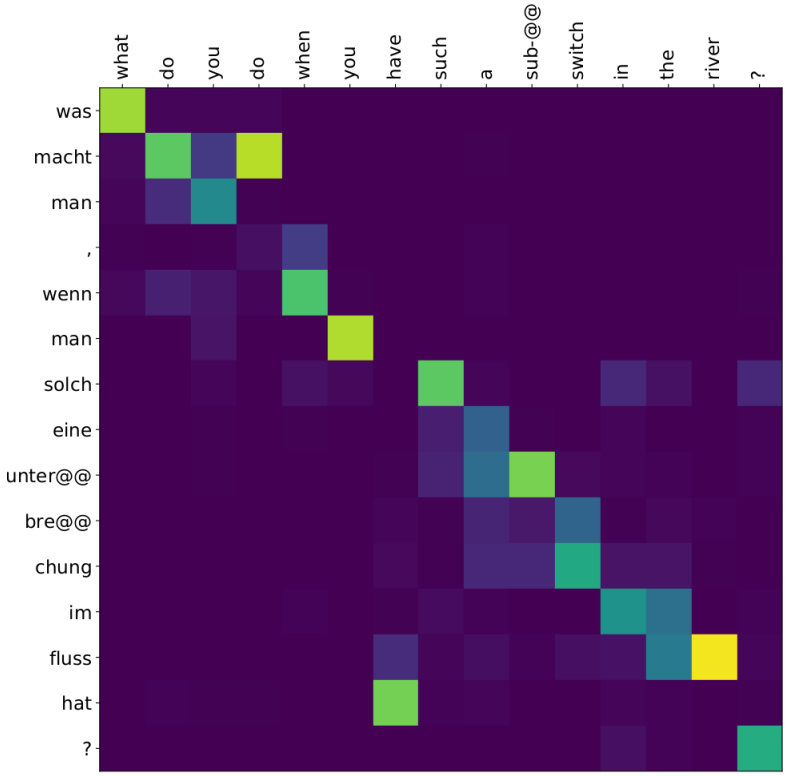
Here's an example, target tokens as columns and source tokens as rows:



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:



2.3.4 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.

Test Set Evaluation

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

```
python -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_metrics` 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:

```
2024-01-15 13:25:07,213 - INFO - joeynmt.prediction - Decoding on dev set... (device:␣
↪cuda, n_gpu: 1, use_ddp: False, fp16: True)
2024-01-15 13:25:07,213 - INFO - joeynmt.prediction - Predicting 1000 example(s)...␣
↪(Greedy decoding with min_output_length=1, max_output_length=30, return_prob='none',␣
↪generate_unk=True, repetition_penalty=-1, no_repeat_ngram_size=-1)
2024-01-15 13:25:20,203 - INFO - joeynmt.prediction - Generation took 12.9892[sec].
2024-01-15 13:25:20,301 - INFO - joeynmt.metrics - ␣
↪nrefs:1|case:mixed|eff:no|tok:13a|smooth:exp|version:2.4.0
2024-01-15 13:25:20,302 - INFO - joeynmt.prediction - Evaluation result (greedy): bleu: ␣
↪95.06, 0.0860[sec]
2024-01-15 13:25:20,302 - INFO - joeynmt.prediction - Decoding on test set... (device:␣
↪cuda, n_gpu: 1, use_ddp: False, fp16: True)
2024-01-15 13:25:20,302 - INFO - joeynmt.prediction - Predicting 1000 example(s)...␣
↪(Greedy decoding with min_output_length=1, max_output_length=30, return_prob='none',␣
↪generate_unk=True, repetition_penalty=-1, no_repeat_ngram_size=-1)
2024-01-15 13:25:32,532 - INFO - joeynmt.prediction - Generation took 12.2290[sec].
2024-01-15 13:25:32,725 - INFO - joeynmt.metrics - ␣
↪nrefs:1|case:mixed|eff:no|tok:13a|smooth:exp|version:2.4.0
2024-01-15 13:25:32,725 - INFO - joeynmt.prediction - Evaluation result (greedy): bleu: ␣
↪95.19, 0.1821[sec]
```

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.

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
python -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
```

Interactive Translation

If you just want to try a few examples, run

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

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

Let's try a challenging long one:

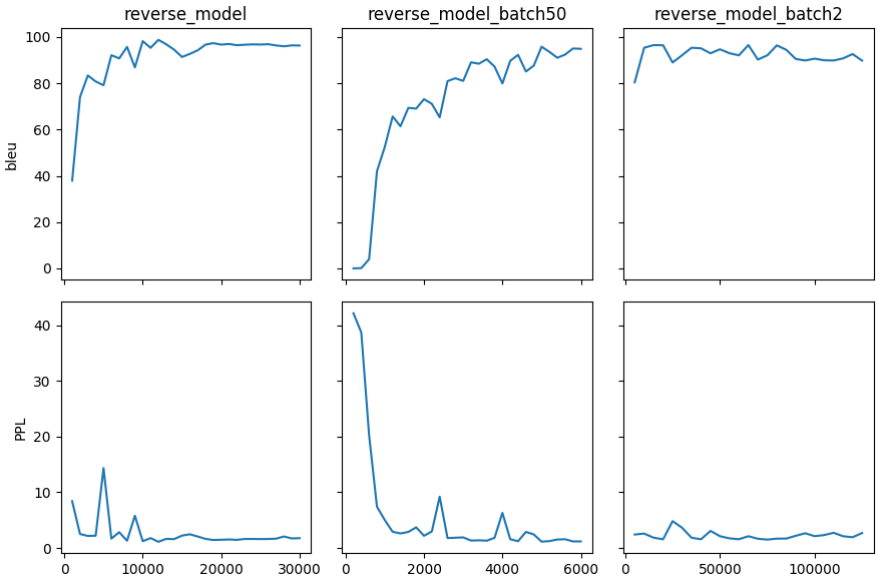
```
Please enter a source sentence:
1 23 23 43 34 2 2 2 2 2 4 5 32 47 47 47 21 20 0 10 10 10 10 10 8 7 33 36 37
Joey NMT:
33 10 10 37 10 10 0 20 21 47 47 47 32 5 4 2 2 2 2 2 2 34 43 23 1
```

Warning: Interactive `translate` mode doesn't work with Multi-GPU. Please run it on single GPU or CPU.

2.3.5 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. In 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 noticeably 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 possibly 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.

2.3.6 6. What's next?

If you want to implement something new in Joey NMT or dive a bit deeper, you should take a look at the *Module Overview* and explore the *API Documentation*.

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.

2.4 Benchmarks

We provide several pretrained models with their benchmark results.

2.4.1 JoeyNMT v2.x

IWSLT14 de/en/fr multilingual

We trained this multilingual model with JoeyNMT v2.3.0 using DDP.

Direction	Architecture	Tokenizer	dev	test	#params	download
en->de	Transformer	sentencepiece	-	28.88	200M	iwslt14_prompt
de->en			-	35.28		
en->fr			-	38.86		
fr->en			-	40.35		

sacrebleu signature: *nrefs:1|case:lc|eff:no|tok:13a|smooth:exp|version:2.4.0*

WMT14 ende / deen

We trained the models with JoeyNMT v2.1.0 from scratch.

cf) wmt14 deen leaderboard in paperswithcode

Direction	Architecture	Tokenizer	dev	test	#params	download
en->de	Transformer	sentencepiece	24.36	24.38	60.5M	wmt14_ende.tar.gz (766M)
de->en	Transformer	sentencepiece	30.60	30.51	60.5M	wmt14_deen.tar.gz (766M)

sacrebleu signature: *nrefs:1|case:mixed|eff:no|tok:13a|smooth:exp|version:2.2.0*

2.4.2 JoeyNMT v1.x

Warning: The following models are trained with JoeynNMT v1.x, and decoded with Joey NMT v2.0. See `config_v1.yaml` and `config_v2.yaml` in the linked tar.gz, respectively. Joey NMT v1.x benchmarks are archived here.

IWSLT14 deen

Pre-processing with Moses decoder tools as in this script.

Direction	Architecture	Tokenizer	dev	test	#params	download
de->en	RNN	subword-nmt	31.77	30.74	61M	rnn_iwslt14_deen_bpe.tar.gz (672M)
de->en	Transformer	subword-nmt	34.53	33.73	19M	transformer_iwslt14_deen_bpe.tar.gz (221M)

sacrebleu signature: *nrefs:1|case:lc|eff:no|tok:13a|smooth:exp|version:2.0.0*

Note: For interactive translate mode, you should specify `pretokenizer: "moses"` in both src's and trg's `tokenizer_cfg`, so that you can input raw sentences. Then `MosesTokenizer` and `MosesDetokenizer` will be applied internally. For test mode, we used the preprocessed texts as input and set `pretokenizer: "none"` in the config.

Masakhane JW300 afen / enaf

We picked the pretrained models and configs (bpe codes file etc.) from masakhane.io.

Direction	Architecture	Tokenizer	dev	test	#params	download
af->en	Transformer	subword-nmt	-	57.70	46M	transformer_jw300_afen.tar.gz (525M)
en->af	Transformer	subword-nmt	47.24	47.31	24M	transformer_jw300_enaf.tar.gz (285M)

sacrebleu signature: *nrefs:1|case:mixed|eff:no|tok:intl|smooth:exp|version:2.0.0*

JParaCrawl enja / jaen

For training, we split JparaCrawl v2 into train and dev set and trained a model on them. Please check the preprocessing script here. We tested then on kftt test set and wmt20 test set, respectively.

Direction	Architecture	Tokenizer	kftt	wmt20	#params	download
af->en	Transformer	sentencepiece	17.66	14.31	225M	jparacrawl_enja.tar.gz (2.3GB)
en->af	Transformer	sentencepiece	14.97	11.49	221M	jparacrawl_jaen.tar.gz (2.2GB)

sacrebleu signature:

- en->ja: *nrefs:1|case:mixed|eff:no|tok:ja-mecab-0.996-IPA|smooth:exp|version:2.0.0*
- ja->en: *nrefs:1|case:mixed|eff:no|tok:intl|smooth:exp|version:2.0.0*

(Note: In wmt20 test set, *newstest2020-enja* has 1000 examples, *newstest2020-jaen* has 993 examples.)

2.5 Module 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 detailed documentation of the API, go to *API Documentation*.

2.5.1 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 is 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.

2.5.2 Training Management

The training process is managed by the *TrainManager :joeynmt: `training.py*. 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.

When "batch_multiplier" > 0 is set, the gradients are accumulated before the model parameters are updated. In the batch loop, we call `loss.backward()` for each batch, but `optimizer.step()` is called every (batch_multiplier)-th time steps only, and then the accumulated gradients (`model.zero_grad()`) are reset.

```
for epoch in range(epochs):
    model.zero_grad()
    epoch_loss = 0.0
    batch_loss = 0.0
    for i, batch in enumerate(train_iter):

        # gradient accumulation:
        # loss.backward() will be called in _train_step()
        batch_loss += _train_step(batch)

        if (i + 1) % args.batch_multiplier == 0:
            optimizer.step()     # update!
            model.zero_grad()    # reset gradients
            steps += 1           # increment counter

            epoch_loss += batch_loss  # accumulate batch loss
            batch_loss = 0            # reset batch loss

    # leftovers are just ignored.
    # (see `drop_last` arg in train_iter.batch_sampler)
```

2.5.3 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.

