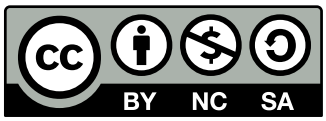
Aalto University
School of Science

Master's Programme in Computer, Communication and Information Sciences

# How well can Machine Learning teach Humans about Machine Learning ?

**Tommi Kulokoski**

Master's Thesis
2024



Aalto University
School of Science

**Author** Tommi Kulokoski

**Title** How well can Machine Learning teach Humans about Machine Learning ?

**Degree programme** Computer, Communication and Information Sciences

**Major** Machine Learning, Data Science and Artificial Intelligence

**Supervisor** Prof. Alexander Jung

**Advisor** Prof. Alexander Jung

**Date** 4 February 2024 **Number of pages** 61+23 **Language** English

**Abstract**

The goal of this paper was to examine to what extent GPT-4 based ChatGPT could be utilized to learn about- and implement a transformer model meant for French to English machine translation. To achieve this goal, ChatGPT was prompted for answers regarding ways to progress the creation of the model.

The evaluation is based on a combination of analyzing the interactions with the chatbot and the performance of the created model. Results revealed that ChatGPT in solitude has serious flaws, but when used in combination with a supporting reference, these flaws can be alleviated, making it a useful tool.

**Keywords** GPT-4, ChatGPT, Machine translation, Deep learning, Natural language processing

Aalto-yliopisto
Perustieteiden
korkeakoulu

**Tekijä** Tommi Kulokoski

**Työn nimi** Kuinka hyvin koneoppiminen voi opettaa ihmisille koneoppimisesta?

**Koulutusohjelma** Tietotekniikka

**Pääaine** Machine Learning, Data Science and Artificial Intelligence

**Työn valvoja** Prof. Alexander Jung

**Työn ohjaaja** Prof. Alexander Jung

**Päivämäärä** 4.2.2024 **Sivumäärä** 61+23 **Kieli** englanti

**Tiivistelmä**

Tämän tutkimuksen tavoitteena oli selvittää, missä määrin GPT-4-pohjaista ChatGPT:tä voitaisiin käyttää oppimaan ja toteuttamaan transformer-malli, joka on tarkoitettu ranskasta englantiin kääntämiseen. Tavoitteen saavuttamiseksi ChatGPT:ltä pyydettiin vastauksia mallin luomisen edistämiseksi.

Arviointi perustuu chatbotin kanssa käytyjen vuorovaikutusten analysointiin ja luodun mallin suorituskyvyn arviointiin. Tulokset paljastivat, että yksinään käytettynä ChatGPT:llä on vakavia puutteita, mutta kun sitä käytetään yhdessä tukiviitteiden kanssa, nämä puutteet voidaan lievittää, mikä tekee siitä hyödyllisen työkalun.

**Avainsanat** GPT-4, ChatGPT, Konekääntäminen, Syväoppiminen, Luonnollisen kielen käsittely

# Preface

I want to thank Professor Alex Jung for his guidance throughout the creation of this paper.

Myllypuro, 19 December 2023

Tommi K. T. Kulokoski

# Contents

# Abbreviations

| | |
|---|---|
| AI | Artificial intelligence |
| BILSTM | Bidirectional long short-term memory |
| BOW | Bag-of-words |
| BPTT | Backpropagation through time |
| DL | Deep learning |
| DNN | Deep neural network |
| EM | Expectation-maximization |
| FFN | Feed-forward network |
| GPT | Generative pre-trained transformer |
| GRU | Gated recurrent unit |
| HMM | Hidden Markov model |
| LLM | Large language model |
| LSTM | Long short-term memory |
| MHA | Multi-head attention |
| ML | Machine learning |
| MMHA | Masked multi-head attention |
| MMI | Maximum mutual information |
| MT | Machine translation |
| NLP | Natural language processing |
| NN | Neural network |
| PCA | Principal component analysis |
| RNN | Recurrent neural network |
| SP | SentencePiece |
| SVM | Support vector machine |
| TF | TensorFlow |
| TF-IDF | Term frequency-inverse document frequency |

# 1 Introduction

Public interest towards artificial intelligence (AI) and machine learning (ML) has rapidly increased in the past few years. Among the most prominent AI applications is ChatGPT, a generative pre-trained transformer (GPT) model based chatbot developed by OpenAI. In the past year it has gained a lot of attention, as can be determined from the fact that it's the most viewed English Wikipedia article of 2023 [1, 2] and also from Google trends depicted in Fig. 1 [3].

