SOTASTREAM: A Streaming Approach to Machine Translation Training

Matt Post **Thamme Gowda** **Roman Grundkiewicz**
Huda Khayrallah **Rohit Jain** **Marcin Junczys-Dowmunt**

Microsoft

Abstract

Many machine translation toolkits make use of a data preparation step wherein raw data is transformed into a tensor format that can be used directly by the trainer. This preparation step is increasingly at odds with modern research and development practices because it produces a static, unchangeable version of the training data, making common training-time needs difficult (e.g., subword sampling), time-consuming (preprocessing with large data can take days), expensive (e.g., disk space), and cumbersome (managing experiment combinatorics). We propose an alternative approach that separates the *generation* of data from the *consumption* of that data. In this approach, there is no separate pre-processing step; data generation produces an infinite stream of permutations of the raw training data, which the trainer tensorizes and batches as it is consumed. Additionally, this data stream can be manipulated by a set of user-definable operators that provide on-the-fly modifications, such as data normalization, augmentation or filtering. We release an open-source toolkit, SOTASTREAM, that implements this approach: https://github.com/marian-nmt/sotastream. We show that it cuts training time, adds flexibility, reduces experiment management complexity, and reduces disk space, all without affecting the accuracy of the trained models.

1 Introduction

A cumbersome component of training machine translation systems is working with large amounts of data. Modern high-resource parallel datasets are often on the order of hundreds of millions of parallel sentences, and backtranslation easily doubles that (Kocmi et al., 2022, Appendix A). Because this data is too large to fit into main memory, toolkits such as FAIRSEQ (Ott et al., 2019) and SOCKEYE (Hieber et al., 2022) make use of a preprocessing step, which transforms the training data from its raw state into a static sequence of tensors.

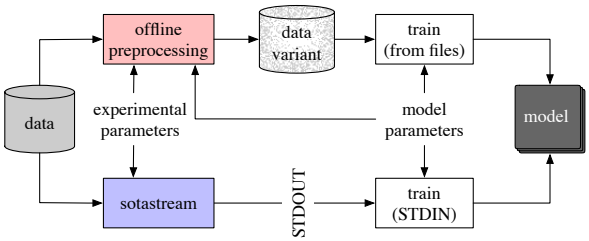

Figure 1: The SOTASTREAM approach separates data generation from consumption. Whereas offline tensorization requires model-specific parameters such as the vocabulary, which ties processed data to a particular training, SOTASTREAM produces data on the fly, avoiding time-consuming production and space-wasting storage of preprocessed data.

These tensors can then be read in via an index and memory-mapped shards, allowing for quick assembly into batches at training time.

While this offline preprocessing prevents data loading from becoming a bottleneck in training, it creates a number of other problems:

- *it breaks an abstraction*: the tensorized data is tied to specific modeling decisions, such as the vocabulary;
- *it is cumbersome*: the tensorized data cannot be easily changed, and even minor variations of the data must be processed separately and then managed;
- *it is time-consuming*: pre-processing can take considerable time and must be completed before training can start; and
- *it is wasteful*: each data variant replicates the original's disk space.

These problems exist for construction of any model, but are exacerbated in research settings, which often explore variations of the training data.

We describe an alternative that factors *generation* of data from the *consumption* of that data by

the training toolkit. This view presents the training data as an infinite stream of permutations of the raw training samples. This stream is then consumed by the training toolkit, which tensorizes it on the fly, consuming data into a buffer from which it can assemble batches. This framework eliminates all the problems above: variants of the data are independent of any model; arbitrary manipulations can be applied on the fly; preprocessing time is amortized over training, which can start as soon as the first batch can be constructed;and no extra disk space or management is required.

We release an open-source implementation of the proposed data generation framework called SOTASTREAM[1]. SOTASTREAM is written in Python and uses Infinibatch[2] to provide a stream of data over permutations of data sources. It additionally provides an easily-extendable set of *mixers*, *augmentors*, and *filters* that allow data to be probabilistically manipulated on the fly. A particular configuration of manipulators is provided by the user in the form of a dynamically-loadable *pipeline*, which defines a parameterizable recipe that can be used for training. SOTASTREAM uses multiprocessing to reach high throughput levels that prevent starvation of the training toolkit. And finally, it employs a standard UNIX API, writing data to STDOUT.