2.5.4 Data Handling

Data Loading

At the current state, we support the following input data formats: - plain txt: one-sentence-per-line; requires language name in the file extension. - tsv: requires header row with src and trg language names. - Huggingface's datasets: requires *translation* field. - (stdin for interactive translation cli)

In the timing of data loading, we only apply preprocess operations such as lowercasing, punctuation deletion, etc. if needed. Tokenization is applied on-the-fly when a batch is constructed during training/prediction. See `get_item()` in BaseDataset class for details.

Mini-Batching

The dataloader samples data points from the corpus and constructs a batch with batch.py. The instances within each mini-batch are sorted by length, so that we can make use of PyTorch's efficient RNN sequence padding and packing functions. For **inference**, we keep track of the original order so that we can revert the order of the model outputs.

Joey NMT v2.3 (or greater) supports DataParallel and DistributedDataParallel in PyTorch. Please refer to external documentation i.e. PyTorch DDP tutorial to learn how those modules dispatch the minibatchs across multiple GPU devices.

Vocabulary

For the creation of the vocabulary (vocabulary.py, all tokens occurring in the training set are collected, sorted and optionally filtered by frequency and then cut off as specified in the configuration. By default, it creates src language vocab and trg language vocab separately. If you want to use joint vocabulary, you need to create vocabulary (build_vocab.py) before you start training. 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.

Token granularity should be specified in the "data" section of the configuration file. Currently, JoeyNMT supports word-based, character-based models and sub-word models with byte-pair-encodings (BPE) as learned with subword-nmt or sentencepiece.

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.

2.5.5 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.

2.6 API Documentation

2.6.1 Module contents

- genindex
- modindex
- search

2.6.2 Submodules

joeynmt.attention module

Attention modules

class joeynmt.attention.**AttentionMechanism**(**args*, ***kwargs*)

Bases: Module

Base attention class

forward(**inputs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class joeynmt.attention.**BahdanauAttention**(*hidden_size: int = 1*, *key_size: int = 1*, *query_size: int = 1*)

Bases: *AttentionMechanism*

Implements Bahdanau (MLP) attention

Section A.1.2 in https://arxiv.org/abs/1409.0473.

compute_proj_keys(*keys: Tensor*) → None

Compute the projection of the keys. Is efficient if pre-computed before receiving individual queries.

Parameters
keys –

Returns

compute_proj_query(*query: Tensor*)

Compute the projection of the query.

Parameters
query –

Returns

forward(*query: Tensor*, *mask: Tensor*, *values: Tensor*) → Tuple[Tensor, Tensor]

Bahdanau MLP attention forward pass.

Parameters

- **query** – the item (decoder state) to compare with the keys/memory, shape (batch_size, 1, decoder.hidden_size)
- **mask** – mask out keys position (0 in invalid positions, 1 else), shape (batch_size, 1, src_length)
- **values** – values (encoder states), shape (batch_size, src_length, encoder.hidden_size)

Returns

- context vector of shape (batch_size, 1, value_size),
- attention probabilities of shape (batch_size, 1, src_length)

class joeynmt.attention.**LuongAttention**(*hidden_size: int = 1*, *key_size: int = 1*)

Bases: *AttentionMechanism*

Implements Luong (bilinear / multiplicative) attention.

Eq. 8 ("general") in http://aclweb.org/anthology/D15-1166.

compute_proj_keys(*keys: Tensor*) → None

Compute the projection of the keys and assign them to *self.proj_keys*. This pre-computation is efficiently done for all keys before receiving individual queries.

Parameters

keys – shape (batch_size, src_length, encoder.hidden_size)

forward(*query: Tensor*, *mask: Tensor*, *values: Tensor*) → Tuple[Tensor, Tensor]

Luong (multiplicative / bilinear) attention forward pass. Computes context vectors and attention scores for a given query and all masked values and returns them.

Parameters

- **query** – the item (decoder state) to compare with the keys/memory, shape (batch_size, 1, decoder.hidden_size)
- **mask** – mask out keys position (0 in invalid positions, 1 else), shape (batch_size, 1, src_length)
- **values** – values (encoder states), shape (batch_size, src_length, encoder.hidden_size)

Returns

- context vector of shape (batch_size, 1, value_size),
- attention probabilities of shape (batch_size, 1, src_length)

joeynmt.batch module

Implementation of a mini-batch.

class joeynmt.batch.**Batch**(*src: Tensor*, *src_length: Tensor*, *src_prompt_mask: Tensor | None*, *trg: Tensor | None*, *trg_prompt_mask: Tensor | None*, *indices: Tensor*, *device: device*, *pad_index: int*, *eos_index: int*, *is_train: bool = True*)

Bases: object

Object for holding a batch of data with mask during training. Input is yielded from *collate_fn()* called by torch.data.utils.DataLoader.

normalize(*tensor: Tensor*, *normalization: str = 'none'*, *n_gpu: int = 1*, *n_accumulation: int = 1*) → Tensor

Normalizes batch tensor (i.e. loss). Takes sum over multiple gpus, divides by nseqs or ntokens, divide by n_gpu, then divide by n_accumulation.

Parameters

- **tensor** – (Tensor) tensor to normalize, i.e. batch loss
- **normalization** – (str) one of {*batch*, *tokens*, *none*}
- **n_gpu** – (int) the number of gpus
- **n_accumulation** – (int) the number of gradient accumulation

Returns

normalized tensor

static score(*log_probs: Tensor*, *trg: Tensor*, *pad_index: int*) → ndarray

Look up the score of the trg token (ground truth) in the batch

sort_by_src_length() → List[int]

Sort by src length (descending) and return index to revert sort

Returns
list of indices

joeynmt.builders module

Collection of builder functions

class joeynmt.builders.**BaseScheduler**(*optimizer: Optimizer*)

Bases: object

Base LR Scheduler decay at "step"

load_state_dict(*state_dict*)

Given a state_dict, this function loads scheduler's state

state_dict()

Returns dictionary of values necessary to reconstruct scheduler

step(*step*)

Update parameters and rate

class joeynmt.builders.**NoamScheduler**(*hidden_size: int*, *optimizer: Optimizer*, *factor: float = 1.0*, *warmup: int = 4000*)

Bases: *BaseScheduler*

The Noam learning rate scheduler used in "Attention is all you need" See Eq. 3 in https://arxiv.org/abs/1706.03762

load_state_dict(*state_dict*)

Given a state_dict, this function loads scheduler's state

state_dict()

Returns dictionary of values necessary to reconstruct scheduler

class joeynmt.builders.**WarmupExponentialDecayScheduler**(*optimizer: Optimizer*, *peak_rate: float = 0.001*, *decay_length: int = 10000*, *warmup: int = 4000*, *decay_rate: float = 0.5*, *min_rate: float = 1e-05*)

Bases: *BaseScheduler*

A learning rate scheduler similar to Noam, but modified: Keep the warm up period but make it so that the decay rate can be tuneable. The decay is exponential up to a given minimum rate.

load_state_dict(*state_dict*)

Given a state_dict, this function loads scheduler's state

state_dict()

Returns dictionary of values necessary to reconstruct scheduler

class joeynmt.builders.**WarmupInverseSquareRootScheduler**(*optimizer: Optimizer*, *peak_rate: float = 0.001*, *warmup: int = 10000*, *min_rate: float = 1e-05*)

Bases: *BaseScheduler*

Decay the LR based on the inverse square root of the update number. In the warmup phase, we linearly increase the learning rate. After warmup, we decrease the learning rate as follows: ` `decay_factor = peak_rate * sqrt(warmup)` `# constant value` `lr = decay_factor / sqrt(step)` ` cf.) https://github.com/pytorch/fairseq/blob/main/fairseq/optim/lr_scheduler/inverse_square_root_schedule.py

load_state_dict(*state_dict*)

Given a state_dict, this function loads scheduler's state

state_dict()

Returns dictionary of values necessary to reconstruct scheduler

joeynmt.builders.**build_activation**(*activation: str = 'relu'*) → Callable

Returns the activation function

joeynmt.builders.**build_gradient_clipper**(*cfg: Dict*) → Callable | None

Define the function for gradient clipping as specified in configuration. If not specified, returns None.

Current options:

- **"clip_grad_val": clip the gradients if they exceed this value,**
  see *torch.nn.utils.clip_grad_value_*
- **"clip_grad_norm": clip the gradients if their norm exceeds this value,**
  see *torch.nn.utils.clip_grad_norm_*

Parameters
cfg – dictionary with training configurations

Returns
clipping function (in-place) or None if no gradient clipping

joeynmt.builders.**build_optimizer**(*cfg: Dict*, *parameters: Generator*) → Optimizer

Create an optimizer for the given parameters as specified in config.

Except for the weight decay and initial learning rate, default optimizer settings are used.

Currently supported configuration settings for "optimizer":

- "sgd" (default): see *torch.optim.SGD*
- "adam": see *torch.optim.adam*
- "adamw": see *torch.optim.adamw*
- "adagrad": see *torch.optim.adagrad*
- "adadelta": see *torch.optim.adadelta*
- "rmsprop": see *torch.optim.RMSprop*

The initial learning rate is set according to "learning_rate" in the config. The weight decay is set according to "weight_decay" in the config. If they are not specified, the initial learning rate is set to 3.0e-4, the weight decay to 0.

Note that the scheduler state is saved in the checkpoint, so if you load a model for further training you have to use the same type of scheduler.

Parameters

- **cfg** – configuration dictionary
- **parameters** –

Returns
optimizer

joeynmt.builders.**build_scheduler**(*cfg: Dict*, *optimizer: Optimizer*, *scheduler_mode: str*, *hidden_size: int = 0*)

Create a learning rate scheduler if specified in config and determine when a scheduler step should be executed.

Current options:

- "plateau": see *torch.optim.lr_scheduler.ReduceLROnPlateau*
- "decaying": see *torch.optim.lr_scheduler.StepLR*
- "exponential": see *torch.optim.lr_scheduler.ExponentialLR*
- "noam": see *joeynmt.builders.NoamScheduler*
- "warmupexponentialdecay": see *joeynmt.builders.WarmupExponentialDecayScheduler*
- "warmupinversesquareroot": see *joeynmt.builders.WarmupInverseSquareRootScheduler*

If no scheduler is specified, returns (None, None) which will result in a constant learning rate.

