The SOCKEYE Neural Machine Translation Toolkit at AMTA 2018

Felix Hieber, Tobias Domhan, Michael Denkowski {fhieber,domhant,mdenkows}@amazon.com
David Vilar, Artem Sokolov, Ann Clifton, Matt Post {dvilar,artemsok,acclift,mattpost}@amazon.com

Abstract

We describe SOCKEYE,[1] an open-source sequence-to-sequence toolkit for Neural Machine Translation (NMT). SOCKEYE is a production-ready framework for training and applying models as well as an experimental platform for researchers. Written in Python and built on MXNET, the toolkit offers scalable training and inference for the three most prominent encoder-decoder architectures: attentional recurrent neural networks, self-attentional transformers, and fully convolutional networks. SOCKEYE also supports a wide range of optimizers, normalization and regularization techniques, and inference improvements from current NMT literature. Users can easily run standard training recipes, explore different model settings, and incorporate new ideas. The SOCKEYE toolkit is free software released under the Apache 2.0 license.

1 Introduction

For all its success, Neural Machine Translation (NMT) presents a range of new challenges. While popular encoder-decoder models are attractively simple, recent literature and the results of shared evaluation tasks show that a significant amount of engineering is required to achieve "production-ready" performance in both translation quality and computational efficiency. In a trend that carries over from Statistical Machine Translation (SMT), the strongest NMT systems benefit from subtle architecture modifications, hyper-parameter tuning, and empirically effective heuristics. To address these challenges, we introduce SOCKEYE, a neural sequence-to-sequence toolkit written in Python and built on Apache MXNET[2] [Chen et al., 2015]. To the best of our knowledge, SOCKEYE is the only toolkit that includes implementations of all three major neural translation architectures: attentional recurrent neural networks [Schwenk, 2012, Kalchbrenner and Blunsom, 2013, Sutskever et al., 2014, Bahdanau et al., 2014, Luong et al., 2015], self-attentional transformers [Vaswani et al., 2017], and fully convolutional networks [Gehring et al., 2017]. These implementations are supported by a wide and continually updated range of features reflecting the best ideas from recent literature. Users can easily train models based on the latest research, compare different architectures, and extend them by adding their own code. SOCKEYE is under active development that follows best practice for both research and production software, including clear coding and documentation guidelines, comprehensive automatic tests, and peer review for code contributions.

[1] `https://github.com/awslabs/sockeye` (version 1.12)
[2] `https://mxnet.incubator.apache.org/`

2 Encoder-Decoder Models for NMT

The main idea in neural sequence-to-sequence modeling is to *encode* a variable-length input sequence of tokens into a sequence of vector representations, and to then *decode* those representations into a sequence of output tokens. We give a brief description for the three encoder-decoder architectures as implemented in SOCKEYE, but refer the reader to the references for more details or to the arXiv version of this paper [Hieber et al., 2017].

2.1 Stacked Recurrent Neural Network (RNN) with Attention [Bahdanau et al., 2014, Luong et al., 2015]

The encoder consists of a bi-directional RNN followed by a stack of uni-directional RNNs. An RNN at layer l produces a sequence of hidden states $\mathbf{h}_1^l \dots \mathbf{h}_n^l$:

$$\mathbf{h}_i^l = f_{enc}(\mathbf{h}_i^{l-1}, \mathbf{h}_{i-1}^l), \tag{1}$$

where f_{rnn} is some non-linear function, such as a Gated Recurrent Unit (GRU) or Long Short Term Memory (LSTM) cell, and $\mathbf{h}_i^{l-1}$ is the output from the lower layer at step i. The bi-directional RNN on the lowest layer uses the embeddings $\mathbf{E}_S\mathbf{x}_i$ as input and concatenates the hidden states of a forward and a reverse RNN: $\mathbf{h}_i^0 = [\overrightarrow{\mathbf{h}}_i^0; \overleftarrow{\mathbf{h}}_i^0]$. With deeper networks, learning turns increasingly difficult [Hochreiter et al., 2001, Pascanu et al., 2012] and residual connections of the form $\mathbf{h}_i^l = \mathbf{h}_i^{l-1} + f_{enc}(\mathbf{h}_i^{l-1}, \mathbf{h}_{i-1}^l)$ become essential [He et al., 2016].

