Faculty Dialog Systems and Machine Learning

Implementing Transformers

Project Report

Author	Nils Reck	2898155
Supervisors	Dr. Carel van Niekerk	Dr. Hsien-Chin Lin

29. 01. 2025

1 Introduction

The Transformer model [Vas17] established the foundation for more performant and context-aware sequence transduction by removing the recurrent or convolutional means of previous state-of-the-art models. This is mainly due to the Transformer's ability to process multiple sequences at once. Up until today, the Transformer model remains the architecture of choice for top-performing large language models, like GPT-4o.
This report showcases my attempt to implement a custom Transformer model for a German-English translation task and aligns with the practicals conducted during the course. In particular, it focuses on providing the reader with detailed knowledge about its components and their interplay, as well as my personal insights and struggles during development and training. Thus, the report commences with a methodology section, explaining the modular components of a Transformer and the overall architecture.

2 Methodology

The methodology section describes the key components of the Transformer-based translation model, including data and model architecture (see Figure 1).

2.1 Data Preprocessing

Before the model can process the input data, the data has to be encoded in a numerical representation that the model can interpret. For this, a shared tokenizer is trained over the source and target sequences, which maps a token (a word or subword) of a sequence to a number and vice versa. Correctly aligning the sequences was particularly challenging and required multiple iterations to get right. A key indicator of misalignment was that the training loss converged to zero, while the validation loss remained high, suggesting poor generalization to unseen data. This led me to realize that the model was learning to always predict the token it was erroneously able to look at.

2.2 Embedding Layers

After the data is correctly preprocessed, the embedding layer creates a 512-dimensional vector representation for each encoded token of the input and target sequence. Consistent with the original Transformer architecture, we apply parameter sharing by using the same set of weights for both embedding layers and the pre-softmax linear transformation, which maps the embeddings back to their respective token index. Sharing parameters between the encoder and decoder embedding layers offers several advantages. First, it can significantly reduce the model size while maintaining model performance [PW17]. Second, parameter sharing reduces the degrees of freedom of the model, thus implicitly applying regularization by forcing different parts of the model to use the same

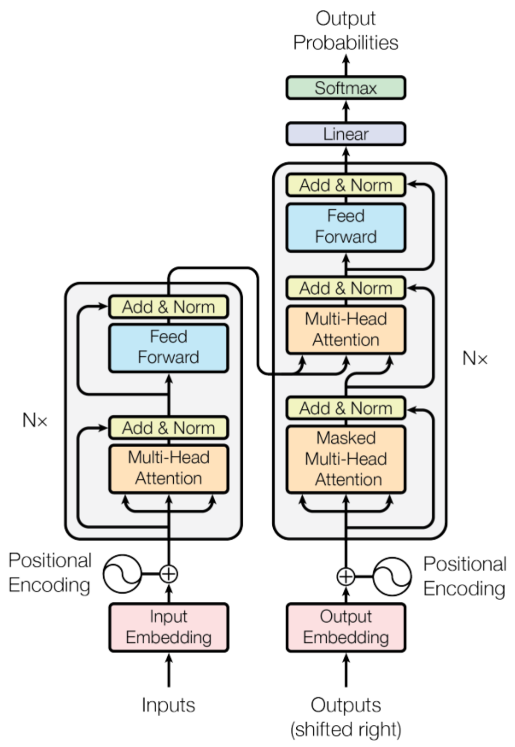

Figure 1: The original transformer architecture, adapted from Vaswani et al. [Vas17]

parameters, preventing the model from overfitting. Additionally, the efficiency of the model improves because shared parameters allow for faster updates and fewer memory operations. Unlike recurrent architectures, which process sequences step by step, the Transformer processes entire sequences in parallel. To compensate for the lack of sequence order awareness, the positional encoding layer enriches the representations with fixed positional information.

2.3 Encoder Stack

The encoder consists of six identical layers, each designed to transform the input sequence into a context-rich representation. As illustrated by Figure 1, each layer comprises two sub-layers: a multi-head self-attention mechanism (Section 2.5) and a position-wise feed-forward network (Section 2.6), each followed by a residual connection [HZRS15] and layer normalization [BKH16] (Section 2.7) to stabilize training and improve gradient flow.
Residual connections, defined as $y = \mathbf{x} + f(\mathbf{x})$, preserve the original signal while adding important features from multi-head attention or feed-forward layers, al-

leviating the problem of vanishing gradients during backpropagation. If the transformation $f(\mathbf{x})$ collapses to zero (e.g. due to all weights and biases being pushed to zero), the output reduces to $y = \mathbf{x}$, ensuring that the original signal is preserved when the layer does not learn anything. Residual connections can also be described by the residual mapping $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$, emphasizing that the network only needs to learn a small transformation when $f(\mathbf{x})$ is close to the identity function. Learning a function close to the identity function, residual blocks slightly refine existing features instead of learning full (high variance) functions from scratch.