Parameters

- **cfg** – training configuration
- **optimizer** – optimizer for the scheduler, determines the set of parameters which the scheduler sets the learning rate for
- **scheduler_mode** – "min" or "max", depending on whether the validation score should be minimized or maximized. Only relevant for "plateau".
- **hidden_size** – encoder hidden size (required for NoamScheduler)

Returns

- scheduler: scheduler object,
- scheduler_step_at: either "validation", "epoch", "step" or "none"

joeynmt.config module

Module for configuration

This can only be a temporary solution. TODO: Consider better configuration and validation cf. https://github.com/joeynmt/joeynmt/issues/196

class joeynmt.config.**BaseConfig**(*name*, *joeynmt_version*, *model_dir*, *device*, *n_gpu*, *num_workers*, *autocast*, *seed*, *train*, *test*, *data*, *model*)

Bases: tuple

autocast: Dict

Alias for field number 6

data: Dict

Alias for field number 10

device: device

Alias for field number 3

joeynmt_version: str | None

Alias for field number 1

model: Dict

Alias for field number 11

model_dir: Path

Alias for field number 2

n_gpu: int

Alias for field number 4

name: str

Alias for field number 0

num_workers: int

Alias for field number 5

seed: int

Alias for field number 7

test: *TestConfig*

Alias for field number 9

train: *TrainConfig*

Alias for field number 8

exception joeynmt.config.**ConfigurationError**

Bases: Exception

Custom exception for misspecifications of configuration

class joeynmt.config.**TestConfig**(*load_model*, *batch_size*, *batch_type*, *max_output_length*, *min_output_length*, *eval_metrics*, *sacrebleu_cfg*, *beam_size*, *beam_alpha*, *n_best*, *return_attention*, *return_prob*, *generate_unk*, *repetition_penalty*, *no_repeat_ngram_size*)

Bases: tuple

batch_size: int

Alias for field number 1

batch_type: str

Alias for field number 2

beam_alpha: int

Alias for field number 8

beam_size: int

Alias for field number 7

eval_metrics: List[str]

Alias for field number 5

generate_unk: bool

Alias for field number 12

load_model: Path | None

Alias for field number 0

max_output_length: int

Alias for field number 3

min_output_length: int

Alias for field number 4

n_best: int

Alias for field number 9

no_repeat_ngram_size: int

Alias for field number 14

repetition_penalty: float

Alias for field number 13

return_attention: bool

Alias for field number 10

return_prob: str

Alias for field number 11

sacrebleu_cfg: Dict | None

Alias for field number 6

class joeynmt.config.**TrainConfig**(*load_model*, *load_encoder*, *load_decoder*, *loss*, *normalization*, *label_smoothing*, *optimizer*, *adam_betas*, *learning_rate*, *learning_rate_min*, *learning_rate_factor*, *learning_rate_warmup*, *scheduling*, *patience*, *decrease_factor*, *weight_decay*, *clip_grad_norm*, *clip_grad_val*, *keep_best_ckpts*, *logging_freq*, *validation_freq*, *print_valid_sents*, *early_stopping_metric*, *minimize_metric*, *shuffle*, *epochs*, *max_updates*, *batch_size*, *batch_type*, *batch_multiplier*, *reset_best_ckpt*, *reset_scheduler*, *reset_optimizer*, *reset_iter_state*)

Bases: tuple

adam_betas: List[float]

Alias for field number 7

batch_multiplier: int

Alias for field number 29

batch_size: int

Alias for field number 27

batch_type: str

Alias for field number 28

clip_grad_norm: float | None

Alias for field number 16

clip_grad_val: float | None

Alias for field number 17

decrease_factor: float

Alias for field number 14

early_stopping_metric: str
Alias for field number 22

epochs: int
Alias for field number 25

keep_best_ckpts: int
Alias for field number 18

label_smoothing: float
Alias for field number 5

learning_rate: float
Alias for field number 8

learning_rate_factor: int
Alias for field number 10

learning_rate_min: float
Alias for field number 9

learning_rate_warmup: int
Alias for field number 11

load_decoder: Path | None
Alias for field number 2

load_encoder: Path | None
Alias for field number 1

load_model: Path | None
Alias for field number 0

logging_freq: int
Alias for field number 19

loss: str
Alias for field number 3

max_updates: int
Alias for field number 26

minimize_metric: bool
Alias for field number 23

normalization: str
Alias for field number 4

optimizer: str
Alias for field number 6

patience: int
Alias for field number 13

print_valid_sents: List[int]
Alias for field number 21

reset_best_ckpt: bool
Alias for field number 30

reset_iter_state: bool
Alias for field number 33

reset_optimizer: bool
Alias for field number 32

reset_scheduler: bool
Alias for field number 31

scheduling: str | None
Alias for field number 12

shuffle: bool
Alias for field number 24

validation_freq: int
Alias for field number 20

weight_decay: float
Alias for field number 15

joeynmt.config.**load_config**(*cfg_file: str = 'configs/default.yaml'*) → Dict
Loads and parses a YAML configuration file.

Parameters
cfg_file – path to YAML configuration file

Returns
configuration dictionary

joeynmt.config.**log_config**(*cfg: Dict*, *prefix: str = 'cfg'*) → None
Print configuration to console log.

Parameters

- **cfg** – configuration to log
- **prefix** – prefix for logging

joeynmt.config.**parse_global_args**(*cfg: Dict = None*, *rank: int = 0*, *mode: str = 'train'*) → *BaseConfig*
Parse and validate global args

Parameters

- **cfg** – config specified in yaml file
- **rank** –
- **mode** –

joeynmt.config.**parse_test_args**(*cfg: Dict = None*, *mode: str = 'test'*) → *TestConfig*
Parse and validate test args

Parameters

- **cfg** – *testing* section in config yaml
- **mode** –

joeynmt.config.**parse_train_args**(*cfg: Dict = None*, *mode: str = 'train'*) → *TrainConfig*
Parse and validate train args

Parameters

- **cfg** – *training* section in config yaml
- **mode** –

joeynmt.config.**set_validation_args**(*args:* TestConfig) → *TestConfig*

Config for validation

Parameters
args – *testing* section in config yaml

joeynmt.data module

Data module

joeynmt.data.**load_data**(*cfg: Dict*, *datasets: list = None*) → Tuple[*Vocabulary*, *Vocabulary*, *BaseDataset* | None, *BaseDataset* | None, *BaseDataset* | None]

Load train, dev and optionally test data as specified in configuration. Vocabularies are created from the training set with a limit of *voc_limit* tokens and a minimum token frequency of *voc_min_freq* (specified in the configuration dictionary).

The training data is filtered to include sentences up to *max_length* on source and target side.

If you set *random_{train|dev}_subset*, a random selection of this size is used from the {train|development} set instead of the full {train|development} set.

Parameters

- **cfg** – configuration dictionary for data ("data" part of config file)
- **datasets** – list of dataset names to load

Returns

- src_vocab: source vocabulary
- trg_vocab: target vocabulary
- train_data: training dataset
- dev_data: development dataset
- test_data: test dataset if given, otherwise None

joeynmt.datasets module

Dataset module

class joeynmt.datasets.**BaseDataset**(*path: str*, *src_lang: str*, *trg_lang: str*, *split: str = 'train'*, *has_trg: bool = False*, *has_prompt: Dict[str, bool] = None*, *tokenizer: Dict[str,* BasicTokenizer*] = None*, *sequence_encoder: Dict[str, Callable] = None*, *random_subset: int = -1*)

Bases: `Dataset`

BaseDataset which loads and looks up data. - holds pointer to tokenizers, encoding functions.

Parameters

- **path** – path to data directory
- **src_lang** – source language code, i.e. *en*
- **trg_lang** – target language code, i.e. *de*

- **has_trg** – bool indicator if trg exists
- **has_prompt** – bool indicator if prompt exists
- **split** – bool indicator for train set or not
- **tokenizer** – tokenizer objects
- **sequence_encoder** – encoding functions

collate_fn(*batch: List[Tuple]*, *pad_index: int*, *eos_index: int*, *device: device = device(type='cpu')*) → *Batch*

Custom collate function. See https://pytorch.org/docs/stable/data.html#dataloader-collate-fn for details. Please override the batch class here. (not in TrainManager)

Parameters

- **batch** –
- **pad_index** –
- **eos_index** –
- **device** –

Returns
joeynmt batch object

get_item(*idx: int*, *lang: str*, *is_train: bool = None*) → List[str]

seek one src/trg item of the given index.

- tokenization is applied here.
- length-filtering, bpe-dropout etc also triggered if self.split == "train"

get_list(*lang: str*, *tokenized: bool = False*, *subsampled: bool = True*) → List[str] | List[List[str]]

get data column-wise.

load_data(*path: Path*, ***kwargs*) → Any

load data

- preprocessing (lowercasing etc) is applied here.

lookup_item(*idx: int*, *lang: str*) → Tuple[str, str]

make_iter(*batch_size: int*, *batch_type: str = 'sentence'*, *seed: int = 42*, *shuffle: bool = False*, *num_workers: int = 0*, *pad_index: int = 1*, *eos_index: int = 3*, *device: device = device(type='cpu')*, *generator_state: Tensor = None*) → DataLoader

Returns a torch DataLoader for a torch Dataset. (no bucketing)

Parameters

- **batch_size** – size of the batches the iterator prepares
- **batch_type** – measure batch size by sentence count or by token count
- **seed** – random seed for shuffling
- **shuffle** – whether to shuffle the order of sequences before each epoch (for testing, no effect even if set to True; generator is still used for random subsampling, but not for permutation!)
- **num_workers** – number of cpus for multiprocessing

- **pad_index** –
- **eos_index** –
- **device** –
- **generator_state** –

Returns
torch DataLoader

reset_indices(*random_subset: int = None*)

property src: List[str]
get detokenized preprocessed data in src language.

property trg: List[str]
get detokenized preprocessed data in trg language.

class joeynmt.datasets.**BaseHuggingfaceDataset**(*path: str*, *src_lang: str*, *trg_lang: str*, *has_trg: bool = True*, *has_prompt: Dict[str, bool] = None*, *tokenizer: Dict[str, BasicTokenizer] = None*, *sequence_encoder: Dict[str, Callable] = None*, *random_subset: int = -1*, ***kwargs*)

Bases: *BaseDataset*

Wrapper for Huggingface's dataset object cf.) https://huggingface.co/docs/datasets

COLUMN_NAME = 'sentence'

get_list(*lang: str*, *tokenized: bool = False*, *subsampled: bool = True*) → List[str] | List[List[str]]
get data column-wise.

load_data(*path: str*, ***kwargs*) → Any

load data

- preprocessing (lowercasing etc) is applied here.

lookup_item(*idx: int*, *lang: str*) → Tuple[str, str]

class joeynmt.datasets.**HuggingfaceTranslationDataset**(*path: str*, *src_lang: str*, *trg_lang: str*, *has_trg: bool = True*, *has_prompt: Dict[str, bool] = None*, *tokenizer: Dict[str, BasicTokenizer] = None*, *sequence_encoder: Dict[str, Callable] = None*, *random_subset: int = -1*, ***kwargs*)

Bases: *BaseHuggingfaceDataset*

Wrapper for Huggingface's *datasets.features.Translation* class cf.) https://github.com/huggingface/datasets/blob/master/src/datasets/features/translation.py

COLUMN_NAME = 'translation'

load_data(*path: str*, ***kwargs*) → Any

load data

- preprocessing (lowercasing etc) is applied here.

class joeynmt.datasets.**PlaintextDataset**(*path: str*, *src_lang: str*, *trg_lang: str*, *split: str = 'train'*, *has_trg: bool = False*, *has_prompt: Dict[str, bool] = None*, *tokenizer: Dict[str, BasicTokenizer] = None*, *sequence_encoder: Dict[str, Callable] = None*, *random_subset: int = -1*, ***kwargs*)

Bases: *BaseDataset*

PlaintextDataset which stores plain text pairs. - used for text file data in the format of one sentence per line.

get_list(*lang: str*, *tokenized: bool = False*, *subsampled: bool = True*) → List[str] | List[List[str]]

Return list of preprocessed sentences in the given language. (not length-filtered, no bpe-dropout)

load_data(*path: str*, ***kwargs*) → Any

load data

- preprocessing (lowercasing etc) is applied here.

lookup_item(*idx: int*, *lang: str*) → Tuple[str, str]

class joeynmt.datasets.**SentenceBatchSampler**(*sampler: Sampler*, *batch_size: int*, *drop_last: bool*, *seed: int*)

Bases: BatchSampler

Wraps another sampler to yield a mini-batch of indices based on num of instances. An instance longer than dataset.max_len will be filtered out.