After presenting this framework (§ 2), we conduct a quality comparison to demonstrate that it does not reduce model quality (§ 4). We then investigate stream bandwidth under various pipelines as well as necessary toolkit consumption needs (§ 5). We conclude by demonstrating a number of use cases (§ 6).

2 Training from data streams

The core idea underlying SOTASTREAM is to cleanly separate data generation from consumption of that data during training. The *data generator* is responsible for producing training samples, and the *trainer* consumes them. This factorization allows us to separate properties of the data (such as their sources, mixing ratios, and augmentations) from properties of training and the model (such as tensor format, batch size, and so on).

The current approach relies on standard UNIX I/O pipes as an interface between these two pieces. However, SOTASTREAM could also be used to generate data for offline uses, or modified to allow consumption through some other API, such as a library call that returns a generator.

2.1 Data generation

SOTASTREAM is a data generator. At a high level, it works by defining a *pipeline*. This pipeline reads from a set of zero or more input data sources, applies any augmentations, and produces a single output stream.

Pipelines Pipelines are implemented by inheriting from the base `Pipeline` class. The class implementation is responsible for defining the input data sources, reading from them, applying augmentations, and returning a single output stream. These are depicted in Figure 2, a simplified presentation that elides other support features, such as providing the mixing weights for the input data sources.

The pipeline has three basic components:

1. Build a stream for each input data source;
2. apply a sequence of augmentors; and
3. merge the streams to a single output stream.

Data sources SOTASTREAM uses Infinibatch to return a generator over a permutation of the samples in a data source. Each `DataSource` object receives two key arguments: a file path to the data source on disk, d, and a processor function, f, to read it. This can be seen in Figure 2 in the call to `create_data_stream`(d, f).

The data is received as a path to a directory of compressed TSV file shards. Infinibatch requires that data be presented in this way.[3] For each data epoch, Infinibatch produces a permutation of these shards. The shards are then passed, in turn, to the function f, which is responsible for opening, reading, and processing the shard. It is important to note that Infinibatch provides an *infinite stream* of data; that is, it will iterate indefinitely over its input data, subject to the constraint that no shard (within a data source) will be seen $n + 1$ times until all shards have been seen n times. See the Multiprocessing section (§ 2.3) below for important caveats related to multiprocessing and MPI training).

Augmentations The second argument to `create_data_stream` is a generator function, f,

[1] https://github.com/marian-nmt/sotastream
[2] https://github.com/microsoft/infinibatch

[3] SOTASTREAM can also receive a path to a single compressed TSV file, in which case it splits the file into shards under a temporary directory. The default shard size is `1e6` lines. The results of this automatic sharding are cached using an MD5 checksum.

```
@pipeline("robust-case")
class RobustCasePipeline(Pipeline):
  def __init__(self, pa_dir: str, bt_dir: str, **kwargs):
    super().__init__(**kwargs)
    pa_stream = self.create_data_stream(pa_dir, processor=Augment)
    bt_stream = self.create_data_stream(bt_dir,
      processor=partial(Augment, tag="[BT]"))  # tag the BT data
    self.stream = Mixer([pa_stream, bt_stream], self.mix_weights)
  # definitions of other class methods go here ...

def LowerCase(stream: Generator[Line]) -> Generator[Line]:
  for line in stream:
    line[0] = line[0].lower()  # lowercase the source side
    yield line

def TitleCase(stream: Generator[Line]) -> Generator[Line]:
  for line in stream:
    line[0], line[1] = line[0].title(), line[1].title()  # titlecase both sides
    yield line

def TagData(stream: Generator[Line], tag: str) -> Generator[Line]}:
  for line in stream:
    line[0] = f"{tag} {line}"  # add a target language tag to the source
    yield line

def Augment(path: str, tag: str = None) -> Generator[Line]:
  stream = UTF8File(path)   # open the path to the shard

  stream = Mixer(  # randomly mix casing variants
    [ stream, LowerCase(stream), TitleCase(stream) ],
    [ 0.95,   0.04,              0.01 ],
  )

  if tag is not None:
    stream = TagData(stream, tag)
  return stream
```