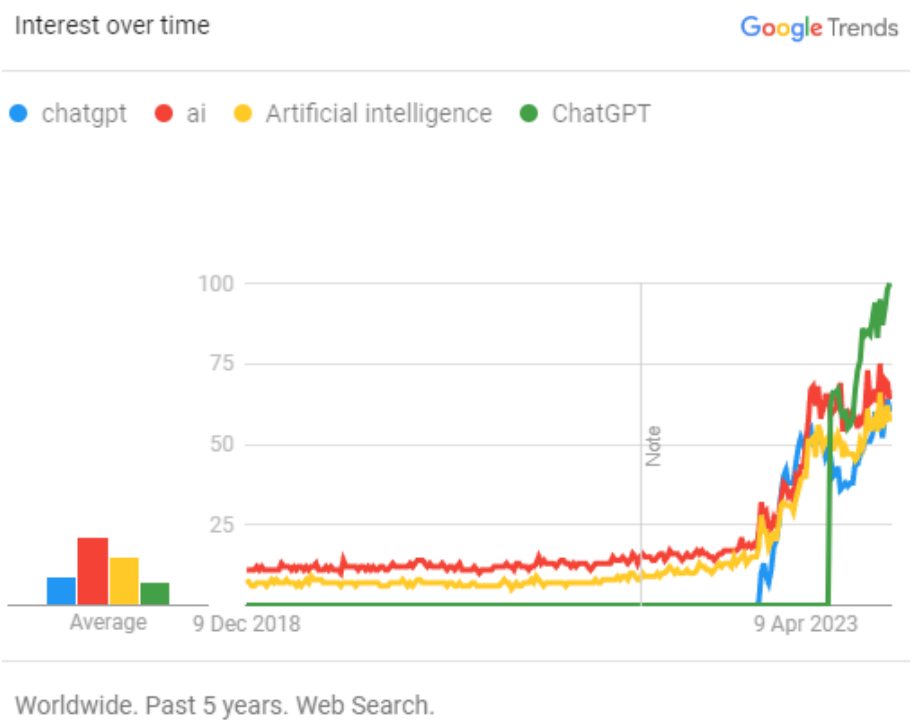



**Figure 1:** Shown are Google trends of the topics chatgpt (blue), ai (red), Artificial intelligence (yellow) and ChatGPT (green) from Dec. 2018 to Dec. 2023. Interest in ChatGPT has greatly risen since the beginning of 2023.

The aim of this thesis is to gauge to what extent the newest model for ChatGPT, based on GPT-4, can be used to implement a machine translation transformer model. The purpose for this is to help understand how ChatGPT can be used to learn and teach deep learning (DL) concepts, as it has not been explored before. More specifically, ChatGPT will be prompted to help in constructing, training and evaluating a transformer model, specifically for machine translation from French to English.
There are two key elements to evaluate how well GPT-4 performs with respect to the task at hand. First there is the evaluation of answers it provides based on different metrics, such as their quality, their correctness, etc. The second part is the evaluation of the resulting model. A combination of these two evaluations will be used to determine the strengths and weaknesses observed during the process.
This thesis will mainly attempt to judge the suitability of GPT-4 as a teacher in the

area of DL. Producing the best model possible is not the goal, but rather creating a model that fulfills its task well enough.
The related theory is mainly about the history of the transformer model, GPT-4, applications and use cases of GPT-4 and similar experiments involving GPT-4.
After introducing the related theory, the experiment itself will be explained. In this section the practices for this project, along with the used resources, will be listed. This section will provide an overview of the project by explaining it step by step and providing insights for certain choices made along the way.
The next section will entail the results of the project. It is a mix of qualitative analysis of interactions with the chatbot and quantitative results gained from both - the conversations with GPT-4 and the created model.
Finally, the last section will discuss and analyze the results from the previous section and summarize the findings of the experiment with respect to the initial goals.
As the topics of this thesis have not yet been explored, it serves as groundwork for potential future work on the subject of using GPT-4 for similar tasks. Because the evaluation is partly qualitative analysis of perceived metrics from answers, there will obviously be human bias that should be taken into account when looking at the conclusions of the project. At the same time, the results should serve to some extent as a benchmark together with the quantitative results from the conversations and the performance of the created model.