Parameters

- **sampler** – Base sampler. Can be any iterable object
- **batch_size** – Size of mini-batch.
- **drop_last** – If *True*, the sampler will drop the last batch if its size would be less than *batch_size*

get_state()

property num_samples: int

Returns number of samples in the dataset. This may change during sampling.

Note: len(dataset) won't change during sampling.
Use len(dataset) instead, to retrieve the original dataset length.

reset() → None

set_seed(*seed: int*) → None

set_state(*state*) → None

class joeynmt.datasets.**StreamDataset**(*path: str*, *src_lang: str*, *trg_lang: str*, *split: str = 'test'*, *has_trg: bool = False*, *has_prompt: Dict[str, bool] = None*, *tokenizer: Dict[str, BasicTokenizer] = None*, *sequence_encoder: Dict[str, Callable] = None*, *random_subset: int = -1*, ***kwargs*)

Bases: *BaseDataset*

StreamDataset which interacts with stream inputs. - called by *translate()* func in *prediction.py*.

lookup_item(*idx: int*, *lang: str*) → Tuple[str, str]

reset_cache()

set_item(*src_line: str*, *trg_line: str | None = None*, *src_prompt: str | None = None*, *trg_prompt: str | None = None*) → None

Set input text to the cache.

Parameters

- **src_line** – (non-empty) str
- **trg_line** – Optional[str]
- **src_prompt** – Optional[str]
- **trg_prompt** – Optional[str]

class joeynmt.datasets.**TokenBatchSampler**(*sampler: Sampler*, *batch_size: int*, *drop_last: bool*, *seed: int*)

Bases: *SentenceBatchSampler*

Wraps another sampler to yield a mini-batch of indices based on num of tokens (incl. padding). An instance longer than dataset.max_len or shorter than dataset.min_len will be filtered out. * no bucketing implemented

Warning: In DDP, we shouldn't use TokenBatchSampler for prediction, because we cannot ensure that the data points will be distributed evenly across devices. *ddp_merge()* (*dist.all_gather()*) called in *predict()* can get stuck.

Parameters

- **sampler** – Base sampler. Can be any iterable object
- **batch_size** – Size of mini-batch.
- **drop_last** – If *True*, the sampler will drop the last batch if its size would be less than *batch_size*

class joeynmt.datasets.**TsvDataset**(*path: str*, *src_lang: str*, *trg_lang: str*, *split: str = 'train'*, *has_trg: bool = False*, *has_prompt: Dict[str, bool] = None*, *tokenizer: Dict[str, BasicTokenizer] = None*, *sequence_encoder: Dict[str, Callable] = None*, *random_subset: int = -1*, ***kwargs*)

Bases: *BaseDataset*

TsvDataset which handles data in tsv format. - file_name should be specified without extension *.tsv* - needs src_lang and trg_lang (i.e. *en*, *de*) in header. see: test/data/toy/dev.tsv

get_list(*lang: str*, *tokenized: bool = False*, *subsampled: bool = True*) → List[str] | List[List[str]]

get data column-wise.

load_data(*path: str*, ***kwargs*) → Any

load data

- preprocessing (lowercasing etc) is applied here.

lookup_item(*idx: int*, *lang: str*) → Tuple[str, str]

joeynmt.datasets.**build_dataset**(*dataset_type: str*, *path: str*, *src_lang: str*, *trg_lang: str*, *split: str*, *tokenizer: Dict = None*, *sequence_encoder: Dict = None*, *has_prompt: Dict = None*, *random_subset: int = -1*, ***kwargs*)

Builds a dataset.

Parameters

- **dataset_type** – (str) one of {*plain*, *tsv*, *stream*, *huggingface*}
- **path** – (str) either a local file name or dataset name to download from remote
- **src_lang** – (str) language code for source
- **trg_lang** – (str) language code for target
- **split** – (str) one of {*train*, *dev*, *test*}
- **tokenizer** – tokenizer objects for both source and target
- **sequence_encoder** – encoding functions for both source and target
- **has_prompt** – prompt indicators
- **random_subset** – (int) number of random subset; -1 means no subsampling

Returns
loaded Dataset

joeynmt.decoders module

Various decoders

class joeynmt.decoders.**Decoder**(**args*, ***kwargs*)

Bases: Module

Base decoder class

property output_size

Return the output size (size of the target vocabulary)

Returns

class joeynmt.decoders.**RecurrentDecoder**(*rnn_type: str = 'gru'*, *emb_size: int = 0*, *hidden_size: int = 0*, *encoder: Encoder = None*, *attention: str = 'bahdanau'*, *num_layers: int = 1*, *vocab_size: int = 0*, *dropout: float = 0.0*, *emb_dropout: float = 0.0*, *hidden_dropout: float = 0.0*, *init_hidden: str = 'bridge'*, *input_feeding: bool = True*, *freeze: bool = False*, ***kwargs*)

Bases: *Decoder*

A conditional RNN decoder with attention.

forward(*trg_embed: Tensor*, *encoder_output: Tensor*, *encoder_hidden: Tensor*, *src_mask: Tensor*, *unroll_steps: int*, *hidden: Tensor = None*, *prev_att_vector: Tensor = None*, ***kwargs*) → Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]

Unroll the decoder one step at a time for *unroll_steps* steps. For every step, the *_forward_step* function is called internally.

During training, the target inputs (*trg_embed') are already known for the full sequence, so the full unrol is done. In this case, `hidden* and *prev_att_vector* are None.

For inference, this function is called with one step at a time since embedded targets are the predictions from the previous time step. In this case, *hidden* and *prev_att_vector* are fed from the output of the previous call of this function (from the 2nd step on).

src_mask is needed to mask out the areas of the encoder states that should not receive any attention, which is everything after the first <eos>.

The *encoder_output* are the hidden states from the encoder and are used as context for the attention.

The *encoder_hidden* is the last encoder hidden state that is used to initialize the first hidden decoder state (when *self.init_hidden_option* is "bridge" or "last").

Parameters

- **trg_embed** – embedded target inputs, shape (batch_size, trg_length, embed_size)
- **encoder_output** – hidden states from the encoder, shape (batch_size, src_length, encoder.output_size)
- **encoder_hidden** – last state from the encoder, shape (batch_size, encoder.output_size)
- **src_mask** – mask for src states: 0s for padded areas, 1s for the rest, shape (batch_size, 1, src_length)
- **unroll_steps** – number of steps to unroll the decoder RNN
- **hidden** – previous decoder hidden state, if not given it's initialized as in *self.init_hidden*, shape (batch_size, num_layers, hidden_size)
- **prev_att_vector** – previous attentional vector, if not given it's initialized with zeros, shape (batch_size, 1, hidden_size)

Returns

- outputs: shape (batch_size, unroll_steps, vocab_size),
- hidden: last hidden state (batch_size, num_layers, hidden_size),
- **att_probs: attention probabilities**
  with shape (batch_size, unroll_steps, src_length),
- **att_vectors: attentional vectors**
  with shape (batch_size, unroll_steps, hidden_size)

class joeynmt.decoders.**TransformerDecoder**(*num_layers: int = 4*, *num_heads: int = 8*, *hidden_size: int = 512*, *ff_size: int = 2048*, *dropout: float = 0.1*, *emb_dropout: float = 0.1*, *vocab_size: int = 1*, *freeze: bool = False*, ***kwargs*)

Bases: *Decoder*

A transformer decoder with N masked layers. Decoder layers are masked so that an attention head cannot see the future.

forward(*trg_embed: Tensor*, *encoder_output: Tensor*, *encoder_hidden: Tensor*, *src_mask: Tensor*, *unroll_steps: int*, *hidden: Tensor*, *trg_mask: Tensor*, ***kwargs*)

Transformer decoder forward pass.

Parameters

- **trg_embed** – embedded targets
- **encoder_output** – source representations
- **encoder_hidden** – unused
- **src_mask** –
- **unroll_steps** – unused
- **hidden** – unused
- **trg_mask** – to mask out target paddings Note that a subsequent mask is applied here.
- **kwargs** –

Returns

- decoder_output: shape (batch_size, seq_len, vocab_size)
- decoder_hidden: shape (batch_size, seq_len, emb_size)
- att_probs: shape (batch_size, trg_length, src_length),
- None

joeynmt.embeddings module

Embedding module

class joeynmt.embeddings.**Embeddings**(*embedding_dim: int = 64*, *scale: bool = False*, *vocab_size: int = 0*, *padding_idx: int = 1*, *freeze: bool = False*, ***kwargs*)

Bases: Module

Simple embeddings class

forward(*x: Tensor*) → Tensor

Perform lookup for input *x* in the embedding table.

Parameters
x – index in the vocabulary

Returns
embedded representation for *x*

load_from_file(*embed_path: Path*, *vocab:* Vocabulary) → None

Load pretrained embedding weights from text file.

- First line is expected to contain vocabulary size and dimension. The dimension has to match the model's specified embedding size, the vocabulary size is used in logging only.
- Each line should contain word and embedding weights separated by spaces.
- The pretrained vocabulary items that are not part of the joeynmt's vocabulary will be ignored (not loaded from the file).
- The initialization (specified in config["model"]["embed_initializer"]) of joeynmt's vocabulary items that are not part of the pretrained vocabulary will be kept (not overwritten in this func).
- This function should be called after initialization!

