Sockeye Alignment Matrix Report

ingus.pret

April 2024

Contents

1 Summary

During experimentation 4 types of systems were tested. For succinctness, let's define these acronyms:

- O - Systems that use original Sockeye code.
- N - Systems that use the new Sockeye code but don't enable alignment matrix training.
- NA - Systems that use the new Sockeye code and use alignment learning/alignment output.
- NSA - Systems that use the new Sockeye code and use alignment learning/alignment output and alignment shifting.

The positive conclusions of our experiments are:

- The changes made to the code not affect the runtime, RAM use, vRAM or translation quality by any considerable amount, if the newly added features are not enabled.
- Enabling alignment learning features does not affect translation quality metrics by any considerable amount.
- The quality of the alignments, learned by NSA systems, are satisfactory for our company's needs. The alignments are comparable to Marian NMT—the framework we used in production before Sockeye.
- Shifting the alignments by one target token forward increases the quality of the alignments learned.

The negative conclusions of our experiments are:

- Shard loading seems a lot slower when alignment learning features are enabled. About 2 minutes for 5'000'000 parallel sentences and their alignments.
- Peak RAM usage is 4x higher when alignment matrix learning is enabled during training.

2 Datasets

For training models we used a dataset of 5'000'000 parallel EN→LV sentences with statistical alignments. We refer to this dataset as the **Training Set** throughout the report.

For validation we used the EN→LV NewsDev set from WMT 2017. We refer to this dataset as the **Newsdev Dataset** throughout the report.

For alignment quality evaluation we also used a small dataset with human created alignments. The dataset contains 512 parallel EN$\rightarrow$LV sentences. We refer to this dataset as the **Evaluation Set** throughout the report.

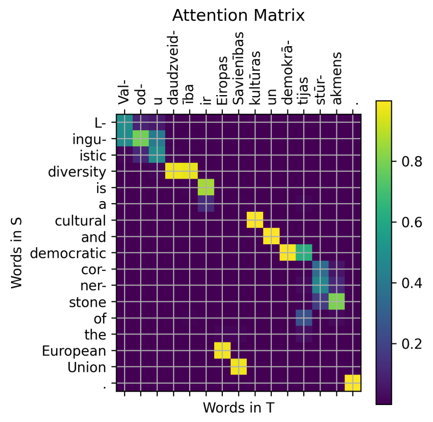
3 Learning Alignments and their Quality

[2] proposes that the alignments are calculated by doing the decoding twice: once with masked attention to construct the translation, the second with full attention to predict the alignments. We, however, opted for learning the alignments in the masked pass, to simplify implementation.

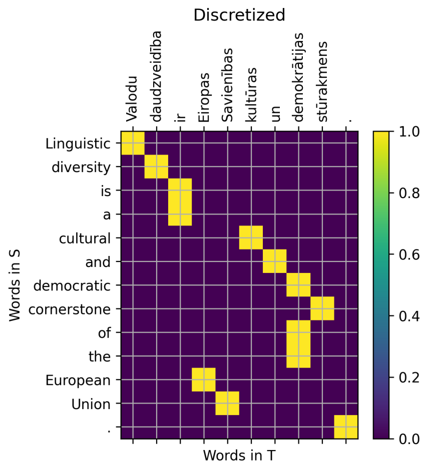
[1] shows that shifting the alignments one target token forward is beneficial. Shifting means the decoding step of a target token predicts the alignments of the previous target token rather than the current one. We added this as an argument –shift-alignments during the data preparation and translation steps. We preformed experiments (see Table 1) that show shifting alignments is indeed beneficial.

3.1 Experiment Metric Details

In further subsections You'll see metrics intended for discrete alignments. These were produced by an in-house heuristic algorithm we use to turn attention matrices over tokens into discrete word alignments (see Figure 1 for example). The discrete alignments were then used for calculating Recall, Precision, F1 and AER.



(a) Token-wise attention matrix. Output of a trained NSA translation model.



(b) Word-level discrete alignments, attained by passing (a) through our heuristic discretization algorithm.

Figure 1: Example input (a) and output (b) of our heuristic discretization algorithm.

3.2 Testing shifting alignments

Question asked in this subsection: Are alignments produced by NSA systems bettter than NA systems?

