# Joey NMT: A Minimalist NMT Toolkit for Novices

**Julia Kreutzer**
Computational Linguistics
Heidelberg University
kreutzer@cl.uni-heidelberg.de

**Joost Bastings**
ILLC
University of Amsterdam
bastings@uva.nl

**Stefan Riezler**
Computational Linguistics & IWR
Heidelberg University
riezler@cl.uni-heidelberg.de

## Abstract

We present Joey NMT, a minimalist neural machine translation toolkit based on PyTorch that is specifically designed for novices. Joey NMT provides many popular NMT features in a small and simple code base, so that novices can easily and quickly learn to use it and adapt it to their needs. Despite its focus on simplicity, Joey NMT supports classic architectures (RNNs, transformers), fast beam search, weight tying, and more, and achieves performance comparable to more complex toolkits on standard benchmarks. We evaluate the accessibility of our toolkit in a user study where novices with general knowledge about Pytorch and NMT and experts work through a self-contained Joey NMT tutorial, showing that novices perform almost as well as experts in a subsequent code quiz. Joey NMT is available at https://github.com/joeynmt/joeynmt.

## 1 Introduction

Since the first successes of neural machine translation (NMT), various research groups and industry labs have developed open source toolkits specialized for NMT, based on new open source deep learning platforms. While toolkits like OpenNMT (Klein et al., 2018), XNMT (Neubig et al., 2018) and Neural Monkey (Helcl and Libovický, 2017) aim at readability and extensibility of their codebase, their target group are researchers with a solid background in machine translation and deep learning, and with experience in navigating, understanding and handling large code bases. However, none of the existing NMT tools has been designed primarily for readability or accessibility for novices, nor has anyone studied quality and accessibility of such code empirically. On the other hand, it is an important challenge for novices to understand how NMT is implemented, what features each toolkit implements exactly, and which

toolkit to choose in order to code their own project as fast and simple as possible.

We present an NMT toolkit especially designed for novices, providing clean, well documented, and minimalistic code, that is yet of comparable quality to more complex codebases on standard benchmarks. Our approach is to identify the core features of NMT that have not changed over the last years, and to invest in documentation, simplicity and quality of the code. These core features include standard network architectures (RNN, transformer, different attention mechanisms, input feeding, configurable encoder/decoder bridge), standard learning techniques (dropout, learning rate scheduling, weight tying, early stopping criteria), and visualization/monitoring tools.

We evaluate our codebase in several ways: Firstly, we show that Joey NMT's comment-to-code ratio is almost twice as high as other toolkits which are roughly 9-10 times larger. Secondly, we present an evaluation on standard benchmarks (WMT17, IWSLT) where we show that the core architectures implemented in Joey NMT achieve comparable performance to more complex state-of-the-art toolkits. Lastly, we conduct a user study where we test the code understanding of novices, i.e. students with basic knowledge about NMT and PyTorch, against expert coders. While novices, after having worked through a self-contained Joey NMT tutorial, needed more time to answer each question in an in-depth code quiz, they achieved only marginally lower scores than the experts. To our knowledge, this is the first user study on the accessibility of NMT toolkits.

## 2 Joey NMT

### 2.1 NMT Architectures

This section formalizes the Joey NMT implementation of autoregressive recurrent and fully-

attentional models.

In the following, a source sentence of length $l_x$ is represented by a sequence of one-hot encoded vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{l_x}$ for each word. Analogously, a target sequence of length $l_y$ is represented by a sequence of one-hot encoded vectors $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{l_y}$.

### 2.1.1 RNN

Joey NMT implements the RNN encoder-decoder variant from Luong et al. (2015).

**Encoder.** The encoder RNN transforms the input sequence $\mathbf{x}_1, \ldots, \mathbf{x}_{l_x}$ into a sequence of vectors $\mathbf{h}_1, \ldots, \mathbf{h}_{l_x}$ with the help of the embeddings matrix $E_{src}$ and a recurrent computation of states

$$\mathbf{h}_i = \mathrm{RNN}(E_{src}\,\mathbf{x}_i, \mathbf{h}_{i-1}); \qquad \mathbf{h}_0 = \mathbf{0}.$$

The RNN consists of either GRU or a LSTM units. For a bidirectional RNN, hidden states from both directions are are concatenated to form $\mathbf{h}_i$. The initial encoder hidden state $\mathbf{h}_0$ is a vector of zeros. Multiple layers can be stacked by using each resulting output sequence $\mathbf{h}_1, \ldots, \mathbf{h}_{l_x}$ as the input to the next RNN layer.

**Decoder.** The decoder uses input feeding (Luong et al., 2015) where an attentional vector $\tilde{\mathbf{s}}$ is concatenated with the representation of the previous word as input to the RNN. Decoder states are computed as follows:

$$\mathbf{s}_t = \mathrm{RNN}([E_{trg}\,\mathbf{y}_{t-1}; \tilde{\mathbf{s}}_{t-1}], \mathbf{s}_{t-1})$$

$$\mathbf{s}_0 = \begin{cases} \tanh(W_{bridge}\,\mathbf{h}_{l_x} + \mathbf{b}_{bridge}) & \text{if bridge} \\ \mathbf{h}_{l_x} & \text{if last} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

$$\tilde{\mathbf{s}}_t = \tanh(W_{att}[\mathbf{s}_t; \mathbf{c}_t] + \mathbf{b}_{att})$$

The initial decoder state is configurable to be either a non-linear transformation of the last encoder state ("bridge"), or identical to the last encoder state ("last"), or a vector of zeros.