Example:
2 5 the -0.0230 -0.0264 0.0287 0.0171 0.1403 at -0.0395 -0.1286 0.0275 0.0254 -0.0932

Parameters

- **embed_path** – embedding weights text file
- **vocab** – Vocabulary object

joeynmt.encoders module

Various encoders

class joeynmt.encoders.**Encoder**(**args*, ***kwargs*)

Bases: Module

Base encoder class

property **output_size**

Return the output size

Returns

class joeynmt.encoders.**RecurrentEncoder**(*rnn_type: str = 'gru'*, *hidden_size: int = 1*, *emb_size: int = 1*, *num_layers: int = 1*, *dropout: float = 0.0*, *emb_dropout: float = 0.0*, *bidirectional: bool = True*, *freeze: bool = False*, ***kwargs*)

Bases: *Encoder*

Encodes a sequence of word embeddings

forward(*src_embed: Tensor*, *src_length: Tensor*, *mask: Tensor*, ***kwargs*) → Tuple[Tensor, Tensor, Tensor]

Applies a bidirectional RNN to sequence of embeddings x. The input mini-batch x needs to be sorted by src length. x and mask should have the same dimensions [batch, time, dim].

Parameters

- **src_embed** – embedded src inputs, shape (batch_size, src_len, embed_size)
- **src_length** – length of src inputs (counting tokens before padding), shape (batch_size)
- **mask** – indicates padding areas (zeros where padding), shape (batch_size, src_len, embed_size)
- **kwargs** –

Returns

- **output: hidden states with**
  shape (batch_size, max_length, directions*hidden),
- **hidden_concat: last hidden state with**
  shape (batch_size, directions*hidden)

class joeynmt.encoders.**TransformerEncoder**(*hidden_size: int = 512*, *ff_size: int = 2048*, *num_layers: int = 8*, *num_heads: int = 4*, *dropout: float = 0.1*, *emb_dropout: float = 0.1*, *freeze: bool = False*, ***kwargs*)

Bases: *Encoder*

Transformer Encoder

forward(*src_embed: Tensor*, *src_length: Tensor*, *mask: Tensor = None*, ***kwargs*) → Tuple[Tensor, Tensor]

Pass the input (and mask) through each layer in turn. Applies a Transformer encoder to sequence of embeddings x. The input mini-batch x needs to be sorted by src length. x and mask should have the same dimensions [batch, time, dim].

Parameters

- **src_embed** – embedded src inputs, shape (batch_size, src_len, embed_size)
- **src_length** – length of src inputs (counting tokens before padding), shape (batch_size)
- **mask** – indicates padding areas (zeros where padding), shape (batch_size, 1, src_len)

- **kwargs** –

Returns

- output: hidden states with shape (batch_size, max_length, hidden)
- None

joeynmt.helpers module

Collection of helper functions

joeynmt.helpers.**adjust_mask_size**(*mask: Tensor*, *batch_size: int*, *hyp_len: int*) → Tensor

Adjust mask size along dim=1. used for forced decoding (trg prompting).

Parameters

- **mask** – trg prompt mask in shape (batch_size, hyp_len)
- **batch_size** –
- **hyp_len** –

joeynmt.helpers.**check_version**(*cfg_version: str = None*) → str

Check joeynmt version

Parameters

cfg_version – version number specified in config

Returns

package version number string

joeynmt.helpers.**clones**(*module: Module*, *n: int*) → ModuleList

Produce N identical layers. Transformer helper function.

Parameters

- **module** – the module to clone
- **n** – clone this many times

Returns

cloned modules

joeynmt.helpers.**delete_ckpt**(*to_delete: Path*) → None

Delete checkpoint

Parameters

to_delete – checkpoint file to be deleted

joeynmt.helpers.**expand_reverse_index**(*reverse_index: List[int]*, *n_best: int = 1*) → List[int]

Expand resort_reverse_index for n_best prediction

ex. 1) reverse_index = [1, 0, 2] and n_best = 2, then this will return [2, 3, 0, 1, 4, 5].

ex. 2) reverse_index = [1, 0, 2] and n_best = 3, then this will return [3, 4, 5, 0, 1, 2, 6, 7, 8]

Parameters

- **reverse_index** – reverse_index returned from batch.sort_by_src_length()
- **n_best** –

Returns

expanded sort_reverse_index

joeynmt.helpers.**flatten**(*array: List[List[Any]]*) → List[Any]

Flatten a nested 2D list. faster even with a very long array than [item for subarray in array for item in subarray] or newarray.extend().

Parameters
array – a nested list

Returns
flattened list

joeynmt.helpers.**freeze_params**(*module: Module*) → None

Freeze the parameters of this module, i.e. do not update them during training

Parameters
module – freeze parameters of this module

joeynmt.helpers.**get_latest_checkpoint**(*ckpt_dir: Path*) → Path | None

Returns the latest checkpoint (by creation time, not the steps number!) from the given directory. If there is no checkpoint in this directory, returns None

Parameters
ckpt_dir –

Returns
latest checkpoint file

joeynmt.helpers.**load_checkpoint**(*path: Path*, *map_location: device | Dict*) → Dict

Load model from saved checkpoint.

Parameters

- **path** – path to checkpoint
- **device** – cuda device name or cpu

Returns
checkpoint (dict)

joeynmt.helpers.**make_model_dir**(*model_dir: Path*, *overwrite: bool = False*) → None

Create a new directory for the model.

Parameters

- **model_dir** – path to model directory
- **overwrite** – whether to overwrite an existing directory

joeynmt.helpers.**read_list_from_file**(*input_path: Path*) → List[str]

Read list of str from file in *input_path*.

Parameters
input_path – input file path

Returns
list of strings

joeynmt.helpers.**remove_extra_spaces**(*s: str*) → str

Remove extra spaces - used in pre_process() / post_process() in tokenizer.py

Parameters
s – input string

Returns
string w/o extra white spaces

joeynmt.helpers.**resolve_ckpt_path**(*load_model: Path*, *model_dir: Path*) → Path

Get checkpoint path. if *load_model* is not specified, take the best or latest checkpoint from model dir.

Parameters

- **load_model** – Path(cfg['training']['load_model']) or Path(cfg['testing']['load_model'])
- **model_dir** – Path(cfg['model_dir'])

Returns
resolved checkpoint path

joeynmt.helpers.**save_hypothese**(*output_path: Path*, *hypotheses: List[str]*, *n_best: str = 1*) → None

Save list hypothese to file.

Parameters

- **output_path** – output file path
- **hypotheses** – hypothese to write
- **n_best** – n_best size

joeynmt.helpers.**set_seed**(*seed: int*) → None

Set the random seed for modules torch, numpy and random.

Parameters
seed – random seed

joeynmt.helpers.**store_attention_plots**(*attentions: ndarray*, *targets: List[List[str]]*, *sources: List[List[str]]*, *output_prefix: str*, *indices: List[int]*, *tb_writer: SummaryWriter | None = None*, *steps: int = 0*) → None

Saves attention plots.

Parameters

- **attentions** – attention scores
- **targets** – list of tokenized targets
- **sources** – list of tokenized sources
- **output_prefix** – prefix for attention plots
- **indices** – indices selected for plotting
- **tb_writer** – Tensorboard summary writer (optional)
- **steps** – current training steps, needed for tb_writer
- **dpi** – resolution for images

joeynmt.helpers.**subsequent_mask**(*size: int*) → Tensor

Mask out subsequent positions (to prevent attending to future positions) Transformer helper function.

Parameters
size – size of mask (2nd and 3rd dim)

Returns
Tensor with 0s and 1s of shape (1, size, size)

joeynmt.helpers.**symlink_update**(*target: Path*, *link_name: Path*) → Path | None

This function finds the file that the symlink currently points to, sets it to the new target, and returns the previous target if it exists.

Parameters

- **target** – A path to a file that we want the symlink to point to. no parent dir, filename only, i.e. "10000.ckpt"
- **link_name** – This is the name of the symlink that we want to update. link name with parent dir, i.e. "models/my_model/best.ckpt"

Returns

- **current_last: This is the previous target of the symlink, before it is** updated in this function. If the symlink did not exist before or did not have a target, None is returned instead.

joeynmt.helpers.**tile**(*x: Tensor*, *count: int*, *dim=0*) → Tensor

Tiles x on dimension dim count times. From OpenNMT. Used for beam search.

Parameters

- **x** – tensor to tile
- **count** – number of tiles
- **dim** – dimension along which the tensor is tiled

Returns
tiled tensor

joeynmt.helpers.**unicode_normalize**(*s: str*) → str

apply unicodedata NFKC normalization - used in pre_process() in tokenizer.py

Parameters
s – input string

Returns
normalized string

joeynmt.helpers.**write_list_to_file**(*output_path: Path*, *array: List[Any]*) → None

Write list of str to file in *output_path*.

Parameters

- **output_path** – output file path
- **array** – list of strings

joeynmt.initialization module

Implements custom initialization

joeynmt.initialization.**compute_alpha_beta**(*num_enc_layers: int*, *num_dec_layers: int*) → Dict[str, Dict]

DeepNet: compute alpha/beta value suggested in https://arxiv.org/abs/2203.00555

joeynmt.initialization.**initialize_model**(*model: Module*, *cfg: dict*, *src_padding_idx: int*, *trg_padding_idx: int*) → None

This initializes a model based on the provided config.

All initializer configuration is part of the *model* section of the configuration file. For an example, see e.g. *https://github.com/joeynmt/joeynmt/blob/main/ configs/iwslt14_ende_spm.yaml*.

The main initializer is set using the *initializer* key. Possible values are *xavier*, *uniform*, *normal* or *zeros*. (*xavier* is the default).

When an initializer is set to *uniform*, then *init_weight* sets the range for the values (-init_weight, init_weight).

When an initializer is set to *normal*, then *init_weight* sets the standard deviation for the weights (with mean 0).

The word embedding initializer is set using *embed_initializer* and takes the same values. The default is *normal* with *embed_init_weight = 0.01*.

Biases are initialized separately using *bias_initializer*. The default is *zeros*, but you can use the same initializers as the main initializer.

Set *init_rnn_orthogonal* to True if you want RNN orthogonal initialization (for recurrent matrices). Default is False.

lstm_forget_gate controls how the LSTM forget gate is initialized. Default is *1*.

Parameters

- **model** – model to initialize
- **cfg** – the model configuration
- **src_padding_idx** – index of source padding token
- **trg_padding_idx** – index of target padding token

joeynmt.initialization.**lstm_forget_gate_init_**(*cell: RNNBase*, *value: float = 1.0*) → None

Initialize LSTM forget gates with *value*.

Parameters

- **cell** – LSTM cell
- **value** – initial value, default: 1

joeynmt.initialization.**orthogonal_rnn_init_**(*cell: RNNBase*, *gain: float = 1.0*) → None

Orthogonal initialization of recurrent weights RNN parameters contain 3 or 4 matrices in one parameter, so we slice it.

joeynmt.initialization.**xavier_uniform_n_**(*w: Tensor*, *gain: float = 1.0*, *n: int = 4*) → None

Xavier initializer for parameters that combine multiple matrices in one parameter for efficiency. This is e.g. used for GRU and LSTM parameters, where e.g. all gates are computed at the same time by 1 big matrix.