Test we performed: To verify that shifting alignments is useful, we trained 2 systems on the Training Set - NA without shifting the alignments and NSA with shifted alignments (see Appendix B for configs). Then we force decoded them on the Evaluation Set.

Result interpretation: All metrics suggest that shifting the alignments during training, indeed produces better alignments (See Table 1).

Table 1: Alignment quality metrics on the alignment Evaluation Set after discretization. NA - new with alignments; NSA - new with shifted alignments. MSE was calculated as the mean over all cells in attention matrices before discretization.

Metric	**NA**	**NSA**
Recall	0.741	**0.769**
Precision	0.671	**0.707**
F1	0.704	**0.737**
AER	0.296	**0.263**
MSE	0.0112	**0.0101**

3.3 Marian Comparison

Question asked in this subsection: Are alignments produced by NSA systems good enough for our purpouses?

How we tested it: We currently use Marian NMT. We wanted to compare the alignments produced by Sockeye and Marian. This however proved difficult. This is because Marian, unlike Sockeye, doesn't support forced decoding.

Never the less, we performed an experiment to see how different Marian's and Sockeye's produced alignments were. The experiment procedure was:

1. Train a Sockeye and Marian model on the same training dataset with alignments.
2. Use Marian to produce translations of the Newsdev Dataset.
3. Force decode Marian's translations using a Sockeye model trained with alignments.
4. Compare the output attentions.

Result interpertaion: Table 2 shows the results of the experiment. Both systems seem to produce very similar alignments. By visualizing a few examples we concluded that both Sockeye and Marian seem to perform comparably well. It's likely the disagreement among the systems arises from them not being very strongly trained models and having to align relatively poor translations.

We concluded that the alignments produced by our changes are satisfactory for our needs.

Table 2: Comparison between Marian and a NSA Sockeye system's produced alignments. Discretized Marian attentions were considered to be "Ground Truth" and Sockeye's discretized attentions were considered the predicting model. MSE was calculated for the attentions pre discretization.

Metric	**Value**
Recall	0.839
Precision	0.854
F1	0.847
AER	0.153
MSE	0.0040

4 Translation Quality Impact

Question asked in this subsection: Do our changes influence translation quality?

How we tested it: We trained 4 different systems (see Appendix B for configs). The data about validation metrics ware just parsed from the training logs.

Result Interpretation: The validation metrics during training can be seen seen in Figure 2 and Table 3.

It seems like translation quality is not impacted significantly by any of our changes even when alignment matrix learning is enabled. Which is a good sign we haven't broken anything.

Table 3: Comparison of validation metrics on best checkpoints for each system.

Metric	**O**	**N**	**NA**	**NSA**
Validation-ppl	26.8	26.7	27.1	27.1
Validation-bleu	0.263	0.265	0.263	0.265
Validation-chrf	53.7	53.7	53.6	53.7
Validation-rouge1	0.489	0.489	0.489	0.489
Validation-rouge2	0.300	0.301	0.302	0.303
Validation-rougel	0.456	0.456	0.459	0.459
Validation-length-ratio-mse	1.007	1.007	1.000	1.006
Validation-ter	59.1	59.0	58.8	58.9
Best Checkpoint Number	43	29	30	21