Figure 2: A simplified pipeline. Streams are built by composing generator functions over input data sources (here, parallel and backtranslated data). This example tags the backtranslated stream, then mixes it with the parallel stream using weights provided on the command line (defaulting to 1:1). It then applies random source-lowercasing (4%) and title-casing (1%).

```
class Line:
  def __init__(self, line: str):
    if line is not None:
      self.fields = line.split("\t")
    else:
      self.fields = []
```

Figure 3: The (simplified) `Line` object, a lightweight wrapper around a single row of tab-separated input data.

an Infinibatch primitive whose task is to open each shard and produce an output data stream. The output is in the form of `Line` objects (Figure 3), each of which is a class representation of the TSV input. By convention in machine translation, fields 0 and 1 are treated as *source* and *target* segments, respectively, but the code itself makes no such assumptions.

The function is not limited in just reading and returning the data. A key feature of SOTASTREAM is augmentations, which are arbitrary manipulations of a data stream that are easy to stack and accumulate. This is accomplished by composing generators. Figure 2 contains a number of examples in the `Augment` function. It first opens a stream on a path (passed from Infinibatch, containing a path to a sharded file name). It then applies lowercasing and title-casing to the input stream probabilistically, using a `Mixer` class to select among them with specified weights. Finally, it prepends a tag to the data, if requested by the caller.

Outputting the stream Finally, at the top level, the (augmented) streams from different data sources are merged into a single stream. This works in the same way as the above `Mixer` class example. One additional feature is that the `Pipeline` class provides the ability to set these top-level data

weights from the command line (`--mix-weights`).

2.2 Data consumption

The main requirements for the trainer are to consume data into a pool, apply subword processing, organize into batches, and run backpropagation against the training objective. Because these are done on the fly, rather than in preprocessing, special considerations must be implemented to ensure that this extra processing does not become a bottleneck for training.

In Section 5, we experiment with an implementation in the Marian toolkit (Junczys-Dowmunt et al., 2018). Marian makes use of multiple worker threads, which pre-fetch data from STDIN into an internal memory pool, where the data is tokenized and integerized. When the pool is filled, it is sorted and batched (according to run-time settings). In the meantime, prefetching continues into a second pool. As training proceeds, these two pools are used alternately for filling via prefetching and batch generation.

2.3 Multiprocessing

In order to sustain a sufficient throughput, SOTASTREAM makes use of multiprocessing. This can be increasingly important if the augmentations applied are expensive to compute. We quantify the effects of multiprocessing for generation under a handful of pipelines of varying complexity in Section 5.

Internally, this is accomplished with the `multiprocessing` library. We create separate subprocesses, each of which is provided with independent access to the data sources. The parent process maintains a pipe to each subprocess, and queries them in sequence, reading a fixed number of lines from each in turn, and passing them to the standard output.

An important issue is raised when working with subprocesses. If each subprocess were to return an independent permutation over the input data, merging subprocesses would not itself result in a permutation. To address this, each of n subprocesses is initalized with $\frac{1}{n}$ of the data shards, themselves assigned in round-robin order across the subprocesses. In this way, we guarantee a permutation in settings where the number of processes evenly divides the number of shards.

When working over MPI, no such coordination takes place. Each MPI instantiation will receive a different randomly-seeded shard permutation.

3 Experimental setup

Our experimental goal is to demonstrate that the many advantages of SOTASTREAM do not come at a cost in accuracy (§ 4) or speed (§ 5). We do this by comparing to a number of other data loading methods. In order to isolate the effects of changing the data loader, we conduct all of our experiments within the Marian toolkit. Marian does not support offline data preprocessing; instead, we compare a number of different streaming settings that cover best-case scenarios for data loading.

3.1 Streaming variations

We compare the following data-loading variations.

- *Full loading*. In this scenario, the trainer has direct memory access to the entire data source. For our experiments, Marian loads the complete datasets into main memory. There is some startup cost, after which all access to the data is immediate.

- *Sequential streaming*. In this approach, the training data is read sequentially, in a loop over the entire training set. Data is prefetched into a pool of a specified size, from which mini-batches are assembled. Since data is read sequentially, there is no randomization across data epochs. The pool size determines an upper bound on memory usage.

- *Randomized sequential streaming*. In this variant of sequential streaming, the lines in each data source are randomly permuted prior to being read, providing a corpus-level permutation on top of sequential streaming's pool-based reordering.

- SOTASTREAM. Our Infinibatch-based streaming approach.