2.4 Decoder Stack

The decoder also consists of six identical layers. In addition to the two sub-layers of the encoder, it has a second multi-head attention mechanism over the outputs of the encoder. Consistent with the encoder, residual connections and layer normalization are employed after each sub-layer. Finally, the output of the decoder undergoes a linear transformation. After that, softmax is applied to convert the output into probabilities to predict the next token.

2.5 Attention

The attention function injects contextual information about related tokens into each token's representation. This process enables the model to capture dependencies between words, regardless of their position in the sequence.
The first step is to create the query(Q), key(K), and value(V) vectors from the encoder or decoder input vectors by multiplying them with three matrices that are learned during training. These matrices must be learned in a way that they reflect meaningful similarity relationships in terms of attention. To ensure that attention is applied correctly, two types of masks are used: a padding mask, which prevents attention from being applied to padding tokens, and a causal mask, which ensures that in the decoder, attention cannot be applied to future tokens.
According to Equation (1), the attention function then first computes a score for each token in the sequence relative to every other token by taking the dot product of the query vector with the transposed key vector:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

This computation, along with all other operations of the attention mechanism, is performed in parallel for all tokens in each sequence across the entire batch. Next, the result is scaled by $\sqrt{d_k}$ to avoid exploding gradients and improve stability. Then a softmax function is applied to maintain relevant words, subside words we can mostly ignore, and prepare the output to be summed up. Finally, by multiplying the softmax scores by V produces a new representation for each token. While it retains most of its original structure, it is enriched with contextual information form the most relevant tokens for our translation task.

2.6 Position-Wise Feed-Forward Networks

According to Equation (2), the FFN introduces a higher-dimensional space to explore non-linear combinations of features present in the token embeddings that it could not explore in the original embedding space:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2}$$

That happens by the first linear transformation $xW_1 + b_1$. Next, $\max(0, xW_1 + b_1)$ (ReLU) introduces non-linearity. The FFN has two linear layers of size $(d_{\text{model}}, d_{\text{ffn}})$ and $(d_{\text{ffn}}, d_{\text{model}})$, respectively. Finally, the non-linearly transformed representation is projected back into the original space, d_{model}, such that the model is forced to focus on the most significant feature combinations.

2.7 Normalization Layer

Layer normalization is applied after each self-attention and feed-forward sublayer according to Equation (3):

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta, \tag{3}$$

where x is the input vector, in our case the token embedding, μ the mean of x, calculated across the features, σ is the standard deviation, also calculated across the features:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad\qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \tag{4}$$

H is the number of features for each token representation, γ and β are optional, learnable parameters to scale and shift the normalized values.
In a transformer architecture, the layer normalization layer serves different purposes: it stabilizes training by normalizing the distributions of the layer inputs, thus preventing exploding or vanishing gradients, which would also have adverse, covariate effects on the surrounding layers in the forward and backward passes. Additionally, contrary to batch normalization, layer normalization handles variations in sequence length better, since it computes the mean and variance along the features of the token and not across the individual features across the batch.

3 Training

This section covers the training regime and a comparison between CPU and GPU training. The code is available on GitLab[1].

[1]`https://git.hhu.de/nirec101/transformer_project`

3.1 Data

We train on the WMT 17 German-to-English dataset[2], consisting of about 4.9 million sentence pairs after filtering out sequences exceeding 64 tokens in length. To not inflict unnecessary load on the GPU during training, we preprocess the datasets beforehand. The sequences are encoded using byte-pair encoding [BGLL17], which has a shared source-target vocabulary of 50000 tokens. We use only 5% of the training data for benchmarking CPU vs. GPU performance with a batch size of 32 on both. For the final model performance reported in Section 4, however, we train on the entire dataset with a batch size of 256 sentence pairs.