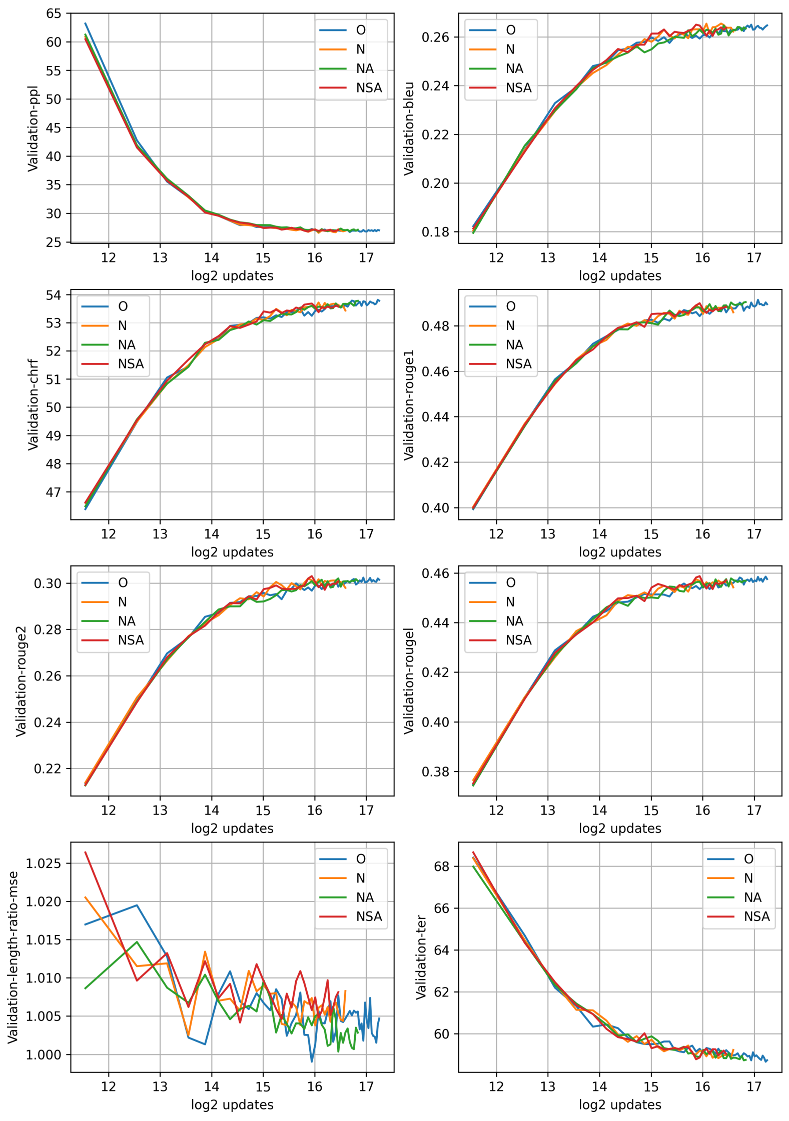


Figure 2: Validation metrics of systems during training. Horizonal axis is a logarithmic scale of update count.

5 Training Time Impact

Question asked in this section: How do our changes impact training time?

How did we test this: We trained 4 systems for 1000 Training Set batches of 9000 tokens (see Appendix D for configs).

Training time data was extracted from the Sockeye logs.

Result interpretation: Results can be seen in Table 4 and Table 5.

There was no impact on training time if the new features were not enabled. This is a good sign we haven't broken anything.

The total training time seemed to differ by about 2 minutes when using alignments (see Table 5). This discrepancy was investigated. It seems like shard loading without alignment matrices takes about 3 seconds, while shard loading with alignment matrices takes about 2 minutes.

The good news is training with alignments seems to have a negligible impact on training speed itself (see Table 4).

Table 4: Time it took (in seconds) to train on batches 100-1000 with each config/system. O - original Sockeye; N - new without alignments; NA - new with alignments; NSA - new with shifted alignments.

Resource Usage	**O**	**N**	**NA**	**NSA**
Time	352	351	356	354

Table 5: Time it took (in seconds) to train system from start to finish on 1000 batches (including loading data and creating checkpoint) with 5 million samples of prepared data (the Training Set). O - original sockeye; N - new without alignments; NA - new with alignments; NSA - new with shifted alignments. NA and NSA systems seem to be by about 2 minutes slower.

Resource Usage	**O**	**N**	**NA**	**NSA**
Time	705	706	831	830

6 Translation/Inference Time Impact

Question asked in this section: How is inference time impacted by our changes?

How we tested it: We produced four systems O, N, NA, NSA, each trained for one batch with a learning rate of 10^{-10}. And then used it to translate the Newsdev Dataset. (see Appendix C for configs)

We used untrained models, because of the assumption that untrained models are unlikely to accidentally predict EOS. This means all systems would just decode to their maximum translation limit, giving a fair comparison.

Inference times are just taken from the output sockeye translation logs.

Result interpretation:

Results can be seen in Table 6. The translation time doesn't seem to have been considerably affected.

Table 6: Time it took to translate 2002 samples from newsdev. O - original sockeye; N - new without alignments; NA - new with alignments; NSA - new with shifted alignments.