For toolkits that support preprocessing, it is typical to construct an index, which organizes the presorted and tensorized data into memory-mappable shards. Marian does not have a preprocessing option, which means that we have no comparison to a setting where tensorization is done offline. We thus consider full-loading to be the closest equivalent, since preprocessing is in fact a stand-in for full loading. This can only possible affect speed comparisons (§ 5).

3.2 Model parameters

We conduct experiments in a large-data and small-data setting. Our large-data setting is English–German. We train on 297m lines of Paracrawl v9 (Bañón et al., 2020) from WMT22 (Kocmi et al., 2022). We use a 32k shared unigram subword model (Kudo, 2018) using SentencePiece (Kudo and Richardson, 2018), trained jointly over both sides. We train a standard base Transformer model (Vaswani et al., 2017) with 6/6 encoder/decoder layers, an embedding size of 1024, a feed-forward size of 4096, and 8 attention heads. The large model is trained for 20 virtual epochs. Since there are roughly 7.4 billion target-side tokens after tokenizing the data, this equates to roughly three passes over the data.

For the small-data setting, we train on Czech–Ukrainian, also from WMT22. This dataset has roughly 12m parallel lines. We use the same model and parameter settings, but train for only five virtual epochs, or roughly 30 data epochs, since the model converges by then.

3.3 Evaluation

We evaluate on the WMT21/en-de and WMT22/cs-uk test sets. We use a number of metrics to capture variation:

- **BLEU** (Papineni et al., 2002) and **chrF** (Popović, 2015), both computed with sacrebleu[4] (Post, 2018).
- **COMET20/22** (Rei et al., 2020), using model `wmt20-comet-da` (EN-DE) or `wmt22-comet-da` (CS-UK).

4 Quality Comparison

Table 1 contains metric results for both our high- and low-resource settings. For English–German, we observe rough equivalence across all training methods and metrics, which establishes SOTASTREAM as a viable data preparation tool. A similar pattern holds for Czech–Ukrainian, except for the odd outlier of the sequential streaming approach. This approach simply 'cat'ed the training data repeatedly until model convergence. This result is strongest for COMET and less pronounced for BLEU and chrF. We have no clear explanation for this; one guess is that in smaller data settings, with no filtering, curriculum effects may be more pronounced, and this is the only data generation approach with no randomization. Among approaches that permuted the data, SOTASTREAM is on par with the others. We therefore consider it to pass the quality benchmark.

5 Speed

Next we ask whether SOTASTREAM has a negative effect on speed. We examine speed in three settings: generation speed (§ 5.1), Marian's consumption speed (§ 5.2), and total runtime (§ 5.3).

5.1 Data generation

We first examine how fast SOTASTREAM can write data to STDOUT.

Our benchmark consists of a producer and a consumer connected by UNIX pipe. The producer varies among the tools we compare in our benchmark (described below), while the consumer is a lightweight script, whose sole purpose is to count records from STDIN and report the yield rate (the number of lines per second). All benchmarks are run one at a time, on the same machine,[5] with no other CPU- or I/O-intensive processes are competing for resources. We run each benchmark multiple times and report the average.

We compare the following generation tools:

- `zcat`: A wrapper to GNU gzip[6] that decompresses and outputs lines. This serves as the best case scenario, where the producer is implemented in an efficient way (e.g. C/C++) and has no time-consuming augmentations.
- `zcat.py`: Similar to `zcat`, but based on `gzip` API from Python's standard library.[7]
- `default` pipeline: SOTASTREAM' default, returning lines from a single data source (§ 2.1) with no augmentations.
- `case augmentor` pipeline: the pipeline from Figure 2. It mixes two data sources, applies case transformations, and prepends a "[BT]" tag to the backtranslated data.

We benchmark multiple worker subprocesses: $n \in \{1, 2, 4, 8, 16, 32\}$. The throughput measured is

[4] Version 2.3.1 with default settings.

[5] An Intel Xeon E5-2620 CPU with 32 cores, 660 GB of RAM, and running Ubuntu 20.04 LTS.