The decoder consists of an RNN to predict one target word at a time through a state vector $\mathbf{s}$:

$$\mathbf{s}_t = f_{dec}([\mathbf{E}_T\mathbf{y}_{t-1}; \bar{\mathbf{s}}_{t-1}], \mathbf{s}_{t-1}), \tag{2}$$

where f_{dec} is a multi-layer RNN, $\mathbf{s}_{t-1}$ the previous state vector, and $\bar{\mathbf{s}}_{t-1}$ the source-dependent *attentional vector*. Providing the attentional vector as an input to the first decoder layer is also called *input feeding* [Luong et al., 2015]. The initial decoder hidden state is a non-linear transformation of the last encoder hidden state: $\mathbf{s}_0 = \tanh(\mathbf{W}_{init}\mathbf{h}_n + \mathbf{b}_{init})$. The attentional vector $\bar{\mathbf{s}}_t$ combines the decoder state with a *context vector* $\mathbf{c}_t$:

$$\bar{\mathbf{s}}_t = \tanh(\mathbf{W}_{\bar{s}}[\mathbf{s}_t; \mathbf{c}_t]), \tag{3}$$

where $\mathbf{c}_t$ is a weighted sum of encoder hidden states: $\mathbf{c}_t = \sum_{i=1}^{n} \alpha_{ti}\mathbf{h}_i$. The attention vector $\boldsymbol{\alpha}_t$ is computed by an attention network [Bahdanau et al., 2014, Luong et al., 2015]:

$$\begin{aligned} \alpha_{ti} &= \mathrm{softmax}(\mathrm{score}(\mathbf{s}_t, \mathbf{h}_i)) \\ \mathrm{score}(\mathbf{s}, \mathbf{h}) &= \mathbf{v}_a^\top \tanh(\mathbf{W}_u\mathbf{s} + \mathbf{W}_v\mathbf{h}). \end{aligned} \tag{4}$$

2.2 Self-attentional Transformer

The transformer model [Vaswani et al., 2017] uses attention to replace recurrent dependencies, making the representation at time step i independent from the other time steps. This requires the explicit encoding of positional information in the sequence by adding fixed or learned positional embeddings to the embedding vectors.

The encoder uses several identical blocks consisting of two core sublayers, self-attention and a feed-forward network. The self-attention mechanism is a variation of the dot-product attention [Luong et al., 2015] generalized to three inputs: a query matrix $\mathbf{Q} \in \mathbb{R}^{n \times d}$, a key matrix $\mathbf{K} \in \mathbb{R}^{n \times d}$, and a value matrix $\mathbf{V} \in \mathbb{R}^{n \times d}$, where d denotes the number of hidden units. Vaswani et al. [2017] further extend attention to multiple *heads*, allowing for focusing on different parts of the input. A single *head* u produces a context matrix

$$\mathbf{C}_u = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{W}_u^Q(\mathbf{K}\mathbf{W}_u^K)^\top}{\sqrt{d_u}}\right)\mathbf{V}\mathbf{W}_u^V, \tag{5}$$

where matrices $\mathbf{W}_u^Q, \mathbf{W}_u^K$ and $\mathbf{W}_u^V$ are in $\mathbb{R}^{d \times d_u}$. The final context matrix is given by concatenating the heads, followed by a linear transformation: $\mathbf{C} = [\mathbf{C}_1; \ldots; \mathbf{C}_h]\mathbf{W}^O$. The form in Equation 5 suggests parallel computation across all time steps in a single large matrix multiplication. Given a sequence of hidden states $\mathbf{h}_i$ (or input embeddings), concatenated to $\mathbf{H} \in \mathbb{R}^{n \times d}$, the encoder computes self-attention using $\mathbf{Q} = \mathbf{K} = \mathbf{V} = \mathbf{H}$. The second subnetwork of an encoder block is a feed-forward network with ReLU activation defined as

$$FFN(\mathbf{x}) = \max(0, \mathbf{x}\mathbf{W}_1 + \mathbf{b_1})\mathbf{W}_2 + \mathbf{b}_2, \tag{6}$$

which is also easily parallelizable across time steps. Each sublayer, self-attention and feed-forward network, is followed by a post-processing stack of dropout, layer normalization [Ba et al., 2016], and residual connection.