Resource Usage	**O**	**N**	**NA**	**NSA**
Time	772.7	758.9	749.6	758.6

7 RAM usage Impact

Question asked in this section: How is RAM usage impacted by our changes during training and translation?

How we tested it: For testing training we measured RAM usage during training on the Training Set for 1000 batches of size 9000 (see Appendix D for configurations).

For testing translation we measured RAM usage on 4 systems that were trained for 1 batch with a learning rate of 10^{-10}. (see Appendix C for configuration)

RAM use was measured by logging the RAM of the specific process being run and all it's child processes (See Appendix A for code).

Result interpretation: The results are seen in Table 7.

During training, when not enabling the alignment learning features, the changes in memory use seem negligible - about 1%.

However, enabling alignment learning seemed to increase RAM use considerably. This was investigated. It seems like peak RAM use occurs during shard loading.

No apparent differences in RAM use a visible during translation.

Table 7: Peak RAM usage during training and inference in MiB. O - original sockeye; N - new without alignments; NA - new with alignments; NSA - new with shifted alignments.

Resource Usage	**O**	**N**	**NA**	**NSA**
VMS RAM Usage Training	28’576	28’600	32’389	32’430
RSS RAM Usage Training	4’268	4’328	16’647	16’638
VMS RAM Usage Translation	11’260	11’261	11’261	11’237
RSS RAM Usage Translation	2’350	2’344	2’359	2’340

8 vRAM usage Impact

Question asked in this section: How is vRAM usage impacted by our changes during training and translation?

How we tested it: For this we measured vRAM use of the same systems as in the previous section.

Model vRAM usage was measured by just looking at vRAM use of the specific GPU overall (See appendix A for code).

Result interpretation: The vRAM usage seems almost unaffected (see Table 8).

Table 8: Peak vRAM usage during training and inference. O - original sockeye; N - new without alignments; NA - new with alignments; NSA - new with shifted alignments.

Resource Usage	**O**	**N**	**NA**	**NSA**
vRAM Usage Training	15’573	15’573	15’593	15’593
vRAM Usage Translation	931	931	937	933

References

[1] Yun Chen, Yang Liu, Guanhua Chen, Xin Jiang, and Qun Liu. Accurate word alignment induction from neural machine translation. *CoRR*, abs/2004.14837, 2020.

[2] Sarthak Garg, Stephan Peitz, Udhyakumar Nallasamy, and Matthias Paulik. Jointly learning to align and translate with transformer models. *CoRR*, abs/1909.02074, 2019.

A RAM and vRAM logging script

In large part this code is courtesy of GPT4. The script is called ”monitor_ram2.py”.

```python
import subprocess
import psutil
import time
import sys
import nvidia_smi

def log_memory_usage(cmd, log_file):
    # Initialize the GPU access
    nvidia_smi.nvmlInit()
    handle = nvidia_smi.nvmlDeviceGetHandleByIndex(2)  # For GPU with ID 2

    # Start the process
    process = subprocess.Popen(cmd, shell=True)
    pid = process.pid
    main_process = psutil.Process(pid)

    with open(log_file, ’w’) as f:
        # Log memory usage every second
        try:
            while True:
                if process.poll() is not None:  # Check if main process has terminated
                    f.write("Main process terminated.\n")
                    break
                # Check memory info on GPU
                info = nvidia_smi.nvmlDeviceGetMemoryInfo(handle)
                gpu_mem_usage = info.used
                children = main_process.children(recursive=True)
                total_vms = sum([child.memory_info().vms for child in children] + [main_process.memory_info().vms])
                f.write(f"Total VMS: {total_vms} bytes, GPU Memory Usage: {gpu_mem_usage} bytes\n")
                time.sleep(1)  # Log every second
        except psutil.NoSuchProcess:
            f.write("Process completed or terminated unexpectedly.\n")
        except nvidia_smi.NVMLError as e:
            f.write(f"Failed to read GPU memory usage: {e}\n")
        finally:
            # Additional check for any remaining active children
            if any(child.is_running() for child in main_process.children(recursive=True)):
                f.write("Child processes still running after main process terminated.\n")
            else:
                f.write("All processes terminated.\n")
            # Cleanup the NVML handle
            nvidia_smi.nvmlShutdown()

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: python memory_logger.py ’<command>’ ’<log_file>’")
    else:
        _, command, log_file = sys.argv
        log_memory_usage(command, log_file)
```

B Long Training Experiments

The training for different systems was performed on different machines.

