Sockeye 3: Fast Neural Machine Translation with PyTorch

Felix Hieber[*], **Michael Denkowski**[*], **Tobias Domhan**[*], **Barbara Darques Barros**, **Celina Dong Ye**, **Xing Niu**, **Cuong Hoang**, **Ke Tran**, **Benjamin Hsu**, **Maria Nadejde**, **Surafel Lakew**, **Prashant Mathur**, **Marcello Federico**, **Anna Currey**
Amazon

Abstract

Sockeye 3 is the latest version of the Sockeye toolkit for Neural Machine Translation (NMT). Now based on PyTorch, Sockeye 3 provides faster model implementations and more advanced features with a further streamlined codebase. This enables broader experimentation with faster iteration, efficient training of stronger and faster models, and the flexibility to move new ideas quickly from research to production. When running comparable models, Sockeye 3 is up to 126% faster than other PyTorch implementations on GPUs and up to 292% faster on CPUs. Sockeye 3 is open source software released under the Apache 2.0 license.

1 Introduction

Sockeye[1] provides a fast, reliable, and extensible codebase for Neural Machine Translation (NMT). As of version 3, Sockeye is based on PyTorch[2] (Paszke et al., 2019), offering researchers a familiar starting point for implementing their ideas and running experiments. Sockeye's distributed mixed-precision training and quantized inference also enable users to quickly build production-ready NMT systems. Inference benchmarks show that Sockeye is up to 126% faster than other PyTorch implementations on GPUs and up to 292% faster on CPUs. Sockeye supports a range of advanced NMT features including source and target factors, source and target prefixes, lexical shortlists, and fast hybrid decoders. Sockeye powers Amazon Translate[3] and has been used in numerous scientific publications.[4] Sockeye is developed as open source software on GitHub, where community contributions are welcome.

In the following sections, we describe Sockeye 3's scalable training (§2), optimized inference (§3), and advanced features (§4). We then share the results of a PyTorch NMT benchmark (§5) and case studies that use Sockeye features to implement formality and verbosity customization (§6). We conclude with a discussion of Sockeye's development philosophy (§7) and include a minimal usage example in an appendix (§A).

2 Training

Sockeye 3 implements optimized mixed precision training that scales to any number of GPUs and any size of training data.

2.1 Parallel Data Preparation

Sockeye provides an optional preprocessing step that splits training data into random shards, converts the shards to a binary format, and writes them to disk. During training, the shards are sequentially loaded and unloaded to enable training on arbitrarily large data with a fixed memory budget. Sockeye 3's data preparation step supports datasets of any size (subject to disk space) and runs in parallel on any number of CPUs. See Section A.2 for instructions on how to run data preparation.

2.2 Distributed Mixed Precision Training

By default, Sockeye training runs in FP32 on a single GPU. Activating mixed precision mode runs some or all of the model in FP16.[5] This yields a direct speedup from faster calculations and an indirect speedup from fitting larger batches into memory. Turning on distributed mode enables scaling to

[*]Corresponding authors:
{fhieber,mdenkows,domhant}@amazon.com

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

[2] https://pytorch.org/

[3] https://aws.amazon.com/translate/

[4] https://github.com/awslabs/sockeye#research-with-sockeye

[5] PyTorch AMP runs a mix of FP16 and FP32 operations to balance speed and precision: https://pytorch.org/docs/stable/amp.html. Apex AMP (O2) runs the entire model in FP16 to maximize speed: https://nvidia.github.io/apex/amp.html.

GPUs	Precision	Tokens/Sec
1	FP32	8,451
1	FP16 & FP32	28,287
8	FP32	65,280
8	FP16 & FP32	218,688

Table 1: WMT14 En-De big transformer training benchmark on a p3.16xlarge EC2 instance using the large batch recipe described by Ott et al. (2018).

any number of GPUs by launching separate training processes that use PyTorch's distributed data parallelism[6] to synchronize updates. In all cases, Sockeye traces the full encoder-decoder model with PyTorch's optimizing JIT compiler.[7] Shown in Table 1, activating mixed precision yields over 3X training throughput. Scaling to 8 local GPUs yields 7.7X throughput, demonstrating 96.6% GPU efficiency.