[6] https://git.savannah.gnu.org/cgit/gzip.git/tree/zcat.in

[7] https://docs.python.org/3/library/gzip.html

Model	English–German (newstest2021) COMET20	BLEU	chrF	Czech–Ukranian (wmttest2022) COMET22	BLEU	chrF
Best constrained	54.8	31.3	60.7	91.6	34.7	61.5
Full loading	55.9 ± 0.4	34.9 ± 0.1	62.0 ± 0.0	85.5 ± 0.2	27.9 ± 0.4	55.6 ± 0.2
Sequential streaming	56.1 ± 0.2	35.0 ± 0.2	62.1 ± 0.0	86.4 ± 0.1	28.7 ± 0.3	56.6 ± 0.2
Randomized streaming	55.8 ± 0.2	35.1 ± 0.0	62.2 ± 0.0	85.6 ± 0.1	27.8 ± 0.0	55.6 ± 0.2
SOTASTREAM	55.9 ± 0.1	34.9 ± 0.1	62.1 ± 0.1	85.7 ± 0.2	28.5 ± 0.4	56.2 ± 0.2

Table 1: Mean over three runs for our high- and low-resource scenarios. The best constrained system is WeChat-AI (Zeng et al., 2021) for EN-DE and AMU (Nowakowski et al., 2022) for CS-UK.

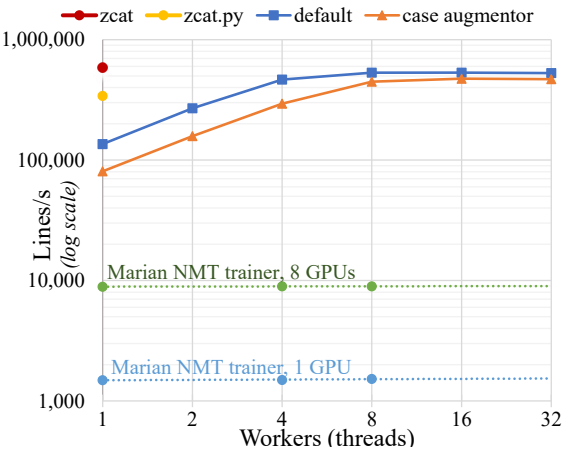

Figure 4: Generation and consumption rates with SOTASTREAM and Marian, respectively. SOTASTREAM is not a bottleneck, but is easily able to generate data and transmit it through the POSIX API to sustain training.

as lines/s and is given Figure 4. `zcat`, being the fastest, yields over 585k lines/s, and Python's alternative (`zcat.py`) yields 342k lines/s.[8] Our SOTASTREAM `default` with a single worker yields approximately 136k lines/s, which can be increased with more workers and plateaus after a certain rate (possibly due to a bottleneck in number of parallel reads supported by underlying storage device). As we add more augmentations and mixture processes, we observe a lower yield rate than no-augmentation baselines (expected). However, yield rate can be improved with more worker processes.

5.2 Consumption

We have shown the rate at which SOTASTREAM can generate data. In this section, we show the rate at which one particular NMT trainer (Marian) consumes training data. Training time and the consumption rate varies based the size of model being trained, and the number of GPUs used for training.

We train smaller Transformer models than used for Table 1, since smaller models train faster and therefore have higher data consumption needs. We use 6/6 encoder/decoder layers, 512-dimensional embeddings, and feedforward sublayers of size 2048. We report consumption rate for six settings: one vs. eight GPUs,[9] and using one, four, or eight prefetching worker threads. As shown in Figure 4, a trainer with single GPU consumes about 1523 lines/s, and with eight GPUs, the consumption rate increases to 8957 lines/s. Even in the best case scenario (smaller model, more GPUs, and more prefetcher threads), the consumption rates of training process are lower than SOTASTREAM production rate.

We recommend running multiple workers when augmentations are slow in order to maintain sufficient output rates. We do not experiment with them here, but in multi-node training settings coordinated with MPI, one (multiprocess) instance of SOTASTREAM should be run per node.

Model	Time (Hours)
Full loading	36.84 ±0.16
Sequential streaming	35.51 ±0.15
Randomized streaming	35.73 ±0.05
SOTASTREAM	35.86 ±0.27

Table 2: End-to-end training time.

5.3 Total time to run

Table 2 verifies that SOTASTREAM's amortized approach is neither slower nor faster than other approaches when total runtime is considered.