4 systems (O, N, NA and NSA) were allowed to train until the early stopping condition was met. The section contains configurations used to train the systems.

For original Sockeye (O):

```
python -m torch.distributed.run --master_port 1337 --no_python
--nproc_per_node 3 python -m sockeye.train \
--prepared-data datanoalign/ \
--output models \
--validation-source ./test.en \
--validation-target ./test.lv \
--dist \
--learning-rate-scheduler-type inv-sqrt-decay \
--learning-rate-warmup 16000 \
--initial-learning-rate 0.06325 \
--optimizer adam \
--optimizer-betas 0.9:0.98 \
--batch-size 9000 \
--update-interval 8 \
--checkpoint-interval 3000 \
--amp \
--batch-type max-word \
--seed 1 \
--quiet-secondary-workers \
--max-num-checkpoint-not-improved 10 \
--checkpoint-improvement-threshold 0.0003 \
--optimized-metric bleu \
--decode-and-evaluate -1 \
--keep-last-params 25 \
-o baselineoriginal
```

For Sockeye with changes but alignment learning disabled (N):

```
python -m torch.distributed.run --master_port 1337 --no_python
--nproc_per_node 2 python -m sockeye.train \
--prepared-data data/ \
--output models \
--validation-source ./test.en \
--validation-target ./test.lv \
--dist \
--learning-rate-scheduler-type inv-sqrt-decay \
--learning-rate-warmup 16000 \
--initial-learning-rate 0.06325 \
--optimizer adam \
--optimizer-betas 0.9:0.98 \
--batch-size 9000 \
--update-interval 12 \
--checkpoint-interval 3000 \
--amp \
--batch-type max-word \
--seed 1 \
--quiet-secondary-workers \
--max-num-checkpoint-not-improved 10 \
--checkpoint-improvement-threshold 0.0003 \
--optimized-metric bleu \
--decode-and-evaluate -1 \
--keep-last-params 25 \
-o baseline
```

For Sockeye with alignment learning (NA):

```
python -m torch.distributed.run --master_port 1338 --no_python
--nproc_per_node 2 python -m sockeye.train \
--prepared-data data/ \
--output models \
--validation-source ./test.en \
--validation-target ./test.lv \
--dist \
--learning-rate-scheduler-type inv-sqrt-decay \
--learning-rate-warmup 16000 \
--initial-learning-rate 0.06325 \
--optimizer adam \
--optimizer-betas 0.9:0.98 \
--batch-size 9000 \
--update-interval 12 \
--checkpoint-interval 3000 \
--amp \
--batch-type max-word \
--seed 1 \
--quiet-secondary-workers \
--max-num-checkpoint-not-improved 10 \
--checkpoint-improvement-threshold 0.0003 \
--optimized-metric bleu \
--decode-and-evaluate -1 \
--keep-last-params 25 \
--attention-alignment-layer 4 \
--alignment-matrix-weight 0.05 \
-o baseline
```

For Sockeye with shifted alignment learning (NSA):

```
python -m torch.distributed.run --master_port 1337 --no_python
--nproc_per_node 2 python -m sockeye.train \
--prepared-data datashifted/ \
--output models \
--validation-source ./test.en \
--validation-target ./test.lv \
--dist \
--learning-rate-scheduler-type inv-sqrt-decay \
--learning-rate-warmup 16000 \
--initial-learning-rate 0.06325 \
--optimizer adam \
--optimizer-betas 0.9:0.98 \
--batch-size 9000 \
--update-interval 12 \
--checkpoint-interval 3000 \
--amp \
--batch-type max-word \
--seed 1 \
--quiet-secondary-workers \
--max-num-checkpoint-not-improved 10 \
--checkpoint-improvement-threshold 0.0003 \
--optimized-metric bleu \
--decode-and-evaluate -1 \
--keep-last-params 25 \
--attention-alignment-layer 4 \
--alignment-matrix-weight 0.05 \
--shift-alignments \
-o baseline
```

C Translation Speed, RAM and vRAM Measurement

These experiments were done on a server with 100GB of RAM, and a Quadro RTX 6000 GPU.

