Attention is All You Need - Reproducibility

VU Deep Learning for Natural Language Processing

Maximilian Kleinegger*
Vienna University of Technology

Lukas Mahler*
Vienna University of Technology

Josef Taha*
Vienna University of Technology

*All authors contributed equally to this research.

1 Introduction

Transformers have fundamentally changed NLP by replacing recurrent architectures with self-attention mechanisms. The original paper, *Attention is All You Need* [2] introduced this novel architecture, leading to state-of-the-art results in sequence modeling tasks.

In this project, we aimed to (i) reproduce the original Transformer model using Tensor2Tensor, (ii) re-implement the Transformer model in PyTorch and Candle (Rust) and (iii) compare the efficiency, scalability, and challenges of different implementations on the translation task.

This report presents our methodology, results, and key takeaways from this reproduction study.

2 Methodology

2.1 Dataset Preprocessing and Evaluation Metrics

To ensure fair comparability across implementations, we required a common dataset and common vocabulary. The original Tensor2Tensor repository provided an English-to-German dataset with 4.5 million translations, pre-tokenized and stored as TFRecords.

(i) We reused this dataset instead of constructing a new vocabulary or tokenization pipeline, ensuring consistency with the original paper and preventing preprocessing-related discrepancies.

(ii) Since TFRecords is TensorFlow-specific, we converted it to Parquet, a universal, technology-independent format. This allowed integration into PyTorch and Candle, ensuring all models trained on identical data for fair comparison.

(iii) The paper did not specify which BLEU variant was used for evaluation. To resolve this, we re-implemented the BLEU scoring function and confirmed that the original model used BLEU-4. We then applied the same scoring function across all experiments to maintain consistent and comparable evaluation metrics.

2.2 Reproducing Tensor2Tensor

Tensor2Tensor (T2T) is a TensorFlow-based library that provides implementations of various models, including the original Transformer implementation introduced by the *Attention is All You Need* paper. It also includes pre-processed datasets and hyperparameter sets used in this paper, as well as training scripts, facilitating direct and reproducible model training and evaluation .

2.2.1 Motivation

The first step in our study was to reproduce the results from the paper using the official implementation available in the T2T repository. This provided a baseline for evaluating our re-implementations and ensured that the reported performance metrics could be verified within our computational constraints.

Our focus was the English-to-German translation task from the WMT 2014 dataset, which consists of approximately 4.5 million sentence pairs. The original paper reports results using two configurations: Transformer-Base and Transformer-Big (see hyperparamters in Table 1). We aimed to replicate these models and their training procedures using Tensor2Tensor.

2.2.2 Implementation Details

The original models were trained using a cluster of eight NVIDIA P100 GPUs, each equipped with 16

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

GB of VRAM. To ensure a comparable computational environment, we utilized eight NVIDIA T4 GPUs, which also feature 16 GB of VRAM but offer approximately 33% higher floating-point operations per second (FLOPs) compared to the P100. The hyperparameter configurations were directly adopted from the Tensor2Tensor library, with necessary adaptations made to the training scripts to accommodate the updated hardware specifications.

However, running the original code presented several challenges due to the discontinuation of Tensor2Tensor and its reliance on outdated dependencies. The repository has not been maintained for over six years, leading to compatibility issues with modern software. Additionally, the implementation was originally developed using Python 2.7, which is no longer officially supported. Further complications arose from the use of older CUDA and TensorFlow versions, which required careful selection of compatible configurations.

To overcome these issues, we created a custom Docker environment that allowed us to replicate the original setup. The environment was configured with Python 2.7 to ensure compatibility with Tensor2Tensor, while CUDA 9.0 was selected as newer versions were incompatible with the required TensorFlow version. We used TensorFlow v1.13.1, which was the last known working version for Tensor2Tensor.

This setup enabled us to successfully execute the original training scripts on our T4 GPUs, providing a direct basis for comparison with the results reported in the original paper.

2.2.3 Challenges and Limitations

The primary challenge was software compatibility rather than algorithmic inconsistencies. While the original code was structured for reproducibility, the lack of maintenance made direct execution on modern systems difficult.

Key challenges included: (i) dependency management, as the requirement for Python 2.7 made it difficult to use modern tools, and many required packages had been deprecated, making it challenging to find a CUDA version compatible with both Tensor2Tensor and our hardware; (ii) hardware differences, as the original paper used P100 GPUs, whereas we used T4 GPUs, resulting into having to consider GPU differences in training time and performance; (iii) reproducibility trade-offs, since we closely replicated the original environment, but relying on legacy software made the approach less sustainable, and ideally, pre-configured Docker images should have been provided to simplify future reproducibility; and (iv) execution complexity, as resolving software conflicts required manual intervention, and debugging issues related to deprecated TensorFlow APIs also consumed additional time.