The decoder uses the same self-attention and feed-forward networks subnetworks. To maintain auto-regressiveness of the model, self-attention on future time steps is masked out accordingly [Vaswani et al., 2017]. In addition to self-attention, a source attention layer which uses the encoder hidden states as key and value inputs is added. Given decoder hidden states $\mathbf{S} \in \mathbb{R}^{m \times s}$ and the encoder hidden states of the final encoder layer $\mathbf{H}^l$, source attention is computed as in Equation 5 with $\mathbf{Q} = \mathbf{S}, \mathbf{K} = \mathbf{H}^l, \mathbf{V} = \mathbf{H}^l$. As in the encoder, each sublayer is followed by a post-processing stack of dropout, layer normalization [Ba et al., 2016], and residual connection.

2.3 Fully Convolutional Models (ConvSeq2Seq)

The convolutional model [Gehring et al., 2017] uses convolutional operations and also dispenses with recurrency. Hence, input embeddings are again augmented with explicit positional encodings.

The convolutional encoder applies a set of (stacked) convolutions that are defined as:

$$\mathbf{h}_i^l = v(\mathbf{W}^l[\mathbf{h}_{i-\lfloor k/2 \rfloor}^{l-1}; ...; \mathbf{h}_{i+\lfloor k/2 \rfloor}^{l-1}] + \mathbf{b}^l) + \mathbf{h}_i^{l-1}, \tag{7}$$

where v is a non-linearity such as a Gated Linear Unit (GLU) [Gehring et al., 2017, Dauphin et al., 2016] and $\mathbf{W}^l \in \mathbb{R}^{d_{cnn} \times kd}$ the convolutional filters. To increase the context window captured by the encoder architecture, multiple layers of convolutions are stacked. To maintain sequence length across multiple stacked convolutions, inputs are padded with zero vectors.

The decoder is similar to the encoder but adds an attention mechanism to every layer. The output of the target side convolution

$$\mathbf{s}_t^{l*} = v(\mathbf{W}^l[\bar{\mathbf{s}}_{t-k+1}^{l-1}; ...; \bar{\mathbf{s}}_t^{l-1}] + \mathbf{b}^l) \tag{8}$$

is combined to form $\mathbf{S}^*$ and then fed as an input to the attention mechanism of Equation 5 with a single attention head and $\mathbf{Q} = \mathbf{S}^*, \mathbf{K} = \mathbf{H}^l, \mathbf{V} = \mathbf{H}^l$, resulting in a set of context vectors $\mathbf{c}_t$. The full decoder hidden state is a residual combination with the context such that

$$\bar{\mathbf{s}}_t^l = \mathbf{s}_t^{l*} + \mathbf{c}_t + \bar{\mathbf{s}}_t^{l-1}. \tag{9}$$

To avoid convolving over future time steps at time t, the input is padded to the left.

3 The SOCKEYE toolkit

In addition to the currently supported architectures introduced in Section 2, SOCKEYE contains a number of model features (Section 3.1), training features (Section 3.2), and inference features (Section 3.3). See the public code repository[3] for a more detailed manual on how to use these features and the references for detailed descriptions.

[3] https://github.com/awslabs/sockeye/

3.1 Model features

Layer and weight normalization To speed up convergence of stochastic gradient descent (SGD) learning methods, SOCKEYE implements two popular techniques: layer normalization [Ba et al., 2016] and weight normalization [Salimans and Kingma, 2016]. If a neuron implements a non-linear mapping $f(\mathbf{w}^\top \mathbf{x} + b)$, its input weights $\mathbf{w}$ are transformed, for layer normalization, as $\mathbf{w}_i \leftarrow (\mathbf{w}_i - \text{mean}(\mathbf{w}))/\text{var}(\mathbf{w})$, where the weight mean and variance are calculated over all input weights of the neuron; and, for weight normalization, as $\mathbf{w} = g \cdot \mathbf{v}/||\mathbf{v}||$, where the scalar g and the vector $\mathbf{v}$ are neuron's new parameters.