6 Example Use Cases

In this section we show example use cases how SOTASTREAM can be used to simply and easily modify data on the fly. This provides all the advantages of training for robustness without the cumbersome task of generating (and managing) data that has

[8] Measured on CPython v3.11; prior versions of CPython are found to be slower.

[9] NVIDIA Tesla V100s with 32GB.

been preprocessed in many different forms, which are combinatorial and impose high costs on the complexity of managing training runs.

6.1 Mixing multiple streams of data

Training machine translation models often requires combining different data sets in desired proportions in order to balance their size or quality or other properties. The example in Figure 2 demonstrates that combining original parallel data and back-translated data can be efficiently achieved in SOTASTREAM by mixing multiple data streams with specific data weighting. The weights for each data stream can be then specified using the command-line options:

```
sotastream robust-case \
  parallel.tsv.gz backtrans.tsv.gz \
  --mix-weights 1 1
```

The weights are normalized and used as probabilities with the `Mixer` augmentor.

This approach, when compared to the traditional offline preparation of the data, is much simpler, more scalable, saves disk space and does not require complicated ratio-computation and data over- or downsampling.

6.2 Data augmentation for robustness

SOTASTREAM's augmentors provide a flexible framework for developing different methods for data augmentation, for example, case manipulation for robustness against different casing variants of the input text. It is demonstrated in the example in Figure 2, where `LowerCase` is an augmentor that lowercases the source text, and `TitleCase` converts both source and target sides to the English title-cased format. The frequency of each variant is easily controlled with the same `Mixer` used to join separate data sources. The on-the-fly approach simplifies experiments when testing multiple variations, which is often needed in order to find optimal augmentation methods and ratios, it minimizes the burden of experiment management.

Many other types of robustness augmentation (Li et al., 2019), such as source-side punctuation removal, spelling errors generation, etc., can be implemented in a similar way.

6.3 Filtering bad data examples

In SOTASTREAM it is straightforward to do data filtering on the fly. This type of filtering is especially useful in scenarios in which external data is used for model training or fine-tuning that cannot be manually filtered in a controlled way.

For example, a `URLFilter` filter that removes lines that have unmatched URLs between the source and target fields can be implemented using the provided `MatchFilter`:

```
def URLFilter(stream):
  pattern = r'\bhttps?:\S+[a-z]\b'
  return MatchFilter(stream, pattern)
```

6.4 Subword tokenization sampling

The boundary separating data generation from consumption can be blurred. For example, instead of producing raw text output, the tool could generate subwords, if provided with a subword model. This facilitates randomized sampling of different subword segmentations from a Unigram LM model with SentencePiece's Python wrapper:

```
import sentencepiece as sp
spm = sp.SentencePieceProcessor(
      model_file=SPM_VOCAB)

def spm_enc(stream, spm, fields=[0, 1]):
  for line in stream:
    for field in fields:
      line[field] = spm.encode(
          line[field], out_type=str,
          enable_sampling=True))
    yield line
```

6.5 Training document-context models

When training document models (e.g., Post and Junczys-Dowmunt (2023)), we can easily construct pseudo-documents on the fly if the training data is augmented with a document identifier field:

```
def read_docs(stream):
    doc, previd = [], None
    for line in stream:
        docid = line[2]
        if len(doc) and docid != previd:
            yield doc
            doc = []
        doc.append(line)
        previd = docid
    if len(doc):
        yield doc
```

A wrapper around this function could merge the source and target sides of the `Line` object, perhaps subject to parameters such as a maximum sequence length, a maximum number of sentences, and structural tokens to be used as affixes.

6.6 Alignments and other data types

SOTASTREAM has been primarily designed for machine translation, which requires providing source and target texts as separate fields. Other data types

or metadata can be generated on the fly or provided as additional fields in the input stream. By design the existing augmentors pass forward the unused fields, which makes introducing new fields that are used only by a subset of augmentors simple.

The example below demonstrates on-the-fly generation of word alignment using SimAlign (Jalili Sabet et al., 2020):

```
import simalign as sa
aln = sa.SentenceAligner()

def align(stream, aln, fields=[0, 1]):
  i, j = fields
  for line in stream,
    res = aln.get_word_aligns(line[i],
                              line[j])
    res = " ".join(f"{p[0]}-{p[1]}"
                   for p in res['mwmf'])
    line.append(res)
    yield line
```

The word alignment can be used directly by the trainer, e.g., for guided alignment training (Chen et al., 2016), or used by subsequent augmentors that may require it, e.g., constrained terminology translation annotations (Bergmanis and Pinnis, 2021).