**Attention.** The context vector $\mathbf{c}_t$ is computed with an attention mechanism scoring the previous decoder state $\mathbf{s}_{t-1}$ and each encoder state $\mathbf{h}_i$:

$$\mathbf{c}_t = \sum_i a_{ti} \cdot \mathbf{h}_i$$

$$a_{ti} = \frac{\exp(\mathrm{score}(\mathbf{s}_{t-1}, \mathbf{h}_i))}{\sum_k \exp(\mathrm{score}(\mathbf{s}_{t-1}, \mathbf{h}_k))}$$

where the scoring function is a multi-layer perceptron (Bahdanau et al., 2015) or a bilinear transformation (Luong et al., 2015).

**Output.** The output layer produces a vector $\mathbf{o}_t = W_{out}\,\tilde{\mathbf{s}}_t$, which contains a score for each token in the target vocabulary. Through a softmax transformation, these scores can be interpreted as a probability distribution over the target vocabulary $\mathcal{V}$ that defines an index over target tokens $v_j$.

$$p(y_t = v_j \mid x, y_{<t}) = \frac{\exp(\mathbf{o}_t[j])}{\sum_{k=1}^{|\mathcal{V}|} \exp(\mathbf{o}_t[k])}$$

### 2.1.2 Transformer

Joey NMT implements the Transformer from Vaswani et al. (2017), with code based on *The Annotated Transformer* blog (Rush, 2018).

**Encoder.** Given an input sequence $\mathbf{x}_1, \ldots, \mathbf{x}_{l_x}$, we look up the word embedding for each input word using $E_{src}\mathbf{x}_i$, add a position encoding to it, and stack the resulting sequence of word embeddings to form matrix $X \in \mathbb{R}^{l_x \times d}$, where $l_x$ is the sentence length and $d$ the dimensionality of the embeddings.

We define the following learnable parameters:[1]

$$A \in \mathbb{R}^{d \times d_a} \quad B \in \mathbb{R}^{d \times d_a} \quad C \in \mathbb{R}^{d \times d_o}$$

where $d_a$ is the dimensionality of the attention (inner product) space and $d_o$ the output dimensionality. Transforming the input matrix with these matrices into new word representations $H$

$$H = \underbrace{\mathrm{softmax}\big(XA\,B^\top X^\top\big)}_{\text{self-attention}} XC$$

which have been updated by attending to all other source words. Joey NMT implements multi-headed attention, where this transformation is computed $k$ times, one time for each head with different parameters $A, B, C$.

After computing all $k$ $H$s in parallel, we concatenate them and apply layer normalization and a final feed-forward layer:

$$H = [H^{(1)}; \ldots; H^{(k)}]$$
$$H' = \text{layer-norm}(H) + X$$
$$H^{(\mathrm{enc})} = \text{feed-forward}(H') + H'$$

We set $d_o = d/k$, so that $H \in \mathbb{R}^{l_x \times d}$. Multiple of these layers can be stacked by setting $X = H^{(\mathrm{enc})}$ and repeating the computation.

---

[1]Exposition adapted from Michael Collins `https://youtu.be/jfwqRMdTmLo`

**Decoder.** The Transformer decoder operates in a similar way as the encoder, but takes the stacked target embeddings $Y \in \mathbb{R}^{l_y \times d}$ as input:

$$H = \underbrace{\text{softmax}\left(YA\,B^\top Y^\top\right)}_{\text{masked self-attention}} YC$$

For each target position attention to future input words is inhibited by setting those attention scores to $-inf$ before the softmax. After obtaining $H' = H + Y$, and before the feed-forward layer, we compute multi-headed attention again, but now between intermediate decoder representations $H'$ and final encoder representations $H^{(\text{enc})}$:

$$Z = \underbrace{\text{softmax}\left(H'A\,B^\top H^{(\text{enc})\top}\right)}_{\text{src-trg attention}} H^{(\text{enc})}C$$

$$H^{(\text{dec})} = \text{feed-forward}(\text{layer-norm}(H' + Z))$$

We predict target words with $H^{(\text{dec})}W_{out}$.

## 2.2 Features

In the spirit of minimalism, we follow the 80/20 principle (Pareto, 1896) and aim to achieve 80% of the translation quality with 20% of a common toolkit's code size. For this purpose we identified the most common features (the bare necessities) in recent works and implementations. Table 6 in Appendix A.1 gives an overview over Joey NMT's features compared with several popular NMT toolkits. As can be seen, Joey NMT covers all features that those toolkits have in common. It includes standard architectures (see §2.1), label smoothing, dropout in multiple places, various attention mechanisms, input feeding, configurable encoder/decoder bridge, learning rate scheduling, weight tying, early stopping criteria, beam search decoding, an interactive translation mode, visualization/monitoring of learning progress and attention, checkpoint averaging, and more.

## 2.3 Documentation

The code itself is documented with doc-strings and in-line comments (especially for tensor shapes), and modules are tested with unit tests. The documentation website[2] contains installation instructions, a walk-through tutorial for training, tuning and testing an NMT model on a toy task[3], an overview of code modules, and a detailed API documentation. In addition, we provide thorough

---

[2] https://joeynmt.readthedocs.io
[3] Demo video: https://youtu.be/PzWRWSIwSYc

| Counts | OpenNMT-py | XNMT | Joey NMT |
|---|---|---|---|
| Files | 94 | 82 | 20 |
| Code | 10,287 | 11,628 | 2,250 |
| Comments | 3,372 | 4,039 | 1,393 |
| Comment/Code Ratio | 0.33 | 0.35 | **0.62** |

Table 1: Python code statistics for OpenNMT-py (commit hash `624a0b3a`), XNMT (`a87e7b94`) and Joey NMT (`e55b615`).

answers to frequently asked questions regarding usage, configuration, debugging, implementation details and code extensions, and recommend resources, such as data collections, PyTorch tutorials and NMT background material.