# 2 Background

This section is dedicated to the related work of the project. Sec. 2.1 contains a brief explanation of what the concept of natural language processing (NLP) entails, as the project is dedicated mainly to the domain of NLP. Sec. 2.2 goes on to set the historical context for the technologies used, to set the current stage of language models. Sec. 2.3 presents the core method, which is being used in the form of ChatGPT and is also the architecture being implemented in the project. As the transformer is dealt with in almost every section, the architecture is presented for the reader as a reference. Sec. 2.4 describes a general way of how an ML process usually looks like. It is intended as a reference to compare the implemented code to on a higher level. The final Sec. 2.5 is about other relevant information concerning ChatGPT and the GPT-models, such as how they are trained, what similar works they have been used for and what kinds of limitations and concerns there are regarding the models.

## 2.1 Natural language processing

Natural language processing (NLP) studies how computers can be used to understand natural languages, with the purpose of performing relevant tasks. The field is a combination of linguistics, computer science, cognitive science and AI. Some of its main branches are machine translation, sentiment analysis, speech recognition and natural language generation and -summarization. [4]

## 2.2 History of language modeling

This section is dedicated to setting the transformer model and the mechanisms it applies into historical context. It is intended to show why language models today are the way they are and what their current state is like.

### 2.2.1 Symbolic rules

Humans have been trying to master language intelligence ever since the Turing test was proposed by Alan Turing in 1950 [5, 6]. The first NLP methods used to attempt this task were rule-based systems, up until the 1980s [4]. A notable example is the Georgetown-IBM experiment conducted in 1954 [7], where more than 60 English sentences were translated into Russian [4]. It was accomplished using a ruleset containing 6 rules and 250 lexical items [8], composed of stems and endings. Albeit the experiment was relatively simplistic compared to today's standards, it served as an important milestone for machine translation and sparked a lot of research in the field. [9]
The advantages of rule-based systems are their transparency, the level of control, the interpretability and the fact that they don't require large amounts of data to function. As for their disadvantages, they can be rigid, they have poor scalability and updates to the rules might require exponential expert labor. Changing or adding rules might

affect previous rules and require for them to also be changed. It is also important to note that they do not use context. [4, 9]

### 2.2.2 Statistical models

Data-driven methods began to be prioritized in the 1980s and 1990s [5, 9, 10] due to the increase in computational power and machine-readable-data availability. The former structural and knowledge-based methods were deemed inferior to the discriminative models, as the former were lacking the uncertainty handling and generalization ability, which helped the latter learn better [4]. Due to the switch to data-driven methods, the time period is also referred to as a time of empirical methods.
Information retrieval methods, such as the bag-of-words (BOW) from 1954 [11], n-gram from the 1950s [12] and term frequency-inverse document frequency (tf-idf) from the 1970s [13], became increasingly important with the popularity of data-driven methods. Of these, the BOW model simply counts the occurrences of words, while disregarding order; the n-gram model is used to predict the next letter, syllable or word in a sequence, while only considering the previous n-1 letters, syllables or words; the tf-idf model is used to measure the importance of a term for a document, by accounting for how often a term occurs in a document and in how many other documents the same term is present. The n-gram can be considered an extension of the BOW, being able to capture some context and order.
Other popular statistical methods used during the 90s include hidden Markov models (HMM) [14], expectation-maximization (EM) [15], maximum mutual information (MMI) [16], principal component analysis (PCA) [17] and Bayesian networks [18]. Additionally, some shallow ML algorithms and techniques used were support vector machines (SVM) [19], the perceptron [20] and for neural networks (NN) the back-propagation algorithm [21]. [4]
While there was a lot of improvement compared to symbolic rules, the results weren't close to human level. One of the problems was the simplicity of most shallow ML models, which made them unable to properly utilize large amounts of training data. Another one was the exponential cost for modelling higher-order language models. To add to this, as features were crafted manually using domain expertise, it added to the difficulty of covering all regularities. Lastly, sparsity also plagued models, as features only occurred once in the training data. All these factors lead to feature design becoming a central problem for statistical NLP.

### 2.2.3 Deep learning

A possible solution to the feature engineering was addressed with "Natural language processing (almost) from scratch" [22], which suggested using DL networks with multiple hidden layers. The advantage of the multiple layers of non-linear units is the formation of higher level features from lower level ones, through the hierarchical nature of the network. This way, training data could be properly utilized to extract more intricate information.

