Implementing Transformer Models
Project Report

Nils Reck

January 17, 2025

1 Introduction

The Transformer model [?] 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 methodolgy section, explaining the modular components of a Transformer.

2 Methodology

The Transformer mainly consists of two principal components, the encoder and decoder. While both process and transform input data, the encoder focuses on generating a context-rich representation of the input sequence, whereas the decoder uses this representation to generate the target sequence step by step. However, before the model can process the input data, it 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.

add info about alignment of sequences (maybe add to training section)

2.1 Embedding Layers

The embedding layer creates a d_{model}-dimensional vector representation for each encoded token of the input and target sequence. 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 positional information. Cosistent with the original Transfromer architecture, we apply parameter sharing by using the same set of weights for both embedding layers and the pre-softmax linear transformation. This technique has shown to improve efficiency and model performance [PW17]. Sharing parameters between the encoder and decoder embedding layers offers several advantages. First, it can significantly reduce the model size while maintaining model performance. Second, parmeter sharing reduces the degrees of freedom of the model, thus implicitly applying regularization by forcing different parts of the model to use the same parameters, preventing the model from overfitting. Additionally, the efficiency of the model improves because shared parameters allow for faster updates and fewer memory operations. Finally, by tying the input and output embeddings together, the model can enhance cross-lingual transfer learning, as aligned word representations across languages make it easier to generalize.

How does the model differentiate between embedding and position?

2.2 Encoder Stack

The encoder consists of six identical layers, each designed to transform the input sequence into a context-rich representation. Each layer comprises two sub-layers: a multi-head self-attention mechanism and a position-wise feed-forward network, each followed by a residual connection [HZRS15] and layer normalization [BKH16] 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, alleviating 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.

Additionally, by focussing on small residuals $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$, the network increases the chance of generalizing better to unseen data, reducing overfitting.

why?

2.3 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. In contrast to the multi-head self-attention layer in the encoder, the inputs to the attention mechanism in the decoder are masked such that the decoder cannot attend to future tokens.

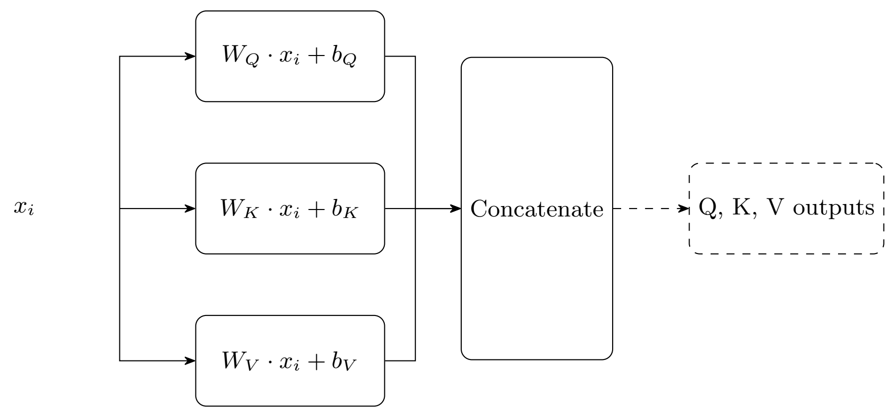
This prevents the decoder from cheating by attending to tokens it has not yet seen. Finally, the output of the decoder undergoes a linear transformation. After that, softmax is applied to convert the output into probabilities to predict the next token.

2.4 Attention

Explain how multiple atttention heads help

It expands the model's ability to focus on different positions. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the actual word itself. If we're translating a sentence like "The animal didn't cross the street because it was too tired", it would be useful to know which word "it" refers to.

2.5 Position-wise Feed-Forward Layer



Three parallel linear transformations

3 Optimization Techniques

3.1 Learning Rate Scheduler

3.2 Optimizer

4 Training

In the backward pass, if you have multiple rank-deficient matrices, your rank becomes even lower. because the composition of rank-deficient matrices leads to a further reduction in the rank, potentially causing the gradients to vanish or lose critical information needed for effective weight updates.

5 Results

1. Make sure you understand the embeddings of the input. Explain why we need the position of input characters in the embedding.

2. Make sure you understand the role of the two different masks in the attention mechanism. Explain the role of each mask in your own words.

3. The model starts with a Query (Q) for the current position. For example, if the model predicts the next token for the word "cat", the Query is derived from the representation of "cat".

6 Questions

6.1 Practical 4

1. Which dimension do the word embeddings need to have?

2. Why do values have a different dimension d_v, compared to queries and keys d_k, wrt the linear projection in mha?

3. How does the model differentiate between embedding and positional encoding?

6.2 Practical 5

1. Why is it called encoder/decoder?

1. Do we put the tokenizer in the TranslationDataset class?

2. What is the word embedding dimension?

7 Positional Encoding

8 Encoder

1. What is the input to where the multiplication with the qkv-matrix happens? A: The embedding matrix

2. What does the fully connected layer look like?

9 Position-Wise Feed-Forward Networks

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{1}$$

The FNN introduces a higher-dimensional space to explore combinations of features present in the token embeddings that it could not explore in the original embedding space. That happens by the first linear transformation $xW_1 + b_1$. Next, ReLU, $\max(0, xW_1 + b_1)$ introduces non-linearity (why does that help?) and helps prevent vanishing gradients (how?). The FFN has two linear layers of size $(d_{\text{model}}, d_{\text{ffn}})$ and $(d_{\text{ffn}}, d_{\text{model}})$, respecetively. Finally, the non-linearly transformed representation is projected back into the original space, d_{model}, such that it (what is it?) is forced to focus on the most significant feature combinations (bring examples).

10 Normalization Layer

Layer normalization is applied after each self-attention and feed-forward sublayer.

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta \tag{2}$$

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:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad\qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \tag{3}$$

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.

Cite Layer Normalization paper

11 Optimizer Initialization

11.1 `AdamW`

Both, in Adam and AdamW, Equation (4) 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:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{4}$$

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 (5) is important because, without it, the first and second moments are biased toward zero at early timesteps, because m_0 and v_0 are zero. Consequently, this results in overly careful parameter updates in the beginning, which hinder the performance and convergence of the training process.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{5}$$

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 adaptive gradient update, as shown in Equation (6):

$$\theta_t \leftarrow \theta_t - \eta \lambda \theta_t \tag{6}$$

Equation (7) shows the complete parameter update for the AdamW optimizer, where the weight decay is decoupled from the gradient calculation.

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right) - \eta \lambda \theta_t \tag{7}$$

Where do we apply dropout?

What are learnable parameters in a transformer model?

Inclued questions from tests

References

[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.

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