3 Inference

Inference benefits from previous development for static computation graphs, avoiding dynamic shapes and data-dependent control flow as much as possible. As such, we are able to trace various components of the model with PyTorch's JIT compiler (encoder, decoder, and beam search).

3.1 Quantization

By default, Sockeye runs inference with FP32 model weights. Quantizing these weights to FP16 or INT8 can increase translation speed and reduce the model's memory footprint. Enabling FP16 quantization for GPUs casts the entire model to FP16 at runtime. Enabling INT8 quantization for CPUs activates PyTorch's dynamic quantization[8] that runs linear layers (feed-forward networks) in INT8 while keeping the rest of the model in FP32. Both quantization strategies typically have minimal impact on quality and are recommended for most translation scenarios.

3.2 Efficient Greedy Search

Work on high performance NMT reports that certain types of models produce adequate translations without beam search (Junczys-Dowmunt et al., 2018). For such cases, Sockeye provides a dedicated implementation of greedy search that does not have the computational overhead of maintaining hypotheses in a beam. Table 2 compares Sockeye's beam search with a beam size of 1 to the greedy implementation. Greedy search improves translation speed by 16% on GPUs and 6% on CPUs for a model that is already optimized for speed (quantized weights and lexical shortlists).

	Translation Speed (Sent/Sec) ↑	
	Baseline	Greedy
GPU FP16	11.9	13.1
+Lexical Shortlist	12.0	13.9
CPU FP32	2.6	2.7
+Quantize INT8	4.5	4.9
+Lexical Shortlist	7.2	7.6

Table 2: Benchmark comparing Sockeye's beam search with size 1 (baseline) to greedy search for a big transformer (WMT17 En-De) with batch size 1. GPU inference runs on a g4dn.xlarge EC2 instance and CPU inference runs on a c5.2xlarge EC2 instance.

4 Advanced Features

Sockeye 3 migrates Sockeye 2's advanced NMT features (Domhan et al., 2020) from MXNet (Chen et al., 2015) to PyTorch. Sockeye 3 also introduces new features that are exclusive to the PyTorch version.

4.1 Migrated Sockeye 2 Features

Source Factors (Sennrich and Haddow, 2016): Combine additional representations (factors) with source word embeddings before running the first encoder layer. Factors can encode any pre-computable token level information such as original case or BPE type. This approach enables combining the advantages of a smaller normalized vocabulary (more examples of each type in the training data) and a larger fine-grained vocabulary (distinguish between types with the same normalized form but different original forms).

Lexical Shortlists (Devlin, 2017): When translating an input sequence, limit the target vocabulary to the top k context free translations of each source token. This can significantly increase translation speed by reducing the size of the output softmax that runs at each decoding step. Sockeye provides tools for generating shortlists from the training data

[6] https://pytorch.org/docs/stable/notes/ddp.html

[7] https://pytorch.org/docs/stable/jit.html

[8] https://pytorch.org/tutorials/recipes/recipes/dynamic_quantization.html

		big 6:6 transformer		big 20:2 transformer+SSRU	
		newstest	newstest-UP	newstest	newstest-UP
En-De	baseline	35.64	25.94	34.48	24.97
	+SF	35.52	28.85	34.62	27.37
	+SF+TF	35.18	**33.47**	35.12	**32.82**
Ru-En	baseline	32.99	24.47	33.53	25.99
	+SF	32.82	25.87	33.16	26.72
	+SF+TF	33.18	**31.39**	33.60	**31.49**

Table 3: BLEU scores of different models on newstest and its all-uppercased version (newstest-UP). Using target case factors (+SF+TF) achieves significantly higher BLEU than using source case factors alone (+SF) and the baseline for translating all-uppercased inputs. The training data is augmented with 1% all-uppercased pairs.

with `fast_align`[9] (Dyer et al., 2013) and uses a default value of $k = 200$.

4.2 SSRU Decoder

Sockeye supports replacing self-attention layers in the decoder with Simpler Simple Recurrent Units (SSRUs), which are shown to substantially improve translation throughput (Kim et al., 2019). An SSRU simplifies the LSTM cell by removing the reset gate and replacing the $\tanh$ non-linearity with ReLU:

$$\mathbf{f}_t = \sigma(\mathbf{W}_t\mathbf{x}_t + \mathbf{b}_f)$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + (1 - \mathbf{f}_t) \odot \mathbf{W}\mathbf{x}_t$$
$$\mathbf{h}_t = \mathrm{ReLU}(\mathbf{c}_t)$$

Only the cell state $\mathbf{c}_t$ requires sequential computations while other parts of the SSRU can be computed in parallel.