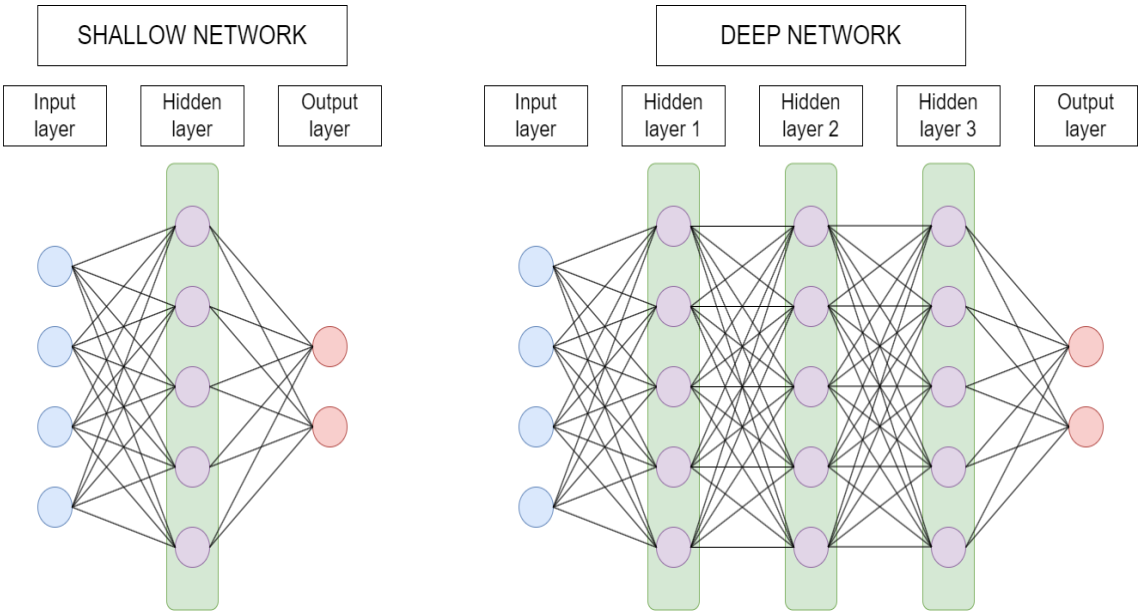



**Figure 2:** The shallow network consists of only one hidden layer, while the deep network consists of 3 separate hidden layers.

Fig. 2 shows the general difference in architecture between a shallow and a deep network. The shallow network consists of only one hidden layer in addition to the input- and output layer. In contrast, the deep network is built of 3 hidden layers in addition to the input- and output layers. The hierarchy of multiple layers can form abstractions of higher levels from features than a shallow one can, which makes it the preferred architecture to extract deeper patterns.

The backward propagation of errors, or simply "backpropagation", is a fundamental mechanism of neural networks. First it calculates the loss of the network output, that is, grossly simplified the difference between the actual target and the prediction. It is followed by the propagation of the error back through the network, to determine how much each neuron contributed to the total loss. The knowledge can then be used to tune the weights accordingly, as to create a better performing model.

Already in the early days of the backpropagation algorithm (1990s), it faced a problem with the signal vanishing. A vanishing signal, also called a vanishing gradient, is the shrinking of the gradients caused by passing them through multiple layers in a network. The main causes to this were a lack of training data, proper design and learning methods. The number of layers increased the issue exponentially, especially in recurrent neural networks, which will be talked about soon. The main issue with the vanishing signal is that it makes the tuning of weights in the network difficult.

An initial overcoming of the problem was to use unsupervised pre-training to learn features that are useful in general [23]. The network is then trained using supervised classification. Furthermore, at Microsoft it was discovered that a large corpus of training data, combined with a deep neural network (DNN) designed with correspondingly large context-dependent output layers and careful engineering lead to far lower errors than with state-of-the-art shallow ML models [24].

Feature engineering also made large progress with neural embeddings, which are lower-dimensional representations of data learned by a NN. Although already proposed in the early 2000s [25], the increase in data availability and computational capabilities in 2013 proved their usefulness [26] in the form of word2vec [27]. Word2vec, developed at Google, produced denser features with contextual information. Its significance was due to its computational efficiency and its capability to capture a large amount of semantic and syntactic relationships between words with ease from unstructured data [4]. These capabilities made word2vec widely used in many areas of NLP. The remaining problem was, however, that the embeddings are static and assign the same vector to a word in every context. Words with multiple meanings or nuances in different settings could thus not be properly described.