Used for measuring N, NA & NSA systems:

```
export CUDA_VISIBLE_DEVICES=2
python monitor_ram2.py "python -m sockeye.train --prepared-data
    ./../en-lv-sockeye-new/datanoalign/ -o Ni --validation-source
    ./../en-lv-sockeye-new/test.en --validation-target ../en-lv-
    sockeye-new/test.lv --batch-size 9000 --max-updates 1 --initial
    -learning-rate 0.000000001" "logNit.txt"
python monitor_ram2.py "python -m sockeye.train --prepared-data
    ./../en-lv-sockeye-new/data/ -o NAi --validation-source ./../en
    -lv-sockeye-new/test.en --validation-target ../en-lv-sockeye-
    new/test.lv --batch-size 9000 --max-updates 1 --initial-
    learning-rate 0.000000001 --align-attention" "logNAit.txt"
python monitor_ram2.py "python -m sockeye.train --prepared-data
    ./../en-lv-sockeye-new/data/ -o NSAi --validation-source ./../
    en-lv-sockeye-new/test.en --validation-target ../en-lv-sockeye-
    new/test.lv --batch-size 9000 --max-updates 1 --initial-
    learning-rate 0.000000001 --shift-alignments --align-attention"
     "logNSAit.txt"
python monitor_ram2.py "python -m sockeye.translate -m Ni --output-
    type json --input ./../en-lv-sockeye-new/test.en >> trash.txt"
    "logNi.txt"
python monitor_ram2.py "python -m sockeye.translate -m NAi --output
    -type json --input ./../en-lv-sockeye-new/test.en >> trash.txt"
     "logNAi.txt"
python monitor_ram2.py "python -m sockeye.translate -m NSAi --
    output-type json --input ./../en-lv-sockeye-new/test.en --shift
    -alignments >> trash.txt" "logNSAi.txt"
```

Used for measuring O system:

```
export CUDA_VISIBLE_DEVICES=2
python monitor_ram2.py "python -m sockeye.train --prepared-data
    ./../en-lv-sockeye-new/datanoalign/ -o Oi --validation-source
    ./../en-lv-sockeye-new/test.en --validation-target ../en-lv-
    sockeye-new/test.lv --batch-size 9000 --max-updates 1 --initial
    -learning-rate 0.000000001" "logOit.txt"
```

```
python monitor_ram2.py "python -m sockeye.translate -m Oi --input
    ./../en-lv-sockeye-new/test.in >> trash.txt" "logOi.txt"
```

D Training Speed, RAM and vRAM Measurment

The commands used to perform training speed, RAM use and vRAM use for N, NA and NSA systems were:

```
export CUDA_VISIBLE_DEVICES=2
python monitor_ram2.py "python -m sockeye.train --prepared-data
    ./../en-lv-sockeye-new/datanoalign/ -o N --validation-source
    ./../en-lv-sockeye-new/test.en --validation-target ../en-lv-
    sockeye-new/test.lv --batch-size 9000 --max-updates 1000" "logN
    .txt"
python monitor_ram2.py "python -m sockeye.train --prepared-data
    ./../en-lv-sockeye-new/data/ -o NA --validation-source ./../en-
    lv-sockeye-new/test.en --validation-target ../en-lv-sockeye-new
    /test.lv --batch-size 9000 --max-updates 1000 --align-attention
    " "logNA.txt"
python monitor_ram2.py "python -m sockeye.train --prepared-data
    ./../en-lv-sockeye-new/data/ -o NSA --validation-source ./../en
    -lv-sockeye-new/test.en --validation-target ../en-lv-sockeye-
    new/test.lv --batch-size 9000 --max-updates 1000 --shift-
    alignments --align-attention" "logNSA.txt"
```

The commands used to perform training speed, RAM use and vRAM use for original sockeye were:

```
export CUDA_VISIBLE_DEVICES=2
python monitor_ram2.py "python -m sockeye.train --prepared-data
    ./../en-lv-sockeye-new/datanoalign/ -o O --validation-source
    ./../en-lv-sockeye-new/test.en --validation-target ../en-lv-
    sockeye-new/test.lv --batch-size 9000 --max-updates 1000" "logO
    .txt"
```