4.3 Target Factors

Factored models have been used to enrich phrase-based MT and NMT with linguistic features (Koehn and Hoang, 2007; García-Martínez et al., 2016). They reduce the output space by decomposing surface words y on different dimensions, such as lemma and morphological tags, and maximize $\mathrm{P}(y^t|y^{<t}\mathbf{x}) = \prod_{i=1}^{n} \mathrm{P}(f_i^t|y^{<t}, \mathbf{x})$.

When target factors are enabled, Sockeye 3 predicts target words (f_1) and attributes ($f_{2...n}$) with independent output layers, and the embeddings of the word and attributes are combined for the next decoder step. It incorporates the dependency between words and attributes by time-shifting attributes so that attributes at time t are actually predicted at time $t + 1$.

Following Niu et al. (2021), we test the effectiveness of using target case factors in translating all-uppercased inputs. We use the same train/dev/test data processing procedures as in other sections, except we (1) uppercase 1% training pairs and add them back to the training; (2) truecase data and deduct case factors as detailed in Niu et al. (2021), and (3) additionally evaluate on all-uppercased newstest sets.

Results in Table 3 show that, with sub-optimal data augmentation, using target case factors (+SF+TF) achieves significantly higher BLEU scores than using source case factors alone (+SF) and the baseline for translating all-uppercased inputs.

4.4 Fine-Tuning with Parameter Freezing

When fine-tuning models, freezing some or most of the parameters can increase training throughput, avoid overfitting on small in-domain data, and yield compact parameter sets for multitask or multilingual systems (Wuebker et al., 2018; Fan et al., 2021). Sockeye supports freezing any model parameter by name as well as presets for freezing everything except a specific part of the model (decoder, output layer, embeddings, etc.). When updating only the decoder, Sockeye turns off autograd for the encoder and skips its backward pass. This yields faster training updates and lowers memory usage, which enables larger batch sizes.

4.5 Source and Target Prefixes

Adding artificial source and target tokens has become a staple technique for NMT with applications ranging from multilingual models (Johnson et al., 2017) to output length customization (Lakew et al., 2022). While these tokens can be added to training data with simple pre-processing scripts, adding them during inference requires extended support in the NMT toolkit. Sockeye 3 enables users to specify arbitrary prefixes (sequences of tokens) on both the source and target sides for any input. Source

[9] https://github.com/clab/fast_align

prefixes are automatically added to the beginning of each input. When inputs are split into multiple chunks,[10] the source prefix is included at the beginning of each chunk. When a target prefix is specified, Sockeye 3 forces the decoder to generate the N prefix tokens as the first N decoder steps before continuing the translation normally. Because target prefixes have diverse use cases, Sockeye 3 allows users to choose whether to apply prefixes when translating all input chunks and whether to strip prefixes out of the translation output. For instance, multilingual NMT requires special tokens to be added to each chunk to identify the output language, but removes these artificial tokens from the final translation. By contrast, continuing partial translations requires that the prefix is only added to the first chunk and includes that prefix as part of the translation.

As an example of leveraging special tokens, let us consider a multilingual NMT model where the output language is specified on the source side. Using this model to translate into German requires adding the token `<2DE>` to the beginning of each input. Such source prefixes can be specified using Sockeye's JSON input format:

```
{"text": "The boy ate the waff@@
le .", "source_prefix": "<2DE>"}
```

This adds `<2DE>` to the beginning of each source chunk. If the model uses special target tokens to determine output language, a target prefix can be specified:

```
{"text": "The boy ate the waff@@
le .", "target_prefix": "<2DE>"}
```

This forces the decoder to generate `<2DE>` as its first target token. Finally, Sockeye 3 supports adding source and target prefix factors. For example:

```
{"text": "The boy ate the waff@@
le .", "target_prefix": "<2DE>",
"target_prefix_factors": ["O O B"
]}
```

Here `<2DE>` is force-decoded as the first target token and aligns with target factor `O`. The next two target tokens after `<2DE>` are assigned target factors `O` and `B`.