Weight tying Sharing weights of the input embedding layer and the top-most output layer has been shown to improve language modeling quality [Press and Wolf, 2016] and to reduce memory consumption for NMT. It is implemented in SOCKEYE by setting $\mathbf{W}_o = \mathbf{E}_T$. For jointly-built BPE vocabularies [Sennrich et al., 2017a], SOCKEYE also allows setting $\mathbf{E}_S = \mathbf{E}_T$.

RNN attention types Attention is a core component of NMT systems. Equation 4 gave the basic mechanism for attention for RNN based architectures, but a wider family of functions can be used to compute the score function. SOCKEYE supports the following attention types: MLP, dot, bilinear, and location from Luong et al. [2015] and multi-head from Vaswani et al. [2012].[4]

RNN context gating Tu et al. [2017] introduce context gating for RNN models as a way to better guide the generation process by selectively emphasizing source or target contexts. This is accomplished by introducing a gate $\mathbf{z}_t = \sigma(\mathbf{W}_z \mathbf{E}_T(y_{t-1}) + \mathbf{U}_z \mathbf{s}_{i-1} + \mathbf{C}_z \bar{\mathbf{s}}_t)$, where $\mathbf{W}_z$, $\mathbf{U}_z$ and $\mathbf{C}_z$ are trainable weight matrices. SOCKEYE implements the "both" variant of Tu et al. [2017], where $\mathbf{z}_t$ multiplies the hidden state $\mathbf{s}_t$ of the decoder and $1 - \mathbf{z}_t$ multiplies the source context $\mathbf{h}_t$.

RNN attention feeding When training multi-layer RNN systems, there are several possibilities for selecting the hidden state that is used for computation of the attention score. By default SOCKEYE uses the last decoder RNN layer and combines the computed attention with the hidden RNN state in a feed-forward layer similar to Luong et al. [2015]. Following Wu et al. [2016], SOCKEYE also supports the option to use the first layer of the decoder to compute the attention score, which is then fed to the upper decoder layers.

3.2 Training features

Optimizers SOCKEYE can train models using any optimizer from MXNET's library, including stochastic gradient descent (SGD) and Adam [Kingma and Ba, 2014]. SOCKEYE also includes its own implementation of the Eve optimizer, which extends Adam by incorporating information from the objective function [Koushik and Hayashi, 2016]. This allows learning to accelerate over flat areas of the loss surface and decelerate when saddle points cause the objective to "bounce".

Learning schedules Recent work has shown the value of annealing the base learning rate α throughout training, even when using optimizers such as Adam [Vaswani et al., 2017, Denkowski and Neubig, 2017]. SOCKEYE's 'plateau-reduce' scheduler implements rate annealing as follows. At each training checkpoint, the scheduler compares validation set perplexity against the best previous checkpoint. If perplexity has not surpassed the previous best in N checkpoints, the learning rate α is multiplied by a fixed constant and the counter is reset. Optionally, the scheduler can reset model and optimizer parameters to the best previous point, simulating a perfect prediction of when to anneal the learning rate.

[4] Note that the transformer and convolutional architectures cannot use these attention types.

Monitoring Training progress is tracked in a metrics file that contains statistics computed at each checkpoint. It includes the training and validation perplexities, total time elapsed, and optionally a BLEU score on the validation data. To monitor BLEU scores, a subprocess is started at every checkpoint that decodes the validation data and computes BLEU. Note that this is an approximate BLEU score, as source and references are typically tokenized and possibly byte-pair encoded. All statistics can also be written to a Tensorboard event file that can be rendered by a standalone Tensorboard fork.[5]

Regularization SOCKEYE supports standard techniques for regularization, such as dropout. This includes dropout on input embeddings for both the source and the target and the proposed dropout layers for the transformer architecture. One can also enable *variational dropout* [Gal and Ghahramani, 2016] to sample a fixed dropout mask across recurrent time steps, or *recurrent dropout without memory loss* [Semeniuta et al., 2016]. SOCKEYE can also use MXNET's *label smoothing* [Pereyra et al., 2017] feature to efficiently back-propagate smoothed cross-entropy gradients without explicitly representing a dense, smoothed label distribution.