Table 1: Transformer Hyperparameters (Base vs. Big)

Hyperparameter	**Base**	**Big**
Source Vocabulary Size (`src_vocab_size`)	33710	33710
Target Vocabulary Size (`tgt_vocab_size`)	33710	33710
Model Dimension (`d_model`)	512	1024
Number of Heads (`num_heads`)	8	8
Number of Layers (`num_layers`)	6	6
Feed-Forward Dimension (`d_ff`)	2048	4096
Maximum Sequence Length (`max_seq_length`)	512	512
Dropout Rate (`dropout`)	0.1	0.3
Label Smoothing (`label_smoothing`)	0.1	0.1
Batch Size (`batch_size`)	16	700
Training Steps (`total_steps`)	100000	300000

2.3 Reimplementation in PyTorch

After completing this course, PyTorch needs no introduction.

2.3.1 Motivation

The Transformer has been implemented in PyTorch multiple times. However, in terms of reproducibility, it is interesting to explore whether (a) we can reproduce the results without access to existing code and (b) if newer Python versions improve overall performance. Therefore, by reimplementing the Transformer in PyTorch, we aim to determine whether (a) is achievable and (b) holds true.

[0] `https://medium.com/@bavalpreetsinghh/transformer-from-scratch-using-pytorch-28a5d1b2e033`
[0] `https://towardsdatascience.com/build-your-own-transformer-from-scratch-using-pytorch-84c850470dcb`

2.3.2 Implementation Details

Implementing the Transformer in PyTorch is straightforward. Using multiple sources as guidance and in combination with following the paper, this can be achieved rather quickly. However, obtaining a fully functional and high-performing version required multiple iterations.

- In the first iteration, we used the data provided by the tensor2tensor library. This approach worked overall; however, we noticed that we did not include a start-of-sentence (<sos>) symbol, which resulted in a loss of information.

- In the second iteration, we addressed this issue by including a start-of-sentence (<sos>) symbol. This adjustment improved the results.

- In the final iteration, we increased the number of dropout layers within the model to assess whether it could further enhance performance.

2.3.3 Training

For training, we used the hyperparameters from the base model in the original paper, as we faced computational constraints compared to Google. Where necessary, we made adaptations, as shown in Table 1. Furthermore, we encountered several challenges, which we discuss in the following section.

When running our implementation, we initially encountered the issue that the batch size of 25,000 tokens for input and target samples was far too large for the standard implementation to handle. To resolve this, we reduced the batch size to just 16 records instead of the possible 700 per batch.

Additionally, we faced the problem that only one GPU was being utilized. We initially did not consider DeepSpeed [1], but instead used *torch.nn.parallel.DistributedDataParallel*, which replicates the model across all available GPUs and averages weights across layers after each backpropagation step. This approach allowed us to utilize multiple GPUs rather than just a single one. However, we were still constrained by memory, which limited us to a batch size of 16. In this scenario, DeepSpeed could have been beneficial, but we did not utilize it.

Lastly, we observed that implementations found on GitHub and in tutorials differ in how they implement the Add & Norm layer. Many do not normalize the original output within the residual layer but instead only normalize the output from the Attention Head or Feed-Forward Layer.

[1]`https://github.com/microsoft/DeepSpeed`

2.3.4 Inference

During inference, we initially encountered issues loading the model due to a misconfiguration of the embedding dimensions, which was quickly resolved. However, overlooking such small details can result in translations that appear generally acceptable but ultimately lead to repetitive outputs, where the same word is repeated over and over.

Furthermore, we initially used a greedy selection approach but quickly transitioned to beam search. Both methods produced similar results, but we observed that beam search led to fewer errors and slightly better overall performance. Therefore, all reported results were obtained using beam search.

2.4 Reimplementation in Candle (Rust)

Candle is a deep learning (DL) framework fully written in Rust by Hugging Face, which can be similarly used as PyTorch.

2.4.1 Motivation

Traditionally, when conducting DL tasks in Python in order to satisfy the high performance requirements, a framework written in a non garbage collected compiled language like C or C++ is used, which provides Python bindings as interface to use the framework in Python code.

In contrast, candle provides a different approach: by utilizing zero-cost abstractions, a language feature rust is well known for, it allows to implement the model in a high-level interface directly in Rust, eliminating cross-language dependencies which simplifies the deployment process considerably, allowing to build a single executable which can be run directly on a target system without installing dozens of Python dependencies.

2.4.2 Implementation Details

The first draft of the implementation is based on optimus, a tutorial showcasing how to implement a Transformer model using Candle [1] . However, it is incomplete, as it only contains class definitions, no code for testing or inference. Additionally, large parts of the tutorial code had to be re-written, because it contained multiple bugs and did some implementation details differently than described in the paper.