[10] During Sockeye inference, inputs that exceed the maximum sequence length set during training are split into smaller "chunks" that are translated independently.

4.6 Neural Vocabulary Selection

Instead of selecting the target vocabulary out of context as in lexical shortlisting (§4), Neural Vocabulary Selection (NVS) (Domhan et al., 2022) uses the encoder's hidden representation to predict the set of target words and is learned jointly with the translation model. Similarly, it results in lower translation latency via reduced computation per decoder step. The advantage of NVS lies in its simplicity, as no external alignment model is required and predictions are made in context, resulting in a higher recall of target words for given target vocabulary size.

5 Benchmark

We conduct a reproducible benchmark of PyTorch-based neural machine translation toolkits that includes Sockeye, Fairseq[11] (Ott et al., 2019), and OpenNMT[12] (Klein et al., 2017). For each toolkit, we train a standard big transformer model and run inference on GPUs and CPUs. The scripts used to conduct this benchmark are publicly available.[13]

5.1 Training

We select two translation tasks for which preprocessed data sets are available: WMT17 English-German (5.9M sentences) and Russian-English (25M sentences).[14] We further process the data by applying byte-pair encoding[15] (Sennrich et al., 2016) with 32K operations and filtering out sentences longer than 95 tokens. We use each toolkit to train a big transformer model (Vaswani et al., 2017) on 8 local GPUs (p3.16xlarge EC2 instance) using the large batch recipe described by Ott et al. (2018). Models are trained for either 25K updates (En-De) or 70K updates (Ru-En) with checkpoints every 500 updates. The 8 best checkpoints are averaged to produce the final model weights.

We use the fastest known settings for each toolkit that do not change the model architecture or training recipe. This includes enabling NVIDIA's Apex[16] extensions for PyTorch and running the entire model in FP16. Shown in Table 4, Sockeye and Fairseq are fastest, training models with

[11] https://github.com/pytorch/fairseq
[12] https://github.com/OpenNMT/OpenNMT-py
[13] https://github.com/awslabs/sockeye/tree/arxiv_sockeye3/arxiv
[14] https://data.statmt.org/wmt17/translation-task/preprocessed
[15] https://github.com/rsennrich/subword-nmt
[16] https://github.com/NVIDIA/apex

	WMT17 En-De		WMT17 Ru-En	
	Training Time (Hours) ↓	BLEU ↑	Training Time (Hours) ↓	BLEU ↑
Sockeye	9.9	35.3	28.1	33.1
Fairseq	10.0	35.3	28.0	33.0
OpenNMT	13.7	35.2	39.4	32.2

Table 4: Big transformer training benchmark using 8 GPUs on a p3.16xlarge EC2 instance. Models are trained using the large batch recipe described by Ott et al. (2018) for either 25K (En-De) or 70K updates (Ru-En).

	Translation Speed (Sent/Sec) ↑		
	Sockeye	Fairseq	OpenNMT
GPU FP16 Batch 64	67.8	66.1	47.8
+Lexical Shortlist	76.0	–	–
GPU FP16 Batch 1	8.4	3.2	4.2
+Lexical Shortlist	9.5	–	–
CPU FP32 Batch 1	1.2	1.1	1.2
+Quantize INT8	2.4	–	–
+Lexical Shortlist	4.7	–	–

Table 5: Big transformer WMT17 En-De inference benchmark. GPU inference runs on a g4dn.xlarge EC2 instance and CPU inference runs on a c5.2xlarge EC2 instance. All reported values are averages over 3 runs. The listed techniques do not significantly impact translation quality; BLEU scores for all settings are within 0.2 of the FP32 baseline.

	WMT17 En-De		WMT17 Ru-En	
	Training Time (Hours) ↓	BLEU ↑	Training Time (Hours) ↓	BLEU ↑
Big 6:6	9.9	35.3	28.1	33.1
Big 20:2	14.7	34.7	41.2	33.5
Big 20:2 SSRU	15.6	34.9	44.2	33.0

Table 6: Sockeye model architecture training benchmark using 8 GPUs on a p3.16xlarge EC2 instance. Models are trained using the large batch recipe described by Ott et al. (2018) for either 25K updates (En-De) or 70K updates (Ru-En). Model checkpoints are saved every 500 updates and the 8 best checkpoints are averaged.