## 2.4 Code Complexity

In order to facilitate fast code comprehension and navigation (Wiedenbeck et al., 1999), Joey NMT objects have at most one level of inheritance. Table 1 compares Joey NMT with OpenNMT-py and XNMT (selected for their extensibility and thoroughness of documentation) in terms of code statistics, i.e. lines of Python code, lines of comments and number of files.[4] OpenNMT-py and XNMT have roughly 9-10x more lines of code, spread across 4-5x more files than Joey NMT . These toolkits cover more than the essential features for NMT (see §2.2), in particular for other generation or classification tasks like image captioning and language modeling. However, Joey NMT's comment-to-code ratio is almost twice as high, which we hope will give code readers better guidance in understanding and extending the code.

## 2.5 Benchmarks

Our goal is to achieve a performance that is comparable to other NMT toolkits, so that novices can start off with reliable benchmarks that are trusted by the community. This will allow them to build on Joey NMT for their research, should they want to do so. We expect novices to have limited resources available for training, i.e., not more than one GPU for a week, and therefore we focus on benchmarks that are within this scope. Pretrained models, data preparation scripts and configuration files for the following benchmarks will be made available on https://github.com/joeynmt/joeynmt.

---

[4] Using https://github.com/AlDanial/cloc

| System | Groundhog RNN | | Best RNN | | | Transformer | |
|---|---|---|---|---|---|---|---|
| | en-de | lv-en | layers | en-de | lv-en | en-de | lv-en |
| NeuralMonkey | 13.7 | 10.5 | 1/1 | 13.7 | 10.5 | – | – |
| OpenNMT-Py | 18.7 | 10.0 | 4/4 | 22.0 | 13.6 | – | – |
| Nematus | 23.9 | 14.3 | 8/8 | 23.8 | 14.7 | – | – |
| Sockeye | 23.2 | 14.4 | 4/4 | 25.6 | 15.9 | 27.5 | 18.1 |
| Marian | 23.5 | 14.4 | 4/4 | 25.9 | 16.2 | 27.4 | 17.6 |
| Tensor2Tensor | – | – | – | – | – | 26.3 | 17.7 |
| **Joey NMT** | 23.5 | 14.6 | 4/4 | 26.0 | 15.8 | 27.4 | 18.0 |

Table 2: Results on WMT17 `newstest2017`. Comparative scores are from Hieber et al. (2018).

**WMT17.** We use the settings of Hieber et al. (2018), using the exact same data, pre-processing, and evaluation using WMT17-compatible Sacre-BLEU scores (Post, 2018).[5] We consider the setting where toolkits are used out-of-the-box to train a Groundhog-like model (1-layer LSTMs, MLP attention), the 'best found' setting where Hieber et al. train each model using the best settings that they could find, and the Transformer base setting.[6] Table 2 shows that Joey NMT performs very well compared against other shallow, deep and Transformer models, despite its simple code base. [7]

**IWSLT14.** This is a popular benchmark because of its relatively small size and therefore fast training time. We use the data, pre-processing, and word-based vocabulary of Wiseman and Rush (2016) and evaluate with SacreBLEU.[8] Table 3 shows that Joey NMT performs well here, with both its recurrent and its Transformer model. We also included BPE results for future reference.

| System | de-en |
|---|---|
| Wiseman and Rush (2016) | 22.5 |
| Bahdanau et al. (2017) | 27.6 |
| **Joey NMT** (RNN, word) | 27.1 |
| **Joey NMT** (RNN, BPE32k) | 27.3 |
| **Joey NMT** (Transformer, BPE32k) | 31.0 |

Table 3: IWSLT14 test results.

---

[5] BLEU+case.mixed+lang.[en-lv|en-de]+numrefs.1+smooth.exp+test.wmt17+tok.13a+version.1.3.6

[6] Note that the scores reported for other models reflect their state when evaluated in Hieber et al. (2018).

[7] Blog posts like Rush (2018) and Bastings (2018) also offer simple code, but they do not perform as well.

[8] BLEU+case.lc+numrefs.1+smooth.exp+tok.none+version.1.3.6

# 3 User Study

The target group for Joey NMT are novices who will use NMT in a seminar project, a thesis, or an internship. Common tasks are to re-implement a paper, extend standard models by a small novel element, or to apply them to a new task. In order to evaluate how well novices understand Joey NMT, we conducted a user study comparing the code comprehension of novices and experts.

## 3.1 Study Design

**Participants.** The novice group is formed of eight undergraduate students with a Computational Linguistics major that have all passed introductory courses to Python and Machine Learning, three of them also a course about Neural Networks. None of them had practical experience with training or implementing NMT models nor PyTorch, but two reported theoretic understanding of NMT. They attended a 20h crash course introducing NMT and Pytorch basics (see §A.3 for details). Note that we did not teach Joey NMT explicitly in class, but the students independently completed the Joey NMT tutorial.