Fault tolerance SOCKEYE saves the training state of all training components after every checkpoint, including elements like the shuffling data iterator and optimizer states. Training can therefore easily be continued from the last checkpoint in the case of aborted process.

Mult-GPU training SOCKEYE can take advantage of multipe GPUs using MXNET's data parallelism mechanism. Training batches are divided into equal-sized chunks and distributed to the different GPUs which perform the computations in parallel.[6]

3.3 Inference features

SOCKEYE supports beam search on CPUs and GPUs through MXNET. Our beam search implementation is optimized to make use of MXNET's symbolic and imperative API and uses its operators as much as possible to let the MXNET framework efficiently schedule operations. Hypotheses in the beam are length-normalized with a configurable length penalty term as in Wu et al. [2016].

Ensemble decoding SOCKEYE supports both linear and log-linear ensemble decoding, which combines word predictions from multiple models. Models can use different architectures, but must use the same target vocabulary.

Batch decoding Batch decoding allows decoding multiple sentences at once. This is particularly helpful for large translation jobs such as back-translation [Sennrich et al., 2015], where throughput is more important than latency.

Vocabulary selection Each decoding time step requires the translation model to produce a distribution over target vocabulary items. This output layer requires matrix operations dominated by the size of the target vocabulary, $|\mathbf{V}_{trg}|$. One technique for reducing this computational cost involves using only a subset of the target vocabulary, $\mathbf{V}'_{trg}$, for each sentence based on the source [Devlin, 2017]. SOCKEYE can use a probabilistic translation table[7] for dynamic vocabulary selection during decoding. For each input sentence, $\mathbf{V}'_{trg}$ is limited to the top K translations for each source word, reducing the size of output layer matrix operations by 90% or more.

Attention visualization In cases where the attention matrix of RNN models is used in downstream applications, it is often useful to evaluate its soft alignments. For this, SOCKEYE supports returning or visualizing the attention matrix of RNN models.

[5] `https://github.com/dmlc/tensorboard`

[6] It is important to adapt the batch size accordingly.

[7] For example, as produced by `fast_align` [Dyer et al., 2013].

Toolkit	Architecture	EN→DE	LV→EN
FAIRSEQ	CNN	23.37	15.38
MARIAN	RNN	25.93	16.19
	Transformer	27.41	17.58
NEMATUS	RNN	23.78	14.70
NEURALMONKEY	RNN	13.73	10.54
OPENNMT-LUA	RNN	22.69	13.85
OPENNMT-PY	RNN	21.95	13.55
T2T	Transformer	26.34	17.67
SOCKEYE	CNN	24.59	15.82
	RNN	25.55	15.92
	Transformer	27.50	18.06

Table 1: BLEU scores for evaluated toolkits and architectures using "best found" settings on WMT newstest2017.

4 Experiments and comparison to other toolkits

We benchmarked each of SOCKEYE's supported architectures against other popular open-source toolkits on two language directions: English into German (EN→DE) and Latvian into English (LV→EN). Models in both language pairs were based on the complete parallel data provided for each task as part of the Second Conference on Machine Translation [Bojar et al., 2017]. We ran each toolkit in a "best available" configuration, where we took settings from recent relevant papers that used the toolkit, or communicated directly with authors. Toolkits evaluated include FAIRSEQ [Gehring et al., 2017], MARIAN [Junczys-Dowmunt et al., 2016], NEMATUS [Sennrich et al., 2017b], NEURALMONKEY [Helcl and Jindřich Libovický, 2017], OPENNMT [Klein et al., 2017], and TENSOR2TENSOR (T2T)[8].[9] Shown in Table 1, SOCKEYE's RNN model is competitive with MARIAN, the top-performer, the CNN model outperforms FAIRSEQ, and the transformer outperforms all models from all other toolkits. See [Hieber et al., 2017] for more extensive comparisons and further details.