	Translation Speed (Sent/Sec) ↑		
	Big 6:6	Big 20:2	Big 20:2 SSRU
GPU FP16 Batch 64	73.8	116.3	142.8
GPU FP16 Batch 1	9.9	17.4	18.5
CPU INT8 Batch 1	4.5	7.8	9.5

Table 7: Sockeye model architecture WMT17 En-De inference benchmark. GPU inference runs on a g4dn.xlarge EC2 instance and CPU inference runs on a c5.2xlarge EC2 instance. All configurations use lexical shortlists. All reported values are averages over 3 runs.

comparable BLEU scores in comparable time.

5.2 Inference

We benchmark inference on GPUs (g4dn.xlarge EC2 instance) and CPUs (c5.2xlarge EC2 instance with 4 physical cores). Shown in Table 5, Sockeye matches or outperforms other toolkits on GPUs and CPUs with and without batching. When activating NMT optimizations that are only natively supported by Sockeye (lexical shortlists and CPU INT8 quantization[17]), Sockeye is fastest across the board: +15% for batched GPU inference, +126% for non-batched GPU inference, and +292% for CPU inference.

5.3 Alternate Model Architectures

Domhan et al. (2020) report that transformer models with deep encoders and shallow decoders (20:2) can translate significantly faster than standard models (6:6) with similar quality (±1 BLEU). The speedup can be attributed to better parallelization in the encoder (sequence-level operations versus per-step operations) and fewer calculations per encoder layer (no encoder-decoder cross-attention and a single forward pass versus beam search).

We benchmark three versions of Sockeye's transformer: (1) the standard big 6:6 model from Section 5.1, (2) a big 20:2 model, and (3) a big 20:2 model that replaces decoder self-attention with SSRUs as described in Section 4.2. Shown in Tables 6 and 7, moving from a 6:6 model to a 20:2 model yields up to a 76% inference speedup and moving to SSRUs yields up to a 23% additional speedup (87%-111% faster than the baseline). These models do take longer to train. The 20:2 transformers have substantially more parameters (46 sub-layers versus 30) and SSRUs do not parallelize as well as self-attention during training. These trade-offs make models with deep encoders, shallow decoders, and SSRUs a good match for tasks where decoding time and costs dominate. This includes experiments that translate large amounts of data (e.g., back-translation) and applications where NMT models are deployed for large volume translation.

6 Case Studies

6.1 Formality Control

We present a case study on using Sockeye 3 transformer models with deep encoders and shallow SSRU decoders (20:2) introduced in Sections 4.2 and 5.3, and the source prefix feature introduced in Section 4.5. We train unconstrained baseline and formality controlled models for 6 language pairs for the 2022 IWSLT shared task on Formality Control for Spoken Language Translation.[18] The baseline models and fine-tuning instructions are publicly available.[19]

The English-German and English-Spanish models were trained on 20M pairs sampled from ParaCrawl v9 (Bañón et al., 2020), using WMT newstest for development. The English-Japanese model was trained on all 10M pairs from JParaCrawl v2 (Morishita et al., 2020) using the IWSLT17 development set. The English-Hindi model was trained on all 15M pairs from CCMatrix (Schwenk et al., 2021), using the WMT newsdev2014 for development and newstest2014 for testing.

For evaluating generic quality, we used the WMT newstests[20] as well as the MuST-C test sets (Di Gangi et al., 2019). To train and evaluate formality-controlled models we use the CoCoA-MT dataset and benchmark (Nădejde et al., 2022). We replicate the experiments in Nădejde et al. using Sockeye 3 models: we fine-tune the generic baseline MT model on labeled contrastive translation pairs augmented by an equal number of randomly sampled unlabeled generic training data. The contrastive translation pairs are labeled using a special source prefix that specifies the formality level of the target:

```
src: <FORMAL> 'Are you tired?'
trg: 'Sind Sie muede?'
src: <INFORMAL> 'Are you tired?'
trg: 'Bist du muede?'
```

At inference time, we use the source prefix to control the formality level in the output. We report evaluation results in Table 8 showing formality-controlled models have high targeted accuracy while preserving generic quality.