As a control group (the "experts"), six graduate students with NMT as topic of their thesis or research project participated in the study. In contrast to the novices, this group of participants has a solid background in Deep Learning and NMT, had practical experience with NMT. All of them had previously worked with NMT in PyTorch.

**Conditions.** The participation in the study was voluntary and not graded. Participants were not allowed to work in groups and had a maximum time of 3h to complete the quiz. They had pre-

viously locally installed Joey NMT[9] and could browse the code with the tools of their choice (IDE or text editor). They were instructed to explore the Joey NMT code with the help of the quiz, informed about the purpose of the study, and agreed to the use of their data in this study. Both groups of participants had to learn about Joey NMT in a self-guided manner, using the same tutorial, code, and documentation. The quiz was executed on the university's internal e-learning platform. Participants could jump between questions, review their answers before finally submitting all answers and could take breaks (without stopping the timer). Answers to the questions were published after all students had completed the test.

**Question design.** The questions are not designed to test the participant's prior knowledge on the topic, but to guide their exploration of the code. The questions are either free text, multiple choice or binary choice. There are three blocks of questions (§A.6 contains the full list):

1. **Usage of Joey NMT** : nine questions on how to interpret logs, check whether models were saved, interpret attention matrices, pre-/post-process, and to validate whether the model is doing what it is built for.

2. **Configuring Joey NMT** : four questions that make the users configure Joey NMT in such a way that it works for custom situations, e.g. with custom data, with a constant learning rate, or creating model of desired size.

3. **Joey NMT Code**: eighteen questions targeting the detailed understanding of the Joey NMT code: the ability to navigate between python modules, identify dependencies, and interpret what individual code lines are doing, hypothesize how specific lines in the code would have to get changed to change the behavior (e.g. working with a different optimizer). The questions in this block were designed in a way that in order to find the correct answers, every python module contained in Joey NMT had to be visited at least once.

Every question is awarded one point if answered correctly. Some questions require manual grading,

most of them have one correct answer. We record overall completion time and time per question.[10]

## 3.2 Analysis

**Total duration and score.** Experts took on average 77 min to complete the quiz, novices 118 min, which is significantly slower (one-tailed t-test, $p < 0.05$). Experts achieved on average 82% of the total points, novices 66% (details in §A.5). According to the t-test the difference in total scores between groups is significant at $p < 0.05$. An ANOVA reveals that there is a significant difference in total duration and scores within the novices group, but not within the experts group.

**Per question analysis.** No question was incorrectly answered by everyone. Three questions (#6, #11, #18) were correctly answered by everyone—they were appeared to be easiest to answer and did not require deep understanding of the code. In addition, seven questions (#1, #13, #15, #21, #22, #28, #29) were correctly answered by all experts, but not all novices—here their NMT experience was useful for working with hyperparameters and peculiarities like special tokens. However, for only one question, regarding the differences in data processing between training and validation (#16), the difference between average expert and novice score was significant (at $p < 0.05$). Six questions (#9, #18, #21, #25, #31) show a significantly longer average duration for novices than experts. These questions concerned post-processing, initialization, batching, end conditions for training termination and plotting, and required detailed code inspection.

**LME.** In order to analyze the dependence of scores and duration on particular questions and individual users, we performed a linear mixed effects (LME) analysis using the R library `lme4` (Bates et al., 2015) (details in §A.7). Participants and questions are treated as random effects (categorical), the level of expertise as fixed effect (binary). Duration and score per question are response variables. [11] For both response variables the variability is higher depending on the question than on the user (6x higher for score, 2x higher for

---

[10]Time measurement is noisy, since full minutes are measured and students might take breaks at various points in time.

[11]Modeling expertise with higher granularity instead of the binary classification into expertise groups (individual variables for experience with PyTorch, NMT and background in deep learning) did not have a significant effect on the model, since the number of participants is relatively low.

time). The intercepts of the fixed effects show that novices score on average 0.14 points less while taking 2.47 min longer on each question than experts. The impact of the fixed effect is significant at $p < 0.05$.

### 3.3 Findings

First of all, we observe that the design of the questions was engaging enough for the students because all participants invested at least 1h to complete the quiz voluntarily. The experts also reported having gained new insights into the code through the quiz. We found that there are significant differences between both groups: Most prominently, the novices needed more time to answer each question, but still succeeded in answering the majority of questions correctly. There are larger variances within the group of novices, because they had to develop individual strategies to explore the code and use the available resources (documentation, code search, IDE), while experts could in many cases rely on prior knowledge.

## 4 Conclusion

We presented Joey NMT, a toolkit for sequence-to-sequence learning designed for NMT novices. It implements the most common NMT features and achieves performance comparable to more complex toolkits, while being minimalist in its design and code structure. In comparison to other toolkits, it is smaller in size and but more extensively documented. A user study on code accessibility confirmed that the code is comprehensibly written and structured. We hope that Joey NMT will ease the burden for novices to get started with NMT, and can serve as a basis for teaching.

## Acknowledgments

## References

Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron C. Courville, and Yoshua Bengio. 2017. An actor-critic algorithm for sequence prediction. In *ICLR*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *ICLR*.

Joost Bastings. 2018. The annotated encoder-decoder with attention.

Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. 2015. Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67(1):1–48.

Jindřich Helcl and Jindřich Libovický. 2017. Neural Monkey: An Open-source Tool for Sequence Learning. *PBML*.

Felix Hieber, Tobias Domhan, Michael Denkowski, David Vilar, Artem Sokolov, Ann Clifton, and Matt Post. 2018. The sockeye neural machine translation toolkit at amta 2018. In *AMTA*.