5 Summary

We have presented SOCKEYE, a mature, open-source framework for neural sequence-to-sequence learning that implements the three major architectures for neural machine translation (the only one to do so, to our knowledge). Written in Python, it is easy to install, extend, and deploy; built on top of MXNET, it is fast parallelizable across GPUs. In the interest of future comparisons and cooperative development through friendly competition, we have provided the system outputs and training and evaluation scripts used in our experiments. We invite feedback and collaboration on the web at https://github.com/awslabs/sockeye.

References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[8] https://github.com/tensorflow/tensor2tensor

[9] All training scripts are available at https://github.com/awslabs/sockeye/tree/arxiv_1217.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Shujian Huang, Matthias Huck, Philipp Koehn, Qun Liu, Varvara Logacheva, Christof Monz, Matteo Negri, Matt Post, Raphael Rubino, Lucia Specia, and Marco Turchi. Findings of the 2017 Conference on Machine Translation (WMT17). In *WMT*, 2017.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. *CoRR*, abs/1612.08083, 2016.

Michael Denkowski and Graham Neubig. Stronger baselines for trustable results in neural machine translation. In *EMNLP Workshop on NMT*, 2017.

Jacob Devlin. Sharp models on dull hardware: Fast and accurate neural machine translation decoding on the cpu. In *EMNLP*, 2017.

Christopher Dyer, Victor Chahuneau, and Noah Smith. A simple, fast, and effective reparameterization of IBM Model 2. In *NAACL*, 2013.

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *NIPS*, 2016.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. *CoRR*, abs/1705.03122, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

Jindřich Helcl and Jindřich Libovický. Neural Monkey: An open-source tool for sequence learning. *The Prague Bulletin of Mathematical Linguistics*, (107):5–17, 2017.

Felix Hieber, Tobias Domhan, Michael Denkowski, David Vilar, Artem Sokolov, Ann Clifton, and Matt Post. Sockeye: A Toolkit for Neural Machine Translation. *ArXiv e-prints*, December 2017.

Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

Marcin Junczys-Dowmunt, Tomasz Dwojak, and Hieu Hoang. Is neural machine translation ready for deployment? A case study on 30 translation directions. In *IWSLT*, 2016.

Nal Kalchbrenner and Phil Blunsom. Recurrent continuous translation models. In *EMNLP*, 2013.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Guillaume. Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M. Rush. OpenNMT: Open-Source Toolkit for Neural Machine Translation. *CoRR*, abs/1701.02810, 2017.

Jayanth Koushik and Hiroaki Hayashi. Improving stochastic gradient descent with feedback. *CoRR*, abs/1611.01505, 2016.

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

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

Gabriel Pereyra, George Tucker, Jan Chorowski, Lukasz Kaiser, and Geoffrey E. Hinton. Regularizing neural networks by penalizing confident output distributions. *CoRR*, abs/1701.06548, 2017.

Ofir Press and Lior Wolf. Using the output embedding to improve language models. *CoRR*, abs/1608.05859, 2016.

Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *CoRR*, abs/1602.07868, 2016.

Holger Schwenk. Continuous space translation models for phrase-based statistical machine translation. In *COLING*, 2012.

Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. *CoRR*, abs/1603.05118, 2016.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. *CoRR*, abs/1511.06709, 2015.

Rico Sennrich, Alexandra Birch, Anna Currey, Ulrich Germann, Barry Haddow, Kenneth Heafield, Antonio Valerio Miceli Barone, and Philip Williams. The University of Edinburgh's neural MT systems for WMT17. In *WMT*, 2017a.

Rico Sennrich, Orhan Firat, Kyunghyun Cho, Alexandra Birch, Barry Haddow, Julian Hitschler, Marcin Junczys-Dowmunt, Samuel Läubli, Antonio Valerio Miceli Barone, Jozef Mokry, and Maria Nadejde. Nematus: a toolkit for neural machine translation. In *EACL Demo*, 2017b.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

Zhaopeng Tu, Yang Liu, Zhengdong Lu, Xiaohua Liu, and Hang Li. Context gates for neural machine translation. *TACL*, 5:87–99, 2017.

Ashish Vaswani, Liang Huang, and David Chiang. Smaller alignment models for better translations: Unsupervised word alignment with the l_0-norm. In *ACL*, 2012.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.