6.2 Isometric MT

We present another case study on Isometric MT where the task is to generate translations similar in length to the source text. In this setup, we experiment with the verbosity control (VC) work of

[17] At the time of writing, activating OpenNMT's INT8 mode does not appear to have any impact.

[18] https://iwslt.org/2022/formality

[19] https://github.com/amazon-research/contrastive-controlled-mt/tree/main/IWSLT2022/models

[20] We used newstest 2020 for German, 2014 for Spanish, 2014 for Hindi, 2020 for Japanese

		M-ACC - CoCoA-MT test			BLEU	
Lang.	System	F	I	Avg.	WMT	TED
EN-DE	generic	-	-	-	42.1	32.7
	controlled	97.8	45.0	71.4	41.4	32.1
EN-ES	generic	-	-	-	35.1	36.7
	controlled	89.1	47.8	68.4	35.0	36.9
EN-HI	generic	-	-	-	10.0	-
	controlled	96.3	36.7	66.5	9.9	-
EN-JA	generic	-	-	-	21.7	14.3
	controlled	68.8	83.2	76.0	22.2	14.3

Table 8: Accuracy of generic baseline and formality-controlled models on the CoCoA-MT test set. The TED test sets are MuST-C for EN-DE,ES and IWSLT for EN-JA. For controlled models, M-Acc (F)/(I) scores are computed using formal/informal translations respectively, resulting in performance upper bounds of 100%.

Lang. Pair	Test set	System	BERTScore	LC	BERTScore×LC
En-De	MuST-C	Baseline	0.837	41.3	34.6
		VC	0.834	56.6	47.2
	IMT	Baseline	0.757	51.5	39.0
		VC	0.757	56.5	42.8
		VC+Rank	0.743	63.5	47.2
En-Fr	MuST-C	Baseline	0.867	38.7	33.6
		VC	0.860	53.6	46.1
	IMT	Baseline	0.778	39.5	30.7
		VC	0.778	58.0	45.1
		VC+Rank	0.772	65.5	50.6
En-Es	MuST-C	Baseline	0.846	60.0	50.8
		VC	0.845	66.7	56.4
	IMT	Baseline	0.802	59.0	47.3
		VC	0.799	62.5	50.0
		VC+Rank	0.789	64.0	50.5

Table 9: Results comparing a standard NMT model (Baseline), NMT with output verbosity control (VC), and VC with N-best re-ranking (VC+Rank) on the Ted Talks MuST-C test set released for the isometric MT (IMT) shared task. Models are evaluated using BERTScore and length compliance within ±10% (LC), and the final system ranking metric (BERTScore×LC).

Lakew et al. (2019), specifically the length token approach using Sockeye's source prefix implementation (Section 4.5). We train and evaluate models using data from the constrained setting in the 2022 IWSLT shared task on Isometric Spoken Language Translation.[21] Evaluation is done on MuST-C v1.2 (Cattoni et al., 2021) and a blind set released as part of the shared task.[22] All models are evaluated on three language pairs: English-German, English-French, and English-Spanish using BERTScore (Zhang et al., 2020). There is also a length compliance (LC) metric (Lakew et al., 2022) which measures whether the translation is within $\pm 10\%$ of the source length and a final system ranking metric that combines BERTScore and LC. For preprocessing, we leverage SentencePiece (Kudo and Richardson, 2018) with 16.5K operations. Models are trained with the transformer base (6:6) architecture on 8 GPUs (p3.16xlarge instances). At training time, we apply `<short>`, `<normal>`, and `<long>` prefixes to the source side of the parallel training data based on the length compliance of the target side. During inference, we add the `<normal>` prefix to generate translations that are similar in length to source. Following (Lakew et al., 2019), we also run an ablation study for the blind set where we re-rank the N-best list to find the best translation in terms of translation quality and length. We report results in Table 9 that show improvements when adding verbosity control (VC) to baseline models and further improvements when applying N-best re-ranking (VC+Rank).

7 Development

Sockeye is developed as open source software under the Apache 2.0 license and hosted on GitHub. All contributions are publicly reviewed using GitHub's pull request system. Sockeye is written in PEP 8 compatible Python 3 code. Functions are documented with Sphinx-style docstrings and include type hints for static code analysis. Sockeye

[21] https://iwslt.org/2022/isometric

[22] https://github.com/amazon-research/isometric-slt/tree/main/dataset

includes an extensive suite of unit, integration, and system tests covering the toolkit's core functionality and advanced features. New code is required to pass all tests (and add new tests to cover new functionality) plus type checking and linting in order to be merged. Sockeye 3 retires some older features such as lexical constraints. We welcome pull requests from community members interested in porting these features from Sockeye 2.