Sequence models, such as recurrent neural networks (RNN) are good for handling context. RNN are one of the earliest NN models able to handle sequential data, such as text or time series. The rough concept is shown in 3. The RNN handles sequential data in time steps, while maintaining its internal states. That way, backpropagation is basically backpropagation through time, as errors are propagated through earlier states. Outputs thereby are affected by the previous inputs and the state of the combined hidden layer. The model enables capturing temporal dependencies as well as context. Its major shortcoming is that the architecture causes it to suffer from vanishing gradients.

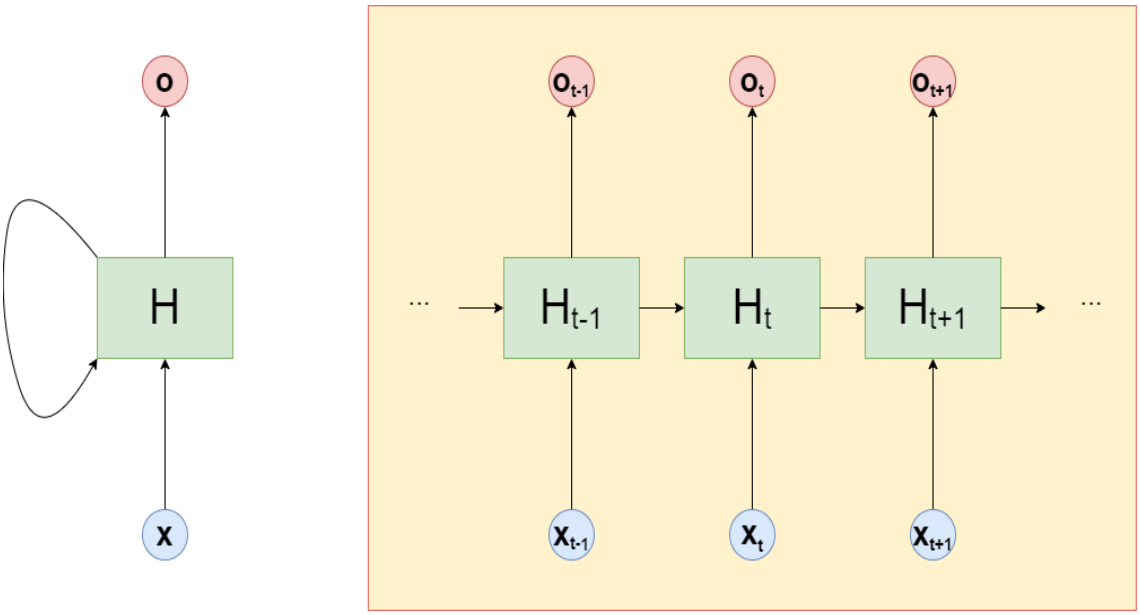



**Figure 3:** The RNN model compactly shown on the left and unfolded on the right. **x** and **o** are the input and the output respectively. Layer H consists of multiple hidden layers. Weights and biases of the hidden layers are shared between states.

To handle the pitfalls of the RNN, long short-term memory (LSTM) [28] units and one of its variations, the gated recurrent unit (GRU) [29], were developed. Both apply gating in their respective architectures to maintain and modulate information

across time steps. These gates selectively retain crucial information and prevent it from diminishing over time. This way important information and gradients are enabled to flow through long sequences without diminishing. Consequently, both gated variants are able to capture long-range dependencies better than basic RNNs.
Another variation of the LSTM is the bidirectional long short-term memory (BiLSTM) [30] unit, which allows the processing of sequences in both directions. The architecture consists of 2 LSTMs running in parallel. It provides richer dependencies, but is in turn less efficient than the GRU.
The encoder-decoder architecture [31] can work with variable length input and output, capturing context in a fixed length context vector in the encoder and passing it into the decoder. It provides better performance in complex tasks and can also be combined with the previously mentioned sequence models, which lead to its widespread adoption in models.