3.2 Training and Schedule

We train our models on a single node with five processing cores and a single NVIDIA A100 GPU for around 170000 steps (10 epochs). Since GPUs are optimized for parallel computation, they are well-suited for the highly parallel nature of sequence processing in Transformer models. Unlike CPUs, which are designed for handling a wide range of sequential operations, the A100 distributes the workload across its many cores, consisting of 8192 FP32 CUDA cores and 432 Tensor cores[3]. For benchmarking CPU vs. GPU performance under identical conditions, we use the exact same set of hyperparameters on both setups, with the CPU configuration consisting of five cores and 64GB of memory. During full model training, each step took about 0.1 seconds, utilizing mixed precision.

3.3 AdamW Optimizer

In all the experiments, we use the AdamW optimizer [LH19] with $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\epsilon = 10^{-8}$. This section elaborates the core differences between the Adam optimizer [KB17] used in the original Transformer architecture and AdamW.
Both, in Adam and AdamW, Equation (5) shows that the learning rate is adjusted for each parameter independently based on the history of gradients. The running averages, m_{t-1} and v_{t-1}, make it possible to include the history of the gradients in the calculation of the first and second moment:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{5}$$

The calculation of the first and second moment in this fashion ensures that parameters with larger gradient variances are updated more slowly than those with larger gradient variances to stabilize the optimization process.
The bias correction from Equation (6) is important because, without it, the first and second moments are biased toward zero at early time steps, because m_0 and v_0 are zero. Consequently, this results in overly careful parameter updates in

[2]`https://www.statmt.org/wmt17/translation-task.html`

[3]`https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`

the beginning, which hinder the performance and convergence of the training process.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{6}$$

In the original Adam, weight decay is added directly to the gradient. Consequently, this means that the weight decay term is included in the moment estimates (m_t and v_t). The AdamW optimizer circumvents this problem: The weight decay is applied directly to the weights after the gradient update, as shown in Equation (7):

$$\theta_t \leftarrow \theta_t - \eta\lambda\theta_t \tag{7}$$

Equation (8) shows the complete parameter update for the AdamW optimizer, where the weight decay is decoupled from the gradient calculation.

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right) - \eta\lambda\theta_t \tag{8}$$

3.4 Estimation of Memory Requirements

Estimating memory requirements involves accounting for not only the model parameters but also the optimizer states, activation storage during the forward and backward passes for proper gradient computation, and additional buffers used in matrix operations, softmax computations, etc.
PyTorch uses `float32` precision by default, meaning it uses 4 bytes to represent a single parameter. The full Transformer model comprises $\sim$ 95 million parameters, resulting in $\sim$ 360MB of storage. However, that does not account for the optimizer states, which contribute two extra values (first and second moment) per parameter (excluding bias terms), effectively tripling the memory requirements. On top of that, the activations during the forward pass have to be considered, as well as the gradients for the backward pass, each contributing tensors of size $(256, 64, 512)$ for each layer.
For our model, this results in an approximate memory requirement of 6GB, which is consistent with the memory usage logs from training. The tensor size is heavily sensible to the batch size, which could be empirically validated during training; a run with a larger batch size would crash with an out-of-memory error, even though theoretically, our GPUs have 40-80GB of VRAM, an issue to be further investigated.

4 Results

This section provides the results of the CPU vs. GPU comparison, as well as the performance of the Transformer model on the translation task.

4.1 GPU versus CPU Training

Table 1 illustrates the results of the performance comparison between CPU and GPU training. The GPU significantly accelerates the overall training process by a factor of 8.07. Accordingly, the GPU processes a single epoch, as well as a forward pass, faster by approximately the same margin. The GPU achieves the most significant speed-up (20x) during the backward pass. This highlights the GPU's superior efficiency in handling computationally expensive tasks, such as gradient computation, which heavily rely on parallelized matrix calculations. Surprisingly, the GPU uses 38 times less memory per epoch compared to the CPU, which can mainly be attributed to the small batch size of 32 for both setups.
Due to approaching deadlines and long queues on the high performance cluster (HPC), I postpone further optimizations of the GPU codebase. For instance, offloading BLEU score calculations to the CPU, combined with mixed-precision training and other optimizations, can approximately halve overall training time– a speedup that has been observed in practical 11.

While the performance benefits speak for themselves, GPU training entails some disadvantages. Working on the HPC, I experienced long queues before the jobs could start, as well as complexity in setting up the code and accompanying libraries to run on the GPU nodes. This results in longer feedback cycles because each attempt requires submitting a new job and waiting. If errors occur, they only got discovered after the job queue processed the experiment, further extending debugging time. Additionally, estimating the memory resources for optimizing throughput was time-consuming and often required trial and error, especially without easy access to a GPU for testing.