Guillaume Klein, Yoon Kim, Yuntian Deng, Vincent Nguyen, Jean Senellart, and Alexander Rush. 2018. Opennmt: Neural machine translation toolkit. In *AMTA*.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *EMNLP*.

Graham Neubig, Matthias Sperber, Xinyi Wang, Matthieu Felix, Austin Matthews, Sarguna Padmanabhan, Ye Qi, Devendra Sachan, Philip Arthur, Pierre Godard, John Hewitt, Rachid Riad, and Liming Wang. 2018. XNMT: The extensible neural machine translation toolkit. In *AMTA*.

Vilfredo Pareto. 1896. *Cours d'économie politique: professé á l'Université de Lausanne*, volume 1. F. Rouge.

Matt Post. 2018. A call for clarity in reporting BLEU scores. In *WMT*.

Alexander Rush. 2018. The annotated transformer.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*.

Susan Wiedenbeck, Vennila Ramalingam, Suseela Sarasamma, and Cynthia L Corritore. 1999. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11(3):255–282.

Sam Wiseman and Alexander M. Rush. 2016. Sequence-to-sequence learning as beam-search optimization. In *EMNLP*, Austin, Texas.

# A Supplemental Material

## A.1 NMT Features

Table 6 gives an overview over Joey NMT's features compared with several popular NMT toolkits implemented in Python, such as Sockeye, Neural Monkey, fair-seq, tensor2tensor, XNMT and OpenNMT-py. Sockeye is based on MXNet, Neural Monkey and tensor2tensor on TensorFlow, XNMT on Dynet and fair-seq, OpenNMT-py and JoeyNMT on PyTorch. We filled the table to our best knowledge with information obtained from GitHub repositories, published papers and provided documentation.

## A.2 Extra Results

**WMT14.** WMT14 has been a popular benchmark to compare MT systems, even though different pre-/post-processing methods make comparisons noisy.[12] We train a recurrent 1-layer ("shallow") and 4-layer ("deep") model on the same data as Luong et al. (2015). Training the shallow models took about 5 days on one P40 GPU; the deep models took around 9 days. For comparative purposes we report (Moses-)tokenized and compound-splitted (only en-de) `multibleu` scores. Table 4 compares the Joey NMT models against GNMT and Luong et al. (2015). Without checkpoint averaging and extensive hyperparameter tuning, Joey NMT achieves results that come close to these systems.

| System | en-de | en-fr |
|---|---|---|
| Luong et al. (2015) | 18.1 | 31.5 |
| GNMT | 24.6 | 39.0 |
| **Joey NMT RNN** | 22.5 | 35.7 |
| **Joey NMT RNN** (deep) | 24.0 | 37.4 |

Table 4: `newstest2014` results.

**IWSLT En-Vi.** We also compared our RNNs against Tensorflow NMT and XNMT on the IWSLT15 en-vi data set as pre-processed by Stanford. Table 5 shows the results. The first three systems were trained on sentences of up to 50 tokens, while last two systems were trained on sentences of up to 110 tokens. Our BLEU scores were computed with SacreBLEU with version string BLEU+case.mixed+numrefs.1+smooth.exp+ tok.none+version.1.3.6. We use the original tokenization and data from `https://nlp.stanford.edu/projects/nmt`.

| System | en-vi |
|---|---|
| Luong et al. (2015) | 23.3 |
| TensorFlow NMT | 26.1 |
| **Joey NMT RNN** | 26.5 |
| XNMT | 27.3 |
| **Joey NMT RNN** | 27.7 |

Table 5: IWSLT en-vi

## A.3 Crash Course on NMT and Pytorch Basics

Prior to the study, the novices attended a three-day crash course (ca. 20h in total) where they were introduced to the concepts of feed-forward, recurrent and attentional neural networks, to PyTorch and the encoder-decoder model for sequence-to-sequence learning. In addition to lectures on the theory and background, they completed a subset of the PyTorch and RNN exercises of the Udacity course on

---

[12] See `https://github.com/tensorflow/tensor2tensor/issues/317` for a discussion on post-processing for en-de.

Deep Learning[13], so that they had all implemented and trained a feed-forward neural network for image classification and an LSTM for character-level language modeling. Solutions were discussed in class. In addition, they worked through the "The Annotated Encoder Decoder"[14] (Bastings, 2018) to get a grasp of the building blocks of a NMT implementation in PyTorch. Note that we did not teach Joey NMT explicitly in class, but the students had to self-sufficiently work through a Joey NMT tutorial[15].

## A.4 Quiz Interface

Let's say you have invented a new optimizer and implemented it in Pytorch as torch.optim.MagicOptimizer. Now you want to use it in JoeyNMT by setting "optimizer: magic" in the configuration file.

Which of JoeyNMT's Python files would you have to add the following lines to?

```
elif optimizer_name == "magic":
        # new awesome optimizer
        optimizer = torch.optim.MagicOptimizer(parameters, weight_decay=weight_decay, lr=learning_rate)
```

Antwort: 

How are forward and backward states combined for a bidirectional recurrent encoder? Choose the correct tensor operation:

Wählen Sie eine Antwort:
- ○ torch.addbmm
- ○ torch.sum
- ○ torch.add
- ○ torch.pow
- ○ torch.mul
- ○ torch.cat

Figure 1: A free-text and a multiple-choice question. Instructions "Antwort" (response) are in German since the interface of the e-learning platform is.