Parameters

- **w** – parameter
- **gain** – default 1
- **n** – default 4

joeynmt.metrics module

Evaluation metrics

joeynmt.metrics.**bleu**(*hypotheses: List[str]*, *references: List[str]*, ***sacrebleu_cfg*) → float

Raw corpus BLEU from sacrebleu (without tokenization) cf. https://github.com/mjpost/sacrebleu/blob/master/sacrebleu/metrics/bleu.py

Parameters

- **hypotheses** – list of hypotheses (strings)
- **references** – list of references (strings)

Returns
bleu score

joeynmt.metrics.**chrf**(*hypotheses: List[str]*, *references: List[str]*, ***sacrebleu_cfg*) → float

Character F-score from sacrebleu cf. https://github.com/mjpost/sacrebleu/blob/master/sacrebleu/metrics/chrf.py

Parameters

- **hypotheses** – list of hypotheses (strings)
- **references** – list of references (strings)

Returns
character f-score (0 <= chf <= 1) see Breaking Change in sacrebleu v2.0

joeynmt.metrics.**sequence_accuracy**(*hypotheses: List[str]*, *references: List[str]*) → float

Compute the accuracy of hypothesis tokens: correct tokens / all tokens Tokens are correct if they appear in the same position in the reference. We lookup the references before one-hot-encoding, that is, hypotheses with UNK are always evaluated as incorrect.

Parameters

- **hypotheses** – list of hypotheses (strings)
- **references** – list of references (strings)

Returns

joeynmt.metrics.**token_accuracy**(*hypotheses: List[str]*, *references: List[str]*, *tokenizer: Callable*) → float

Compute the accuracy of hypothesis tokens: correct tokens / all tokens Tokens are correct if they appear in the same position in the reference. We lookup the references before one-hot-encoding, that is, UNK generation in hypotheses is always evaluated as incorrect.

Parameters

- **hypotheses** – list of hypotheses (strings)
- **references** – list of references (strings)

Returns
token accuracy (float)

joeynmt.model module

Module to represents whole models

class joeynmt.model.**DataParallelWrapper**(*module: Module*)

Bases: Module

DataParallel wrapper to pass through the model attributes

ex. 1) for DataParallel

```
>>> from torch.nn import DataParallel as DP
>>> model = DataParallelWrapper(DP(model))
```

ex. 2) for DistributedDataParallel

```
>>> from torch.nn.parallel import DistributedDataParallel as DDP
>>> model = DataParallelWrapper(DDP(model))
```

forward(**args*, ***kwargs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

load_state_dict(**args*, ***kwargs*)

loading the twice-wrapped module.

state_dict(**args*, ***kwargs*)

saving the twice-wrapped module.

class joeynmt.model.**Model**(*encoder: Encoder*, *decoder: Decoder*, *src_embed: Embeddings*, *trg_embed: Embeddings*, *src_vocab: Vocabulary*, *trg_vocab: Vocabulary*)

Bases: Module

Base Model class

forward(*return_type: str = None*, ***kwargs*) → Tuple[Tensor, Tensor, Tensor, Tensor]

Interface for multi-gpu

For DataParallel, We need to encapsulate all model call: *model.encode()*, *model.decode()*, and *model.encode_decode()* by *model.__call__()*. *model.__call__()* triggers model.forward() together with pre hooks and post hooks, which takes care of multi-gpu distribution.

Parameters

return_type – one of {"loss", "encode", "decode"}

log_parameters_list() → None

Write all model parameters (name, shape) to the log.

property loss_function

joeynmt.model.**build_model**(*cfg: Dict = None*, *src_vocab: Vocabulary = None*, *trg_vocab: Vocabulary = None*) → *Model*

Build and initialize the model according to the configuration.

Parameters

- **cfg** – dictionary configuration containing model specifications
- **src_vocab** – source vocabulary
- **trg_vocab** – target vocabulary

Returns
built and initialized model

joeynmt.plotting module

Plot attentions

joeynmt.plotting.**plot_heatmap**(*scores: ndarray*, *column_labels: List[str]*, *row_labels: List[str]*, *output_path: str | None = None*, *dpi: int = 300*) → Figure

Plotting function that can be used to visualize (self-)attention. Plots are saved if *output_path* is specified, in format that this file ends with ('pdf' or 'png').

Parameters

- **scores** – attention scores
- **column_labels** – labels for columns (e.g. target tokens)
- **row_labels** – labels for rows (e.g. source tokens)
- **output_path** – path to save to
- **dpi** – set resolution for matplotlib

Returns
pyplot figure

joeynmt.prediction module

This module holds methods for generating predictions from a model.

joeynmt.prediction.**evaluate**(*valid_scores: Dict*, *valid_hyp: List*, *data: Dataset*, *args: TestConfig*) → Tuple[Dict[str, float], List[str]]

Compute evaluateion metrics

Parameters

- **valid_scores** – scores dict
- **valid_hyp** – decoded hypotheses
- **data** – eval Dataset
- **args** – configuration args

Returns

- valid_scores: evaluation scores
- valid_ref: postprocessed references

joeynmt.prediction.**predict**(*model: Model*, *data: Dataset*, *device: device*, *n_gpu: int*, *rank: int = 0*, *compute_loss: bool = False*, *normalization: str = 'batch'*, *num_workers: int = 0*, *args: TestConfig = None*, *autocast: Dict = None*) → Tuple[Dict[str, float], List[str] | None, List[str] | None, List[List[str]], List[ndarray], List[ndarray]]

Generates translations for the given data. If *compute_loss* is True and references are given, also computes the loss.

Parameters

- **model** – model module
- **data** – dataset for validation
- **device** – torch device
- **n_gpu** – number of GPUs
- **rank** – ddp rank
- **compute_loss** – whether to compute a scalar loss for given inputs and targets
- **normalization** – one of {*batch*, *tokens*, *none*}
- **num_workers** – number of workers for *collate_fn()* in data iterator
- **args** – configuration args
- **autocast** – autocast context

Returns

- valid_scores: (dict) current validation scores,
- valid_ref: (list of str) post-processed validation references,
- valid_hyp: (list of str) post-processed validation hypotheses,
- decoded_valid: (list of list of str) token-level validation hypotheses,
- valid_seq_scores: (list of np.array) log probabilities (hyp or ref)
- valid_attn_scores: (list of np.array) attention scores (hyp or ref)

joeynmt.prediction.**prepare**(*args: BaseConfig*, *rank: int*, *mode: str*) → Tuple[*Model*, Dataset, Dataset, Dataset]

Helper function for model and data loading.

Parameters

- **args** – config args
- **rank** – ddp rank
- **mode** – execution mode

joeynmt.prediction.**test**(*cfg: Dict*, *output_path: str = None*, *prepared: Dict = None*, *save_attention: bool = False*, *save_scores: bool = False*) → None

Main test function. Handles loading a model from checkpoint, generating translations, storing them, and plotting attention.

Parameters

- **cfg** – configuration dict
- **output_path** – path to output
- **prepared** – model and datasets passed from training

- **save_attention** – whether to save attention visualizations
- **save_scores** – whether to save scores

joeynmt.prediction.**translate**(*cfg: Dict*, *output_path: str = None*) → None

Interactive translation function. Loads model from checkpoint and translates either the stdin input or asks for input to translate interactively. Translations and scores are printed to stdout. Note: The input sentences don't have to be pre-tokenized.

Parameters

- **cfg** – configuration dict
- **output_path** – path to output file

joeynmt.search module

Search module

joeynmt.search.**beam_search**(*model: Model*, *beam_size: int*, *encoder_output: Tensor*, *encoder_hidden: Tensor*, *src_mask: Tensor*, *max_output_length: int*, *alpha: float*, *n_best: int = 1*, ***kwargs*) → Tuple[Tensor, Tensor, Tensor]

Beam search with size k. In each decoding step, find the k most likely partial hypotheses. Inspired by OpenNMT-py, adapted for Transformer.

Parameters

- **model** –
- **beam_size** – size of the beam
- **encoder_output** –
- **encoder_hidden** –
- **src_mask** –
- **max_output_length** –
- **alpha** – *alpha* factor for length penalty
- **n_best** – return this many hypotheses, <= beam (currently only 1)

Returns

- stacked_output: output hypotheses (2d array of indices),
- stacked_scores: scores (2d array of sequence-wise log probabilities),
- stacked_attention_scores: attention scores (3d array)

joeynmt.search.**greedy**(*src_mask: Tensor*, *max_output_length: int*, *model: Model*, *encoder_output: Tensor*, *encoder_hidden: Tensor*, ***kwargs*) → Tuple[Tensor, Tensor, Tensor]

Greedy decoding. Select the token word highest probability at each time step. This function is a wrapper that calls recurrent_greedy for recurrent decoders and transformer_greedy for transformer decoders.

Parameters

- **src_mask** – mask for source inputs, 0 for positions after </s>
- **max_output_length** – maximum length for the hypotheses
- **model** – model to use for greedy decoding
- **encoder_output** – encoder hidden states for attention

- **encoder_hidden** – encoder last state for decoder initialization

Returns

- stacked_output: output hypotheses (2d array of indices),
- stacked_scores: scores (2d array of token-wise log probabilities),
- stacked_attention_scores: attention scores (3d array)

joeynmt.search.**search**(*model: Model*, *batch: Batch*, *max_output_length: int*, *beam_size: int*, *beam_alpha: float*, *n_best: int = 1*, ***kwargs*) → Tuple[ndarray, ndarray, ndarray]

Get outputs and attentions scores for a given batch.

Parameters

- **model** – Model class
- **batch** – batch to generate hypotheses for
- **max_output_length** – maximum length of hypotheses
- **beam_size** – size of the beam for beam search, if 0 use greedy
- **beam_alpha** – alpha value for beam search
- **n_best** – candidates to return

Returns

- stacked_output: hypotheses for batch,
- stacked_scores: log probabilities for batch,
- stacked_attention_scores: attention scores for batch

joeynmt.tokenizers module

Tokenizer module

class joeynmt.tokenizers.**BasicTokenizer**(*level: str = 'word'*, *lowercase: bool = False*, *normalize: bool = False*, *max_length: int = -1*, *min_length: int = -1*, ***kwargs*)

Bases: object

SPACE = ' '

SPACE_ESCAPE = ''

post_process(*sequence: List[str] | str*, *generate_unk: bool = True*, *cut_at_sep: bool = True*) → str

Detokenize

pre_process(*raw_input: str*, *allow_empty: bool = False*) → str

Pre-process text

- **ex.) Lowercase, Normalize, Remove emojis,**
  Pre-tokenize(add extra white space before punc) etc.
- applied for all inputs both in training and inference.