Metric	**CPU**	**GPU**	**CPU to GPU Ratio**
Training time (hours)	3.0017	0.3717	8.07
Avg. Epoch times (seconds)	2161.24	267.63	8.07
Avg. Forward pass times (seconds)	0.0770	0.0088	8.75
Avg. Backward pass time (seconds)	0.2020	0.0101	20
Avg. Single step time (seconds)	0.2790	0.0189	14.76
Allocated memory per epoch (MB)	32998.0	865.91	38.1

Table 1: Performance comparison of CPU vs. GPU

4.2 Machine Translation

Figure 2 shows that on the WMT German-to-English translation task, the Transformer model reaches a BLEU score of around 0.34 with a length ratio of 0.99 on the validation set, which implies solid, coherent translations. Training took approximately 11.5 hours.

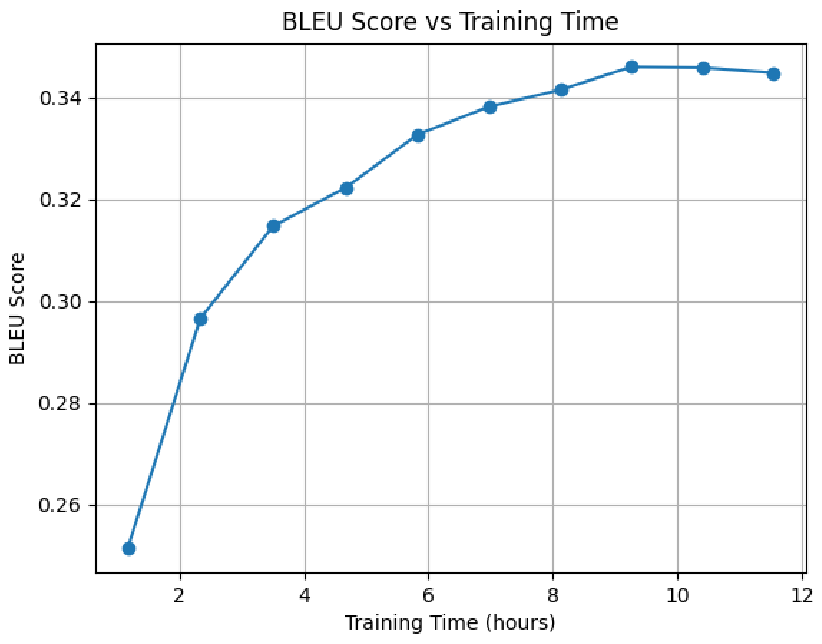

Figure 2: The BLEU score is calculated on the validation set after each epoch. It approaches a value of around 0.34.

Word count: 2282

A Learning Rate Schedule

Figure 3 shows the learning rate schedule during training.

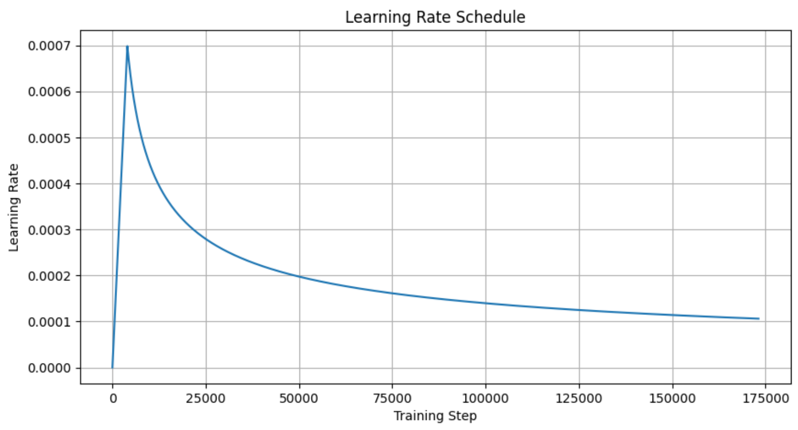

Figure 3: The learning rate schedule with a warmup phase of 4000 steps.

References

[BGLL17] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures, 2017.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[LH19] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[PW17] Ofir Press and Lior Wolf. Using the output embedding to improve language models, 2017.

[Vas17] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.