6.7 Integration with data collection tools

SOTASTREAM can integrate tools like MTData, which automates the collection and preparation of machine translation data sets (Gowda et al., 2021). The following example shows `mtdata` pipeline which downloads the specified data sets and mixes them as per `--mix-weights` argument:

```
sotastream -n 1 mtdata --langs rus-eng \
  Statmt-news_commentary-16-eng-rus \
  Statmt-backtrans_ruen-wmt20-rus-eng \
  OPUS-paracrawl-v9-eng-rus \
  --mix-weights 2 1 1
```

6.8 Generating data sets for offline use

If the training tool does not support consuming training data from the standard input, SOTASTREAM can be used for static data generation. While the real advantages of SOTASTREAM accrue when making use of its on-the-fly data manipulations, this approach retains some of its benefits.

6.9 Other uses

The SOTASTREAM approach to factoring data generation, as well as SOTASTREAM itself, could also be used for generating non-textual content. The benefits of not writing data to disk would be greater in settings where input disk space is larger than plain text, such as translation from visual representations (Salesky et al., 2021). Nor does it need to be limited to sequence-to-sequence settings; we imagine the approach could be useful for training of LLMs.

7 Related Work

To our knowledge, SOTASTREAM is novel in presenting a framework for the generation of training data as a distinct component in the model training pipeline. It emphasizes a clean separation between data generation and training, multithreading for throughput, and the use of the standard UNIX pipeline interface.

It is not the first to propose streaming data, however. Although Fairseq's documentation emphasizes a preprocessing step,[10], Fairseq can also read and process raw data on the fly if it can be loaded completely into memory. Pytorch (Paszke et al., 2019) also provides "iterable-style" DataPipes for iterating over data samples,[11] but as far as we know, they are not widely used for machine translation training. They could, however, provide an interface to SOTASTREAM for Python-based training toolkits.

There are many libraries focused on data augmentation. A number of these are focused just on text augmentations, including nlpaug (Ma, 2019), TextAttack (Morris et al., 2020), and TextFlint (Gui et al., 2021). Another tool is AugLy (Papakipos and Bitton, 2022), a multimodal tool for text, audio, images, and video that provides robust training against adversarial perturbations. Many of these libraries could be useful within SOTASTREAM's general framework.

8 Conclusion

The data-preprocessing approach that is common in machine translation model training makes it possible to work with increasingly large datasets, but this ability does not come without costs. It is time-consuming to copy and process data, and can be expensive to store on disk. If data is compiled with model-specific parameters that tie the data to a particular model training, it prevents or at least complicates reusability. This problem is further exacerbated by research settings where one of the experimental parameters is manipulations of the training data, since each variant (and potentially

[10] https://web.archive.org/web/20230609072600/https://fairseq.readthedocs.io/en/latest/getting_started.html

[11] https://pytorch.org/data/main/torchdata.datapipes.iter.html

their cross-products) must be written to disk and then managed.

We have described an approach that separates data generation from data consumption, and shared SOTASTREAM, an implementation that makes use of the standard UNIX pipeline. The requirement is that preprocessing must now be computed on the fly. Our experiments show that this does not slow down training, nor does it affect the accuracy of the models trained. The approach provides flexibility, saves processing time and disk space, and simplifies experiment management.

Limitations

We have only investigated data consumption rates in a single toolkit, Marian, written in C++. It's possible that the online preprocessing requirements may be too much for toolkits written in languages without a proper thread implementation.

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.

Toms Bergmanis and Mārcis Pinnis. 2021. Facilitating terminology translation with target lemma annotations. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 3105–3111, Online. Association for Computational Linguistics.

Wenhu Chen, Evgeny Matusov, Shahram Khadivi, and Jan-Thorsten Peter. 2016. Guided alignment training for topic-aware neural machine translation. In *Conferences of the Association for Machine Translation in the Americas: MT Researchers' Track*, pages 121–134, Austin, TX, USA. The Association for Machine Translation in the Americas.

Thamme Gowda, Zhao Zhang, Chris Mattmann, and Jonathan May. 2021. Many-to-English machine translation tools, data, and pretrained models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations*, pages 306–316, Online. Association for Computational Linguistics.