Parameters

- **raw_input** – raw input string
- **allow_empty** – whether to allow empty string

Returns
preprocessed input string

set_vocab(*vocab*) → None

Set vocab :param vocab: (Vocabulary)

class joeynmt.tokenizers.**SentencePieceTokenizer**(*level: str = 'bpe'*, *lowercase: bool = False*, *normalize: bool = False*, *max_length: int = -1*, *min_length: int = -1*, ***kwargs*)

Bases: *BasicTokenizer*

copy_cfg_file(*model_dir: Path*) → None

Copy config file to model_dir

post_process(*sequence: List[str] | str*, *generate_unk: bool = True*, *cut_at_sep: bool = True*) → str

Detokenize

set_vocab(*vocab*) → None

Set vocab

class joeynmt.tokenizers.**SubwordNMTTokenizer**(*level: str = 'bpe'*, *lowercase: bool = False*, *normalize: bool = False*, *max_length: int = -1*, *min_length: int = -1*, ***kwargs*)

Bases: *BasicTokenizer*

copy_cfg_file(*model_dir: Path*) → None

Copy config file to model_dir

post_process(*sequence: List[str] | str*, *generate_unk: bool = True*, *cut_at_sep: bool = True*) → str

Detokenize

set_vocab(*vocab*) → None

Set vocab

joeynmt.tokenizers.**build_tokenizer**(*cfg: Dict*) → Dict[str, *BasicTokenizer*]

joeynmt.training module

Training module

class joeynmt.training.**TrainManager**(*rank: int*, *model:* Model, *model_dir: Path*, *device: device*, *n_gpu: int = 0*, *num_workers: int = 0*, *autocast: Dict = None*, *seed: int = 42*, *train_args:* TrainConfig *= None*, *dev_args:* TestConfig *= None*)

Bases: `object`

Manages training loop, validations, learning rate scheduling and early stopping.

class TrainStatistics(*minimize_metric: bool = True*)

Bases: `object`

Train Statistics

Parameters

- **epochs** – epoch counter
- **steps** – global update step counter
- **is_min_lr** – stop by reaching learning rate minimum

- **is_max_update** – stop by reaching max num of updates
- **total_tokens** – number of total tokens seen so far
- **best_ckpt_iter** – store iteration point of best ckpt
- **minimize_metric** – minimize or maximize score
- **total_correct** – number of correct tokens seen so far

is_best(*score*) → bool

is_better(*score: float*, *heap_queue: list*) → bool

load_state_dict(*state_dict: Dict*) → None

Given a state_dict, this function reconstruct the state

state_dict() → Dict

Returns a dictionary of values necessary to reconstruct stats

init_from_checkpoint(*path: Path*, *reset_best_ckpt: bool = False*, *reset_scheduler: bool = False*, *reset_optimizer: bool = False*, *reset_iter_state: bool = False*) → None

Initialize the trainer from a given checkpoint file.

This checkpoint file contains not only model parameters, but also scheduler and optimizer states, see *self._save_checkpoint*.

Parameters

- **path** – path to checkpoint
- **reset_best_ckpt** – reset tracking of the best checkpoint, use for domain adaptation with a new dev set.
- **reset_scheduler** – reset the learning rate scheduler, and do not use the one stored in the checkpoint.
- **reset_optimizer** – reset the optimizer, and do not use the one stored in the checkpoint.
- **reset_iter_state** – reset the sampler's internal state and do not use the one stored in the checkpoint.

init_layers(*path: Path*, *layer: str*) → None

Initialize encoder decoder layers from a given checkpoint file.

Parameters

- **path** – path to checkpoint
- **layer** – layer name; 'encoder' or 'decoder' expected

train_and_validate(*train_data: Dataset*, *valid_data: Dataset*) → None

Train the model and validate it from time to time on the validation set.

Parameters

- **train_data** – training data
- **valid_data** – validation data

joeynmt.training.**train**(*rank: int*, *world_size: int*, *cfg: Dict*, *skip_test: bool = False*) → None

Main training function. After training, also test on test data if given.

Parameters

- **rank** – ddp local rank
- **world_size** – ddp world size
- **cfg** – configuration dict
- **skip_test** – whether a test should be run or not after training

joeynmt.vocabulary module

Vocabulary module

class joeynmt.vocabulary.**Vocabulary**(*tokens: List[str]*, *cfg: SimpleNamespace*)

Bases: object

Vocabulary represents mapping between tokens and indices.

add_tokens(*tokens: List[str]*) → None

Add list of tokens to vocabulary

Parameters
tokens – list of tokens to add to the vocabulary

arrays_to_sentences(*arrays: ndarray*, *cut_at_eos: bool = True*, *skip_pad: bool = True*) → List[List[str]]

Convert multiple arrays containing sequences of token IDs to their sentences, optionally cutting them off at the end-of-sequence token.

Parameters

- **arrays** – 2D array containing indices
- **cut_at_eos** – cut the decoded sentences at the first <eos>
- **skip_pad** – skip generated <pad> tokens

Returns
list of list of strings (tokens)

is_unk(*token: str*) → bool

Check whether a token is covered by the vocabulary

Parameters
token –

Returns
True if covered, False otherwise

log_vocab(*k: int*) → str

first k vocab entities

lookup(*token: str*) → int

look up the encoding dictionary. (needed for multiprocessing)

Parameters
token – surface str

Returns
token id

sentences_to_ids(*sentences: List[List[str]]*, *bos: bool = True*, *eos: bool = True*) → Tuple[List[List[int]], List[int], List[int]]

Encode sentences to indices and pad sequences to the maximum length of the sentences given

Parameters

- **sentences** – list of tokenized sentences
- **bos** – whether to add <bos>
- **eos** – whether to add <eos>

Returns

- padded ids
- original lengths before padding
- prompt_mask

to_file(*file: Path*) → None

Save the vocabulary to a file, by writing token with index i in line i.

Parameters
file – path to file where the vocabulary is written

joeynmt.vocabulary.**build_vocab**(*cfg: Dict*, *dataset:* BaseDataset *= None*, *model_dir: Path = None*) → Tuple[*Vocabulary*, *Vocabulary*]

joeynmt.vocabulary.**sort_and_cut**(*counter: Counter*, *max_size: int = 9223372036854775807*, *min_freq: int = -1*) → List[str]

Cut counter to most frequent, sorted numerically and alphabetically :param counter: flattened token list in Counter object :param max_size: maximum size of vocabulary :param min_freq: minimum frequency for an item to be included :return: list of valid tokens

joeynmt.loss module

Module to implement training loss

class joeynmt.loss.**XentLoss**(*pad_index: int*, *smoothing: float = 0.0*)

Bases: Module

Cross-Entropy Loss with optional label smoothing

forward(*log_probs: Tensor*, ***kwargs*) → Tensor

Compute the cross-entropy between logits and targets.

If label smoothing is used, target distributions are not one-hot, but "1-smoothing" for the correct target token and the rest of the probability mass is uniformly spread across the other tokens.

Parameters
log_probs – log probabilities as predicted by model

Returns
logits

joeynmt.transformer_layers module

Transformer layers

class joeynmt.transformer_layers.**MultiHeadedAttention**(*num_heads: int*, *size: int*, *dropout: float = 0.1*)

Bases: Module

Multi-Head Attention module from "Attention is All You Need"

Implementation modified from OpenNMT-py. https://github.com/OpenNMT/OpenNMT-py

forward(*k: Tensor*, *v: Tensor*, *q: Tensor*, *mask: Tensor | None = None*, *return_weights: bool | None = None*)

Computes multi-headed attention.

Parameters

- **k** – keys [batch_size, seq_len, hidden_size]
- **v** – values [batch_size, seq_len, hidden_size]
- **q** – query [batch_size, seq_len, hidden_size]
- **mask** – optional mask [batch_size, 1, seq_len]
- **return_weights** – whether to return the attention weights, averaged over heads.

Returns

- output [batch_size, query_len, hidden_size]
- attention_weights [batch_size, query_len, key_len]

class joeynmt.transformer_layers.**PositionalEncoding**(*size: int = 0*, *max_len: int = 5000*)

Bases: Module

Pre-compute position encodings (PE). In forward pass, this adds the position-encodings to the input for as many time steps as necessary.

Implementation based on OpenNMT-py. https://github.com/OpenNMT/OpenNMT-py

forward(*emb: Tensor*) → Tensor

Embed inputs.

Parameters
emb – (Tensor) Sequence of word embeddings vectors shape (seq_len, batch_size, dim)

Returns
positionally encoded word embeddings

class joeynmt.transformer_layers.**PositionwiseFeedForward**(*input_size: int*, *ff_size: int*, *dropout: float = 0.1*, *alpha: float = 1.0*, *layer_norm: str = 'post'*, *activation: str = 'relu'*)

Bases: Module

Position-wise Feed-forward layer Projects to ff_size and then back down to input_size.

forward(*x: Tensor*) → Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class joeynmt.transformer_layers.**TransformerDecoderLayer**(*size: int = 0*, *ff_size: int = 0*, *num_heads: int = 0*, *dropout: float = 0.1*, *alpha: float = 1.0*, *layer_norm: str = 'post'*, *activation: str = 'relu'*)

Bases: `Module`

Transformer decoder layer.

Consists of self-attention, source-attention, and feed-forward.

forward(*x: Tensor*, *memory: Tensor*, *src_mask: Tensor*, *trg_mask: Tensor*, *return_attention: bool = False*, ***kwargs*) → Tensor

Forward pass of a single Transformer decoder layer.

First applies target-target self-attention, dropout with residual connection (adding the input to the result), and layer norm.

Second computes source-target cross-attention, dropout with residual connection (adding the self-attention to the result), and layer norm.

Finally goes through a position-wise feed-forward layer.

Parameters

- **x** – inputs
- **memory** – source representations
- **src_mask** – source mask
- **trg_mask** – target mask (so as not to condition on future steps)
- **return_attention** – whether to return the attention weights

Returns

- output tensor
- attention weights

class joeynmt.transformer_layers.**TransformerEncoderLayer**(*size: int = 0*, *ff_size: int = 0*, *num_heads: int = 0*, *dropout: float = 0.1*, *alpha: float = 1.0*, *layer_norm: str = 'post'*, *activation: str = 'relu'*)

Bases: `Module`

One Transformer encoder layer has a Multi-head attention layer plus a position-wise feed-forward layer.

forward(*x: Tensor*, *mask: Tensor*) → Tensor

Forward pass for a single transformer encoder layer. First applies self attention, then dropout with residual connection (adding the input to the result), then layer norm, and then a position-wise feed-forward layer.

Parameters

- **x** – layer input
- **mask** – input mask

Returns
output tensor

2.7 Frequently Asked Questions

2.7.1 Documentation

- **Are there any Notebooks for Joey?**
  - Quick start tutorial: A quick start guide with Tatoeba corpus example.
  - Torchhub tutorial: How to generate translations from a pretrained model.
- **The documentation is too old, and doesn't reflect the latest functionality implemented in the main branch of the repository.**
  We try to keep the documentation up-to-date and aligned with the latest stable release.
- **I can't find the information I'm looking for. What now?**
  Open an issue on GitHub or write an email.