8 Acknowledgements

We would like to thank Vincent Nguyen for helping us configure OpenNMT for an accurate benchmark.

References

Marta Bañón, Pinzhen Chen, Barry Haddow, Kenneth Heafield, Hieu Hoang, Miquel Esplà-Gomis, Mikel L. Forcada, Amir Kamran, Faheem Kirefu, Philipp Koehn, Sergio Ortiz Rojas, Leopoldo Pla Sempere, Gema Ramírez-Sánchez, Elsa Sarrías, Marek Strelec, Brian Thompson, William Waites, Dion Wiggins, and Jaume Zaragoza. 2020. ParaCrawl: Web-scale acquisition of parallel corpora. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4555–4567, Online. Association for Computational Linguistics.

Roldano Cattoni, Mattia Antonino Di Gangi, Luisa Bentivogli, Matteo Negri, and Marco Turchi. 2021. Must-c: A multilingual corpus for end-to-end speech translation. *Computer Speech & Language*, 66:101155.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*.

Jacob Devlin. 2017. Sharp models on dull hardware: Fast and accurate neural machine translation decoding on the CPU. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2820–2825, Copenhagen, Denmark. Association for Computational Linguistics.

Mattia A. Di Gangi, Roldano Cattoni, Luisa Bentivogli, Matteo Negri, and Marco Turchi. 2019. MuST-C: a Multilingual Speech Translation Corpus. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2012–2017, Minneapolis, Minnesota. Association for Computational Linguistics.

Tobias Domhan, Michael Denkowski, David Vilar, Xing Niu, Felix Hieber, and Kenneth Heafield. 2020. The sockeye 2 neural machine translation toolkit at AMTA 2020. In *Proceedings of the 14th Conference of the Association for Machine Translation in the Americas (Volume 1: Research Track)*, pages 110–115, Virtual. Association for Machine Translation in the Americas.

Tobias Domhan, Eva Hasler, Ke Tran, Jonay Trenous, Bill Byrne, and Felix Hieber. 2022. The devil is in the details:on the pitfalls of vocabulary selection in neural machine translation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Seattle, USA. Association for Computational Linguistics.

Chris Dyer, Victor Chahuneau, and Noah A. Smith. 2013. A simple, fast, and effective reparameterization of IBM model 2. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 644–648, Atlanta, Georgia. Association for Computational Linguistics.

Angela Fan, Shruti Bhosale, Holger Schwenk, Zhiyi Ma, Ahmed El-Kishky, Siddharth Goyal, Mandeep Baines, Onur Celebi, Guillaume Wenzek, Vishrav Chaudhary, Naman Goyal, Tom Birch, Vitaliy Liptchinsky, Sergey Edunov, Michael Auli, and Armand Joulin. 2021. Beyond english-centric multilingual machine translation. *Journal of Machine Learning Research*, 22(107):1–48.

Mercedes García-Martínez, Loïc Barrault, and Fethi Bougares. 2016. Factored neural machine translation architectures. In *Proceedings of the 13th International Workshop on Spoken Language Translation*, Seattle, US.

Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2017. Google's multilingual neural machine translation system: Enabling zero-shot translation. *Transactions of the Association for Computational Linguistics*, 5:339–351.

Marcin Junczys-Dowmunt, Kenneth Heafield, Hieu Hoang, Roman Grundkiewicz, and Anthony Aue. 2018. Marian: Cost-effective high-quality neural machine translation in C++. In *Proceedings of the 2nd Workshop on Neural Machine Translation and Generation*, pages 129–135, Melbourne, Australia. Association for Computational Linguistics.

Young Jin Kim, Marcin Junczys-Dowmunt, Hany Hassan, Alham Fikri Aji, Kenneth Heafield, Roman Grundkiewicz, and Nikolay Bogoychev. 2019. From research to production and back: Ludicrously fast neural machine translation. In *Proceedings of the 3rd Workshop on Neural Generation and Translation*, pages 280–288, Hong Kong. Association for Computational Linguistics.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. 2017. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada. Association for Computational Linguistics.