Pre-processing is done by creating batches, this is done in native Rust code. We map those batches to Tensors, which are copied one by one to the GPU when it is their turn to be processed. When the training of the model is finished, the weights are written to the file system in the safetensors format, allowing to be loaded for inference or additional training.

Directly using a compiled language made the pre- and post-processing steps much simpler and faster, as we are able to use native data types instead of having to fall back to a pre-compiled library like Pandas or Numpy as is the case for Python in order to prevent a substantial hit to both the runtime performance as well as memory consumption.

2.4.3 Challenges

Candle has a very similar programming interface to PyTorch, however, some methods are not (yet) implemented in Candle, which we missed, given our previous experience with PyTorch. Following missing features required substantially more complex code than the PyTorch implementation:

- Masking: PyTorch provides the handy masked_fill method, which automatically broadcasts a provided Tensor and applies it as mask on the Tensor we are calling it on. In candle we had to do the masking manually, which was additionally harder because it also lacks most of the binary logical operations (e.g. and, or) and hence we had to fall back to using summing up integer Tensors instead of applying the 'and' operation.

- Loss calculation: Filtering the padding tokens when calculating the loss is directly done by PyTorch by passing the token id for the padding token when calculating the cross-entropy loss. In our however, we unfortunately missed this in our presented version, which lead to a BLEU-score of 0 due to always predicting the padding token. We fixed it by manually filtering the padding tokens first and calculating the loss on the filtered tensors.

3 Results

Table 2: Comparison of BLEU Scores and Training Times for different implementations and runs

Run	Model	BLEU 4	Time	Epochs	GPUs
1	Tensor2Tensor (Big)	28	2.5 days	45	8
2	Tensor2Tensor (Base)	26	8 hours	15	8
3	Tensor2Tensor (Small)	7	20 mins	0.35	8
4	PyTorch (v1)	13	10 hours	0.35	8
5	PyTorch (v2) (<sos>)	14	10 hours	0.35	8
6	PyTorch (v3) (dropout)	13.6	10 hours	0.35	8
7	Candle (Rust)	0	12 hours	0.35	1

We successfully reproduced the BLEU scores reported in the original paper for both the Transformer-Base and Transformer-Big models on the English-to-German translation task. Training time was reduced by 30%, which aligns with the 33% higher FLOPs of our NVIDIA T4 GPUs compared to the P100 GPUs used in the original study. These consistent results in model performance and efficiency confirm the success of our reproduction.

Considering Table 2, we can clearly see that we managed to reproduce some results for the PyTorch implementation. To enable a fair comparison, we trained Tensor2Tensor for 2,500 steps, as this is equivalent in terms of the number of samples processed to training for 100,000 steps with a batch size of 16.

As evident from the results, all PyTorch versions outperform the Tensor2Tensor (small) model in terms of BLEU score. However, in terms of training time, we are significantly outperformed by Tensor2Tensor. Additionally, we observe that incorporating a proper start-of-sentence (SOS) symbol leads to notable improvements, whereas increasing the dropout rate does not necessarily enhance performance.

Due to our computational constraints and the relatively low number of training samples compared to Tensor2Tensor (base), we can indicate that reproduction is possible, but we cannot fully guarantee it, as achieving complete reproducibility would require significantly longer training.

This highlights our biggest drawback—the comparatively poor efficiency of a plain PyTorch implementation compared to the highly optimized Tensor2Tensor library.

4 Key Takeaways

Transformer-Base and Transformer-Big BLEU scores were successfully reproduced using Tensor2Tensor, validating the original results despite outdated dependencies complicating setup. PyTorch, on modern libraries, achieved better BLEU scores but lacked large-scale optimizations, resulting in slower training. Proper start-of-sentence

(SOS) handling improved performance, while increased dropout had minimal impact.

The Candle (Rust) implementation proved feasible but lacked key PyTorch functions (e.g., masking, loss computation), requiring workarounds and increasing bug susceptibility. An error in loss calculation and improper padding masking led to a BLEU score of 0. Limited to single-GPU training, performance was constrained, yet Rust's efficiency and dependency-free deployment make Candle a promising alternative for future research.

Efficiency remains a major challenge: Tensor2Tensor, despite being outdated, remains highly optimized, training faster than PyTorch. While PyTorch and Candle offer more flexible solutions, they require further optimizations (e.g., DeepSpeed, better multi-GPU support) to match Tensor2Tensor's efficiency.

References

[1] Nihal Pasham. *Optimus*. Accessed: 2025-02-02. 2025. URL: `https://github.com/nihalpasham/optimus`.

[2] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: `1706.03762`. URL: `http://arxiv.org/abs/1706.03762`.