2.7.2 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/prediction on GPU (requires CUDA) or to False for training/prediction on CPU.
- **Does Joey NMT support multi-GPU processing?**
  We integrated multi-gpu support (DataParallel) in version 1.0, and multi-node DDP (Distributed DataParallel) in version 2.3. Note that the interactive translation mode currently works on single GPU or CPU only.
- **How can I stop training?**
  Simply press Control+C. In DDP, this keyboard interruption might not be able to stop all the processes. Please clean up the remaining processes manually.
- **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. This evokes random subsampling at the beginning of each epoch. That is, the model will see different random subset of training data at each epoch.
- **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.

```
python 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: `voc_file: "out-of-domain-model/src_vocab.txt"` and likewise for `trg_vocab.txt`. You have to specify this, otherwise Joey NMT 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.

```
name: "finetuning_experiment" # in-domain model
model_dir "in-domain-model"   # new model dir
[...]

data:
    train: "path/to/in-domain-data/train"
    dev: "path/to/in-domain-data/dev"
    test: "path/to/in-domain-data/test"
    [...]
    src:
        [...]
        voc_file: "out-of-domain-model/src_vocab.txt" # reuse the same vocab
    trg:
        [...]
        voc_file: "out-of-domain-model/trg_vocab.txt" # reuse the same vocab
[...]

training:
    load_model: "out-of-domain-model/best.ckpt"  # warm start with pretrained␣
↪weights
    reset_best_ckpt: True  # reset previous high scores under your metric
    reset_scheduler: False # set True i.e. if you use a different scheduler in␣
↪fine tuning
    reset_optimizer: False # set True i.e. if you use a different optimizer in␣
↪fine tuning
    reset_iter_state: True # reset training iteration stats (epoch no. and␣
↪update counts)
    [...]
```

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 (`voc_file`) 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.

2.7.3 Generation

- **Why do I see the same output sentences in the n-best list?**
  In BPE decoding, there are multiple ways to tokenize one sequence. That is, the same output string sequence might appear multiple times in the n-best list, because they have different tokenization and thus different sequence in the generation. For instance, say 3-best generation were:

```
1 best ['', 'N', 'e', 'w', 'York']
2 best ['', 'New', 'York']
3 best ['', 'New', 'Y', 'o', 'r', 'k']
```

  All three were different in next-token prediction, but ended up the same string sequence *New York* after being un-bpe-ed.

- **My translation contains a lot of repetitions. Can I prevent them?**
  - First of all, repetitions can be often observed in a premature stage of training. Please check if your training has converged.
  - Joey NMT offers **Ngram Blocker** and **Repetition Penalty** mechanism to avoid repetitions in generation. **Ngram Blocker** checks. Let's cosider a partial translation "they play in the park in the". Under the option `no_repeat_ngram_size=3`, a trigram `["in", "the", "park"]` will be repeated, if the model generate *NextToken* = *"park"* in the next generation step. So, the probability of the token "park" will be artificially set to zero in order to avid ngram repetition.

```
["they", "play", "in", "the", "park", "in", "the", NextToken]
                                                    ^
                                      set probability of "park" to zero
```

  Note that this `no_repeat_ngram_size` may the process drastically slow down, since it needs to move the tokens on cpu, and put them back to gpu after the ngram computation.

  Repetition Penalty decreases probability of all the tokens already decoded so far; `"they"`, `"play"`, `"in"`, `"the"`, and `"park"`. So the token `"park"` will be ranked lower than it should be, and therefore can be avoided.

```
["they", "play", "in", "the", "park", "in", "the", NextToken]
                                                    ^
                        decrease the probability of already decoded tokens
```

  `repetition_penaly` option takes value between 0.0 and 1.0 to penalize the repeated tokens. This operation is done on GPU, without offloading the tokens to CPU.

2.7.4 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 were, 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.

2.7.5 Tensorboard

- **How can I start Tensorboard for a model that I trained on a remote server?**
  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.

2.7.6 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?**
  Joey NMT 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/transformer_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 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 usually easier to tune. Popel and Bojar (2018) give further tips on how to tune the hyper-parameters for the Transformer.

- **When should I specify voc_file in the config, when not?**
  - *Training*: When you pre-generated the vocabulary (i.e. using build_vocab.py, or get_iwslt14_bpe.sh), you should specify the vocab file path in the config before you start training. Otherwise you can omit the `voc_file` field. In that case, Joey NMT builds vocabularies per-language separately and export them in the `model_dir` during training. When you resume an interrupted training process, or you train a domain-adaptation model, you should put the same vocab files path so that Joey NMT won't create a new vocabulary set.
  - *Testing*: You always should specify the vocabulary files path in the config. You can find the vocabulary files located in the `model_dir` after the training has finished.

2.7.7 Data

- **Does Joey NMT pre-process my data?**
  Yes. When the data are loaded, Joey NMT applies several pre-processing defined in the Tokenizer module, such as lowercasing, unicode normalization etc. You can control it in the data section of the configuration. See `pre_process()` function in the `BasicTokenizer` class.

  Tokenization is triggered on-the-fly during batch construction. Currently, Joey NMT implements wrappers for subword-nmt and sentencepiece library for BPEs, in addition to the simple white-space split (word-level tokenization) and character-level tokenization.

- **Does Joey NMT post-process your data?**
  The Tokenizer module takes care of post-processing like detokenization, recasing etc. If you want to add custom post-process operations, you can extend the `post_process()` function there.

2.7.8 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; it has lots of advice on how to find out what's going wrong and how to fix it. Specifically for NMT, here're 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 terribly 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_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 `batch_size` in training section.

- **I've encountered a file IO error. What should I do?**
  Pay attention to the relative path structure. Most scripts are designed to be called from the project root path. Consider to use absolute path in the configuration file.

2.7.9 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.py 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 Joey NMT 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 Joey NMT, 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 metrics (`eval_metrics`); 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_length" in the data section and "max_output_length" in the testing section?**
  `max_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". In prediction, the vocabulary should **NOT** be re-generated, but reused the same vocabulary created in training. Make sure that you put the correct vocab file paths in config, before you trigger the "test" or "translation" mode.
- **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, make 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.
- **What are the language tags?**
  Language tags are special tokens that control translation directions in multilingual training. These tokens need special handling in tokenization. For example, `"<en> Hello"` should **NOT** be tokenized as `["<", "en", ">", "Hello"]`, but as `["<en>", "Hello"]`. You may need to manually modify :scripts`:build_vocab.py` script for multilingual joint vocab construction.

  Currently, multilingal models with language tags don't work in the interactive translation mode. Use test mode or TorchHub API (See Torchhub tutorial)

2.7.10 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.

2.7.11 Miscellaneous

- **Why should I use Joey NMT 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 the 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".

2.7.12 Contributing

- **How can I contribute?**
  Open an issue on GitHub and make a pull request. To ensure the repository stays clean, unittests and linters are triggered by github's workflow on every push or pull request to `main` branch. Before you create a pull request, you can check the validity of your modifications with the following commands:

  ```
  make test
  make check
  make -C docs clean html
  ```

- **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.

2.7.13 Evaluation

- **Which quality metrics does Joey NMT report?**
  Joey NMT reports BLEU, chrF, sentence- and token-level accuracy. You can choose which of those to report with setting `eval_metrics` 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 Joey NMT using to compute BLEU scores?**
  Joey NMT 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 Joey NMT 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 reproducibility 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.

2.7.14 Distributed Data Parallel

- **How can I evoke DDP training?**
  Add `--use-ddp` flag.

  ```
  python -m joeynmt train configs/ddp_model.yaml --use-ddp --skip-test
  ```

  Currently, we implemented DDP-training only, we don't support DDP-prediction. Don't forget to add `--skip-test` option above!

- **Can I use batch_type: "token"?**
  No. We only support `batch_type: "sentence"`, in DDP. See `DistributedSubsetSampler` class in helpers_for_ddp.py.

- **How can I set MASTER_ADDR and MASTER_PORT env variables?**
  These values are currently hard-coded. See `ddp_setup()` function in helpers_for_ddp.py.

- **It seems the early stopping criterion is not working properly in DDP.**
  Currently, early stopping (`break` in multi-process for-loops) is not always syncronized across devices, presumably. In addition, Keyborad interruption (crtl-c) doesn't stop all the processes. You may need to take care of the remaining processes manually.

2.8 Resources

2.8.1 Neural Machine Translation

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

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)
- Huggingface NLP task guides: Translation

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.
- Huggingface: datasets.

Toolkits

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

2.8.2 PyTorch

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

- Introduction to PyTorch - YouTube Series
- PyTorch Tutorials
- PyTorch Distributed Overview

2.8.3 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.

2.9 Change log

2.9.1 v2.3 - Jan 25, 2024

- introduced DistributedDataParallel
- implemented language tags, see torchhub.ipynb
- released a iwslt14 de-en-fr multilingual model (trained using DDP)
- special symbols definition refactoring
- configuration refactoring
- autocast refactoring
- enabled activation function selection
- bugfixes
- upgrade to python 3.11, torch 2.1.2
- documentation refactoring

2.9.2 v2.2 - Jan 15, 2023

- compatibility with torch 2.0 tested
- torchhub introduced
- bugfixes, minor refactoring

2.9.3 v2.1 - Sep 18, 2022

- upgrade to python 3.10, torch 1.12
- replace Automated Mixed Precision from NVIDA's amp to Pytorch's amp package
- replace discord.py with pycord in the Discord Bot demo
- data iterator refactoring
- add wmt14 ende / deen benchmark trained on v2 from scratch
- add tokenizer tutorial
- minor bugfixes

2.9.4 v2.0 - Jun 2, 2022

Breaking change!

- upgrade to python 3.9, torch 1.11
- `torchtext.legacy` dependencies are completely replaced by `torch.utils.data`
- tokenizers.py: handles tokenization internally (also supports bpe-dropout!)
- datasets.py: loads data from plaintext, tsv, and huggingface's datasets
- build_vocab.py: trains subwords, creates joint vocab
- enhancement in decoding - scoring with hypotheses or references - repetition penalty, ngram blocker - attention plots for transformers
- yapf, isort, flake8 introduced
- bugfixes, minor refactoring

Warning: The models trained with Joey NMT v1.x can be decoded with Joey NMT v2.0. But there is no guarantee that you can reproduce the same score as before.

2.9.5 v1.5 - Jan 18, 2022

- requirements update (Six >= 1.12)

2.9.6 v1.4 - Jan 18, 2022

- upgrade to sacrebleu 2.0, python 3.7, torch 1.8
- bugfixes

2.9.7 v1.3 - Apr 14, 2021

- upgrade to torchtext 0.9 (torchtext -> torchtext.legacy)
- n-best decoding
- demo colab notebook

2.9.8 v1.0 - Oct 31, 2020

- Multi-GPU support
- fp16 (half precision) support

2.9.9 v0.9 - Jul 28, 2019

- pre-release

PYTHON MODULE INDEX

j

INDEX

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X