The introduction of the attention mechanism in 2014 [32] is an integral step to the creation of the transformer [33], which will be properly introduced in Sec. 2.3. Attention allows the model to focus on different parts of the input during output generation. Attention weights are used to find the most relevant parts of the input with respect to the output. Due to having the ability to access any part of the input during output generation, the attention mechanism is better at capturing long-range dependencies than LSTMs or any of its variants. One notable reason is the limitation LSTMs have on how long of an input they can properly handle.
Embeddings from Language Models (ELMo) [34] also shifted the NLP landscape. ELMo generated dynamic word embeddings that changed based on the surrounding context, significantly improving the ability of models to understand nuanced language use.

The significant leaps in the past century have led to large language models, such as bidirectional encoder representations from transformers (BERT) [35] and GPT [36], of which the latest model being GPT-4 [37]. Both BERT and the GPTs are built upon the transformer model and pre-trained on a large corpus. While BERT focuses on the encoder using bidirectional training, GPT focuses on the decoder part of the architecture. Having been trained using a language modeling objective, the first GPT was able to generate coherent and contextually relevant text.

## 2.3 Transformer architecture

This section is dedicated to presenting the transformer architecture proposed by Vaswani et al. [33], displayed in Fig. 4.

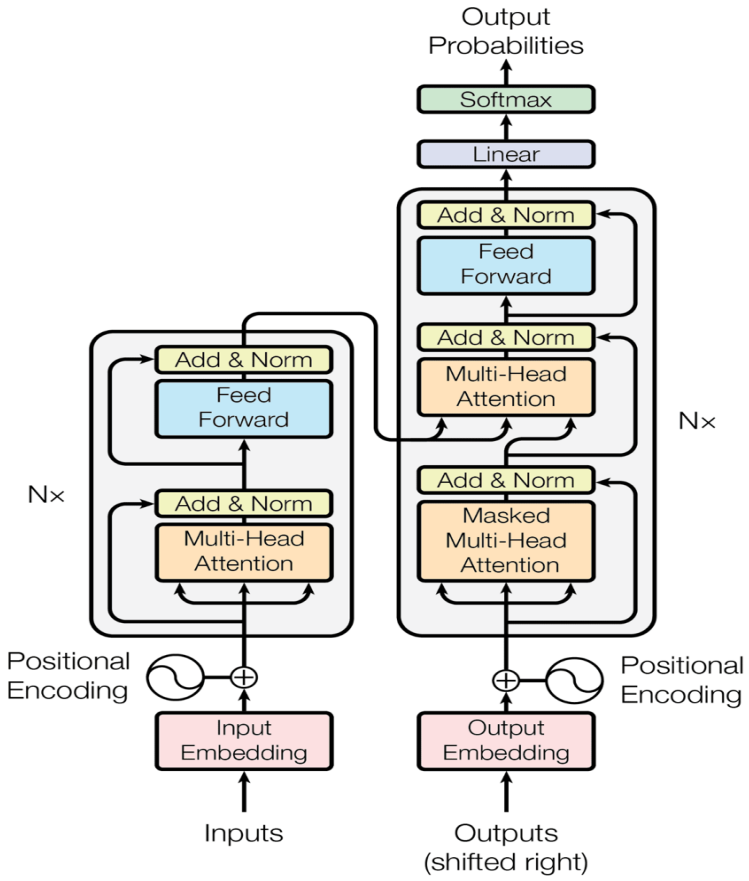



**Figure 4:** The transformer architecture from "Attention is all you need" [33]

### 2.3.1 Embedding and positional encoding

Before the embedding, the input and output sequences need to be tokenized. The act of tokenization is the transformation of sequences into inputs that the transformer can handle, for instance changing words into representative integer numbers. An additional requirement is the use of start of sequence (SOS) and end of sequence (EOS) tokens, which are respectively put in the first- and last positions of the sequences. Tab. 1 shows an example of a possible tokenization for a sentence.
Once tokenized, the output needs to be shifted to the right and masked with a look ahead mask. The right shift is used with the intention of using the current input token to predict the next output token. The look ahead mask is used to only show preceding tokens to the model, as not to get help from future tokens.

| Words | SOS | I | am | a | cat | and | a | horse | . | EOS |
|---|---|---|---|---|---|---|---|---|---|---|
| Tokens | 1 | 14 | 563 | 225 | 54 | 37 | 225 | 662 | 3 | 2 |

**Table 1:** An example of word tokenization, where SOS is encoded as 1, 'I' is encoded as 14, etc.

The embedding of both input and output works the same way. Tokens are represented with a high dimensional fixed size vector, which captures semantic and syntactic information about the corresponding token. The fixed size ensures the uniform dimensions of tokens, which is required for the processing through layers.