Figure 1 shows the interface for two example questions, one as a free-text question, and one as a multiple-choice task.

## A.5 Quiz Statistics

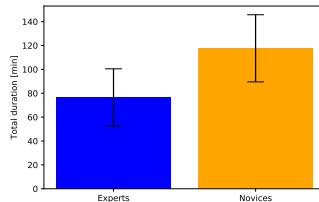Figure 2 compares the total completion time for the quiz, Figure 3 the total points between experts and novices.



Figure 2: Total duration of quiz taken by experts and novices.

---

[13]Parts 1-6 of the publicly available notebooks on https://github.com/udacity/deep-learning-v2-pytorch/tree/master/intro-to-pytorch and https://github.com/udacity/deep-learning-v2-pytorch/tree/master/recurrent-neural-networks/char-rnn, commit hash 9b6001a.

[14] https://github.com/bastings/annotated_encoder_decoder
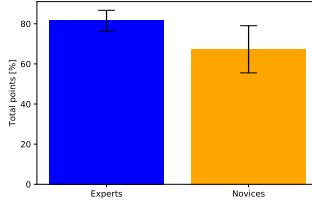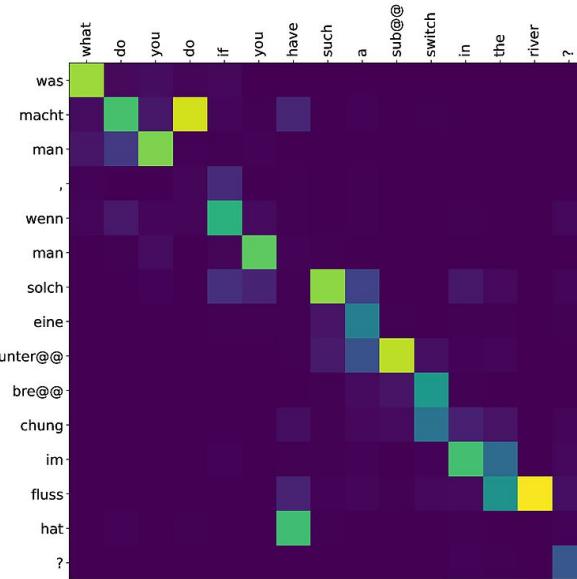
[15]https://JoeyNMT.readthedocs.io

Figure 3: Percentage of points scored by experts and novices.

### A.6 Quiz Questions

1. **Training.** You have successfully installed Joey NMT and written a configuration file `config.yaml`. Which command would you use to start training a model with this configuration?

   - `python3 -m Joey NMT train config.yaml`

2. **Translating.** Model training with `config.yaml` has finished and now you want to translate the pre-processed file `translate-me.txt` and save the translations in file `translated.txt` without specifying the file's path in the configuration file. Which command would you use?

   - `python3 -m Joey NMT translate config.yaml < translate-me.txt > translated.txt`

3. **Saving.** How do you know your model was saved during training?

   - ✓ Check in the validation report whether there's any line ending with "*".
   - ✓ Check the training log if it says it saved checkpoints.
   - ✓ Check if there are any `*.ckpt` files in the model directory.
   - ✗ The model always gets saved during training.

4. **Testing.** When using Joey NMT in test mode, can you specify the checkpoint for testing anywhere outside the configuration file?

   - ✓ True
   - ✗ False

5. **Parameters.** How many parameters does the model specified in `configs/default.yaml` have in total? This includes all parameter weights and biases of the model, including e.g. the embeddings. Hint: Joey NMT computes it for you.

   - 66,376

6. **Attention.** Which source token receives most attention when generating the target word "if"?

   - "wenn"

7. **Speed.** How do you find out how fast your model trains (including validations)?

   - The number of tokens per second is logged and reported in the log file.

8. **Pre-processing.** Which pre-processing does Joey NMT do for you? (if specified)

   - ✗ splitting into sub-word units (BPEs)
   - ✗ data filtering by source/target length ratio
   - ✓ data filtering by source and target length
   - ✗ tokenization

✓ lowercasing

9. **Post-processing.** Which post-processing does Joey NMT for you? (if specified)

   ✗ recasing
   ✗ detokenization
   ✓ subword merging ("un-BPE-ing")
   ✗ delemmatization

10. **Checkpoints.** In a debugging scenario, you don't want to store checkpoints for your current model. There's a line that you can add to your configuration file to make the model not save any checkpoints during training. What is this line?

    - `keep_last_ckpts: 0`

11. **Model size.** Change the following model configuration to use three encoder layers.

```
encoder:
  rnn_type: "lstm"
  embeddings:
    embedding_dim: 16
  hidden_size: 64
  bidirectional: True
```

Which line would you have to add?

- `num_layers: 3`

12. **Data Path.** Which line would you have to add to the data configuration below to use my_home/my_dir/my_data.en as test input file?

```
data:
  src: "en"
  trg: "fr"
  train: "test/data/reverse/train"
  dev: "test/data/reverse/dev"
```

```
level: "word"
lowercase: False
max_sent_length: 25
```

Hint: mind the file ending.