Tao Gui, Xiao Wang, Qi Zhang, Qin Liu, Yicheng Zou, Xin Zhou, Rui Zheng, Chong Zhang, Qinzhuo Wu, Jiacheng Ye, Zexiong Pang, Yongxin Zhang, Zhengyan Li, Ruotian Ma, Zichu Fei, Ruijian Cai, Jun Zhao, Xinwu Hu, Zhiheng Yan, Yiding Tan, Yuan Hu, Qiyuan Bian, Zhihua Liu, Bolin Zhu, Shan Qin, Xiaoyu Xing, Jinlan Fu, Yue Zhang, Minlong Peng, Xiaoqing Zheng, Yaqian Zhou, Zhongyu Wei, Xipeng Qiu, and Xuanjing Huang. 2021. Textflint: Unified multilingual robustness evaluation toolkit for natural language processing. *CoRR*, abs/2103.11441.

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, Anna Currey, and Marcello Federico. 2022. Sockeye 3: Fast neural machine translation with pytorch.

Masoud Jalili Sabet, Philipp Dufter, François Yvon, and Hinrich Schütze. 2020. SimAlign: High quality word alignments without parallel training data using static and contextualized embeddings. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1627–1643, Online. Association for Computational Linguistics.

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.

Tom Kocmi, Rachel Bawden, Ondřej Bojar, Anton Dvorkovich, Christian Federmann, Mark Fishel, Thamme Gowda, Yvette Graham, Roman Grundkiewicz, Barry Haddow, Rebecca Knowles, Philipp Koehn, Christof Monz, Makoto Morishita, Masaaki Nagata, Toshiaki Nakazawa, Michal Novák, Martin Popel, and Maja Popović. 2022. Findings of the 2022 conference on machine translation (WMT22). In *Proceedings of the Seventh Conference on Machine Translation (WMT)*, pages 1–45, Abu Dhabi, United Arab Emirates (Hybrid). Association for Computational Linguistics.

Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia. 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.

Xian Li, Paul Michel, Antonios Anastasopoulos, Yonatan Belinkov, Nadir Durrani, Orhan Firat, Philipp Koehn, Graham Neubig, Juan Pino, and Hassan Sajjad. 2019. Findings of the first shared task on machine translation robustness. In *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*, pages 91–102, Florence, Italy. Association for Computational Linguistics.

Edward Ma. 2019. Nlp augmentation. https://github.com/makcedward/nlpaug.

John Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. 2020. TextAttack: A framework for adversarial attacks, data augmentation, and adversarial training in NLP. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 119–126, Online. Association for Computational Linguistics.

Artur Nowakowski, Gabriela Pałka, Kamil Guttmann, and Mikołaj Pokrywka. 2022. Adam Mickiewicz University at WMT 2022: NER-assisted and quality-aware neural machine translation. In *Proceedings of the Seventh Conference on Machine Translation (WMT)*, pages 326–334, Abu Dhabi, United Arab Emirates (Hybrid). 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 the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota. Association for Computational Linguistics.

Zoe Papakipos and Joanna Bitton. 2022. Augly: Data augmentations for robustness. *CoRR*, abs/2201.06494.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. 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 Köpf, Edward Z. Yang, Zach 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. *CoRR*, abs/1912.01703.

Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal. Association for Computational Linguistics.

Matt Post. 2018. A call for clarity in reporting BLEU scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191, Brussels, Belgium. Association for Computational Linguistics.

Matt Post and Marcin Junczys-Dowmunt. 2023. Escaping the sentence-level paradigm in machine translation.

Ricardo Rei, Craig Stewart, Ana C Farinha, and Alon Lavie. 2020. COMET: A neural framework for MT evaluation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2685–2702, Online. Association for Computational Linguistics.

Elizabeth Salesky, David Etter, and Matt Post. 2021. Robust open-vocabulary translation from visual text representations. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7235–7252, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

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

Xianfeng Zeng, Yijin Liu, Ernan Li, Qiu Ran, Fandong Meng, Peng Li, Jinan Xu, and Jie Zhou. 2021. WeChat neural machine translation systems for WMT21. In *Proceedings of the Sixth Conference on Machine Translation*, pages 243–254, Online. Association for Computational Linguistics.