Philipp Koehn and Hieu Hoang. 2007. Factored translation models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 868–876, Prague, Czech Republic. Association for Computational Linguistics.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

Surafel Melaku Lakew, Mattia Di Gangi, and Marcello Federico. 2019. Controlling the output length of neural machine translation. In *Proceedings of the 16th International Conference on Spoken Language Translation*, Hong Kong. Association for Computational Linguistics.

Surafel Melaku Lakew, Yogesh Virkar, Prashant Mathur, and Marcello Federico. 2022. Isometric MT: neural machine translation for automatic dubbing. In *Proceedings of the 2022 IEEE International Conference on Acoustics, Speech and Signal Processing*.

Makoto Morishita, Jun Suzuki, and Masaaki Nagata. 2020. JParaCrawl: A large scale web-based English-Japanese parallel corpus. In *Proceedings of The 12th Language Resources and Evaluation Conference*, pages 3603–3609, Marseille, France. European Language Resources Association.

Xing Niu, Georgiana Dinu, Prashant Mathur, and Anna Currey. 2021. Faithful target attribute prediction in neural machine translation. *CoRR*, abs/2109.12105.

Maria Nădejde, Anna Currey, Benjamin Hsu, Xing Niu, Marcello Federico, and Georgiana Dinu. 2022. CoCoA-MT: A dataset and benchmark for Contrastive Controlled MT with application to formality. In *Findings of the Association for Computational Linguistics: NAACL 2022*, Seattle, USA. Association for Computational Linguistics.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*.

Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. 2018. Scaling neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 1–9, Brussels, Belgium. Association for Computational Linguistics.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Holger Schwenk, Guillaume Wenzek, Sergey Edunov, Edouard Grave, Armand Joulin, and Angela Fan. 2021. CCMatrix: Mining billions of high-quality parallel sentences on the web. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 6490–6500, Online. Association for Computational Linguistics.

Rico Sennrich and Barry Haddow. 2016. Linguistic input features improve neural machine translation. In *Proceedings of the First Conference on Machine Translation: Volume 1, Research Papers*, pages 83–91, Berlin, Germany. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

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 *Advances in neural information processing systems*, pages 5998–6008.

Joern Wuebker, Patrick Simianer, and John DeNero. 2018. Compact personalized models for neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 881–886, Brussels, Belgium. Association for Computational Linguistics.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*.

A Installation and Usage

A.1 Installation

The easiest way to install Sockeye is via pip:

```
> pip3 install sockeye
```

Once Sockeye is installed, you can use the included command-line tools to train models (`sockeye-train`), translate data (`sockeye-translate`), and more. If you plan to extend or modify the code, you can install Sockeye from source:

```
> git clone https://github.com/
awslabs/sockeye.git
> cd sockeye
> pip3 install --editable ./
```

Using the `editable` flag means that changes to the code will apply directly without needing to reinstall the package.

A.2 Sample Usage

Training a Sockeye model requires parallel (source and target) training and validation data. You can use raw training data directly, though we recommend using `sockeye-prepare-data` to prepare the data ahead of time. This reduces memory consumption and data loading time during training:

```
> sockeye-prepare-data \
  -s [source training data] \
  -t [target training data] \
  -o [output directory]
```

To train a model from scratch, run `sockeye-train` with the prepared training data directory, validation source and target data files, model output directory, and at least one stopping criteria such as number of training steps:

```
> sockeye-train \
  -d [prepared training data] \
  -vs [source validation data] \
  -vt [target validation data] \
  -o [output directory] \
  --max-updates [training steps]
```

To fine-tune an existing model on new data (e.g., for domain adaptation), run `sockeye-train` with the new data and specify a checkpoint from the existing model with the `--params` argument.

Once you have trained a Sockeye model, you can use it to translate inputs by running:

```
> sockeye-translate \
 -m [model directory]
```

This section covers a minimal example of using Sockeye's CLI tools. For a step-by-step tutorial on training a standard transformer model on any size of data, see the WMT 2014 English-German example[23] on GitHub.

[23] https://github.com/awslabs/sockeye/blob/main/docs/tutorials/wmt_large.md