- `test: "my_home/my_dir/my_data"`

13. **Training hyperparameters.** Modify the following training configuration such that it uses a constant learning rate of 0.02.

```
training:
  optimizer: "adam"
  learning_rate: 0.001
  clip_grad_norm: 1.0
  batch_size: 10
  scheduling: "plateau"
  patience: 5
  decrease_factor: 0.5
  early_stopping_metric: "eval_metric"
  epochs: 6
  validation_freq: 1000
  logging_freq: 100
  model_dir: "reverse_model"
  max_output_length: 30
```

Paste the modified configuration below.

- ```
  training:
    optimizer: "adam"
    learning_rate: 0.02
    clip_grad_norm: 1.0
    batch_size: 10
    patience: 5
    decrease_factor: 0.5
    early_stopping_metric: "eval_metric"
    epochs: 6
    validation_freq: 1000
    logging_freq: 100
    model_dir: "reverse_model"
    max_output_length: 30
  ```

14. **Vocabulary generation.** When the vocabulary is extracted from training data, we keep only the src_voc_limit / trg_voc_limit most frequent tokens that occur at least src_min_freq / trg_min_freq times in the training data.

For the example, the vocabulary limit is 15, while the minimum frequency is 3.

After counting the tokens in the training data and filtering by minimum frequency, we have the following counts:

```
i: 22
you: 14
and: 9
```

```
,: 9
to: 7
if: 6
joey: 5
't: 5
anymore: 5
're: 4
scared: 4
be: 4
angry: 4
but: 3
it: 3
out: 3
don: 3
get: 3
oh: 3
the: 3
```

Which of those tokens would *not* end up in the vocabulary, according to Joey NMT 's vocabulary building?

- ✓ it
- ✓ the
- ✓ oh
- ✓ get
- ✗ don
- ✗ but
- ✓ out

15. **Special tokens.** Which is the default token used for marking the end-of-sequence position in Joey NMT ? e.g. $<$end$>$ or $[$EOS$]$?

    - $</$s$>$

16. **Data iterators.** Training and validation data are treated differently in Joey NMT - but in which ways? For example, if you choose "sorting", it means that validation and training data are handled differently with respect to sorting - one gets sorted and the other doesn't.

    - ✓ Shuffling
    - ✓ Filtering
    - ✗ Tokenization
    - ✗ Embedding
    - ✓ Sorting

17. **Training loop.** Where is the training for-loop over epochs defined? Paste the line in the textbox below. (Not the line number)

    - `for epoch_no in range(self.epochs):`
      (in `training.py`)

18. **End of training.** When does training end? (Assuming there are no technical problems like memory errors etc.) We refer to settings in the configuration file, e.g. `learning_rate`.

    - ✓ When the minimum learning rate (`learning_rate_min`) has been reached.

✗ When the maximum validation scores has been reached.

✗ Just after `keep_last_ckpts` checkpoints have been saved.

✗ When Joey NMT gets tired.

✓ When all epochs (`epochs`) have been completed.

✓ When you interrupt the training process with Ctrl+C.

19. **Model.** What does `model.forward()` return?

   - decoder outputs, decoder last hidden state, attention probabilities, attentional vectors

20. **Initialization.** How are forget gates of LSTMs initialized by default?

   ✓ All ones

   ✗ Random normal initialization

   ✗ Random uniform initialization

   ✗ All zeros

   ✗ Xavier initialization

21. **Embeddings.** In the configuration we can "freeze" the embeddings, so that they are not (further) trained:

```
embeddings:
  embedding_dim: 16
  freeze: True
```

Where does the freezing happen in JoeyNMT's code? Please give the freezing function's name.

   - `freeze_params`

22. **Bidirectional.** How are forward and backward states combined for a bidirectional recurrent encoder? Choose the correct tensor operation:

   ✗ `torch.add`

   ✓ `torch.cat`

   ✗ `torch.addbmm`

   ✗ `torch.sum`

   ✗ `torch.mul`

   ✗ `torch.pow`

23. **Bridge.** What's the name of the function that connects encoder and decoder by computing the initial decoder state given the last encoder state?

   ✗ `bridge_layer`

   ✗ `BahdanauAttention`

   ✗ `init_decoder_hidden`

   ✗ `bridge_layer`

   ✗ `_bridge`

   ✗ `_init_decoder_hidden`

   ✗ `init_hidden`

   ✓ `_init_hidden`

   ✗ `bridge`

   ✗ `LuongAttention`

   ✗ `_attend`

✗ `_forward_step`

24. **Loss computation.** Find the place where the batch loss is computed (comparing model outputs with targets), and paste the statement below. e.g. `train_batch_loss = my_loss_function(outputs, targets)`

    - ```
      batch_loss = loss_function(
          input=log_probs.contiguous().view(-1, log_probs.size(-1)),
          target=batch.trg.contiguous().view(-1))
      ```

25. **Batch.** During training, the Batch object in JoeyNMT holds the reference sequence in `trg` for computing the loss and in `trg_input` for feeding it into the decoder.

    What's the difference between those two tensors? (`batch.trg` vs. `batch.trg_input`)

    ✗ <s> is prepended to the first, otherwise no difference
    ✓ </s> is appended to the first and ¡s¿ is prepended to the second
    ✗ <s> is prepended to the second, otherwise no difference
    ✗ <s> is appended to the first and ¡/s¿ is prepended to the second
    ✗ </s> is appended to the first, otherwise no difference

26. **Inference algorithm.** Where in the code is the decision made whether to decode greedily or with beam search? Paste the line below.

    Hint: it's an if-statement.

    - `if beam_size == 0:`

27. **Validation score computation.** Find the place where the validation score (here BLEU, `eval_metric: bleu` is computed and paste the statement below.

    - `current_valid_score = bleu(valid_hypotheses, valid_references)`

28. **BLEU computation.** Which library is used for BLEU score computation?

    - `sacrebleu`

29. **Optimizers.** Let's say you have invented a new optimizer and implemented it in Pytorch as `torch.optim.MagicOptimizer`. Now you want to use it in JoeyNMT by setting `optimizer: magic` in the configuration file.

    Which of JoeyNMT's Python files would you have to add the following lines to?

    ```
    elif optimizer_name == "magic":
        # new awesome optimizer
        optimizer=torch.optim.MagicOptimizer(
            parameters, weight_decay=weight_decay, lr=learning_rate)
    ```

    - `builders.py`

30. **Attention.** For Bahdanau attention, find the line where the attention scores for a decoder hidden state are computed (before masking).

    - ```
      scores = self.energy_layer(
          torch.tanh(self.proj_query + self.proj_keys))
      ```

31. **Plotting.** You want to use a different colormap for attention visualization, namely the one called ``binary''. Give the line of JoeyNMT's code that is responsible for plotting the attention, modified to use the new colormap.

    - ```
      plt.imshow(scores, cmap='binary', aspect='equal',
          origin='upper', vmin=0., vmax=1.)
      ```

### A.7 LMEM Details

### A.8 Score per Question

```
Linear mixed model fit by REML ['lmerMod']
Formula: score ~ group + (1 | item) + (1 | user)

Random effects:
 Groups   Name         Variance Std.Dev.
 item     (Intercept) 0.03668  0.1915
 user     (Intercept) 0.00661  0.0813
 Residual             0.12191  0.3492
Number of obs: 434, groups:  item, 31; user, 14

Fixed effects:
            Estimate Std. Error t value
(Intercept)  0.81532    0.05422  15.036
groupnovice -0.14258    0.05545  -2.571

Correlation of Fixed Effects:
            (Intr)
groupnovice -0.584

all.model_score0: score ~ (1 | item) + (1 | user)
all.model_score: score ~ group + (1 | item) + (1 | user)
                 Df    AIC    BIC  logLik deviance  Chisq Chi Df Pr(>Chisq)
all.model_score0  4 394.39 410.68 -193.20   386.39
all.model_score   5 390.50 410.87 -190.25   380.50 5.8904      1    0.01522 *
```

### A.9 Time per Question

```
Linear mixed model fit by REML ['lmerMod']
Formula: time ~ group + (1 | item) + (1 | user)

Random effects:
 Groups   Name         Variance Std.Dev.
 item     (Intercept)  0.8800   0.9381
 user     (Intercept)  0.4834   0.6953
 Residual             11.2855   3.3594
Number of obs: 434, groups:  item, 31; user, 14

Fixed effects:
            Estimate Std. Error t value
(Intercept)   2.4677     0.4119   5.992
groupnovice   1.3266     0.4972   2.668

Correlation of Fixed Effects:
            (Intr)
groupnovice -0.690

all.model_time0: time ~ (1 | item) + (1 | user)
all.model_time: time ~ group + (1 | item) + (1 | user)
                Df    AIC    BIC  logLik deviance  Chisq Chi Df Pr(>Chisq)
all.model_time0  4 2330.1 2346.4 -1161.0   2322.1
```

```
all.model_time    5 2325.7 2346.1 -1157.8   2315.7 6.3868      1    0.0115 *
```

| Feature | Sockeye | Neural Monkey | fair-seq | T2T | XNMT | OpenNMT-py | Joey NMT |
|---|---|---|---|---|---|---|---|
| *Architecture* | | | | | | | |
| RNN encoder | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| RNN decoder | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Transformer encoder | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Transformer decoder | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ConvS2S encoder | ✓ | ✓ | ✓ | | | ✓ | |
| ConvS2S decoder | ✓ | | ✓ | | | ✓ | |
| Image Encoder | ✓ | ✓ | | ✓ | | ✓ | |
| Audio Encoder | | ✓ | | ✓ | ✓ | ✓ | |
| CTC | | ✓ | | | | | |
| Attention Mechanisms | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| *Tasks* | | | | | | | |
| Embedding Tying | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ |
| Softmax Tying | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Parameter Freezing | ✓ | | ✓ | | | | ✓ |
| Multi-Source | | ✓ | ✓ | | ✓ | | |
| Factored Input | ✓ | ✓ | | | | ✓ | |
| Multi-Task | | ✓ | | ✓ | ✓ | | |
| Sequence Labeling | | ✓ | | | | | |
| Sequence Classification | | ✓ | | | ✓ | | |
| Language Modeling | | ✓ | ✓ | ✓ | ✓ | ✓ | |
| *Inference* | | | | | | | |
| Segmentation Levels (word/char/bpe) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Beam Search | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| n-best outputs | ✓ | ✓ | ✓ | | | | |
| Sampling | ✓ | ✓ | ✓ | | ✓ | ✓ | |
| Rescoring | ✓ | ✓ | | | ✓ | | |
| Checkpoint averaging | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| *Training* | | | | | | | |
| MLE | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| MRT | | ✓ | | | ✓ | | |
| Gradient Clipping | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Dropout | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Weight Decay | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Label Smoothing | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Optimizer | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Scheduler | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Early Stopping | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| *Usage* | | | | | | | |
| CPU/GPU | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Monitoring | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Attention Visualization | ✓ | ✓ | | ✓ | ✓ | | ✓ |

Table 6: Features implemented by popular NMT toolkits in Python as of July 1, 2019.