Due to the nature of transformers, they do not process sequential data in order. To be aware of the relative or absolute positions of words in a sentence, positional information needs to be added to the embeddings. Positional information is added using positional encodings, which are of the same dimensions as the embeddings, for the purpose of summation. In the original paper [33], a sine- and cosine function with different frequencies was used:

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{1}$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{2}$$

In Eq. 1 and Eq. 2 $pos$ is the position, $i$ is the current dimension and $d_{model}$ is the dimensionality of the model embeddings. Using the functions produces a unique encoding for each position. The nature of the sine- and cosine functions also helps to attend by relative positions, as for any offset $k$, $PE_{pos+k}$ can be represented as a linear function of $PE_{pos}$.

### 2.3.2 Multi-head attention

The attention mechanism can be generalized as being a mapping of query (Q) and a set of key (K) and value (V) pairs to an output, where each of them consists of vectors. Outputs are calculated as a weighted sum of V, where the weights are calculated by some compatibility function of Q with its corresponding K. In simple terms, attention is telling what to pay attention to wrt. the input.

For the original transformer model [33] the used attention mechanism is the scaled dot-product attention shown in Fig. 5. It can be described by the following equation:

$$\text{Attention}(Q, K, V) = softmax(\frac{QK}{\sqrt{d_k}})V \tag{3}$$

Here Q and K are of dimension $d_k$ and V is of dimension $d_v$. It is the dot-product of Q and K, scaled by $\sqrt{d_k}$ and with softmax applied to get the weights for V. The softmax function is a mathematical function, which converts raw scores into probabilities.

The entire attention mechanism block in the transformer is called the multi-head attention (mha) mechanism, because it utilizes $h$ attention heads. The amount of heads $h$ determines the amount of differently learned projections of Q, K and V, with dimensions $d_k$, $d_k$ and $d_v$. Each of the different projections are passed through the

scaled dot-product attention in parallel with a $d_v$-dimensional output; concatenated and projected to the final weighted V.

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= concatenate(head_1, ..., head_h)W^O, \\ head_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{4}$$

Here the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

Multi-head attention allows for each head to learn from and focus on a different aspect of the information provided. Using the heads combined provides deeper insights. The described architecture can be seen in Fig. 5.

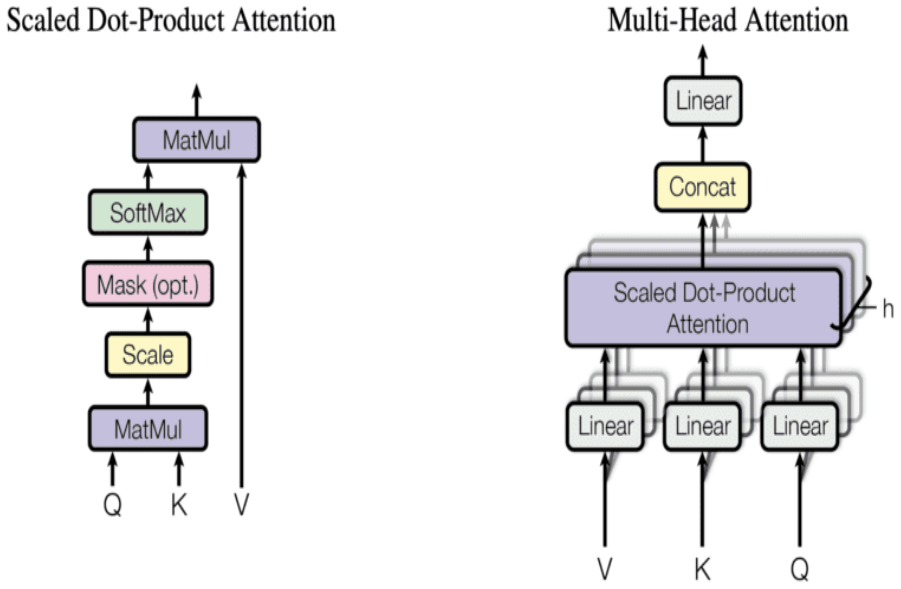



**Figure 5:** Scaled dot-product and multi-head attention from "Attention is all you need" [33]

In the model, there are three different multi-head attention layers, of which two are very similar. The similar ones are both self-attention mechanisms. Q, K and V come from the same source in self-attention. One of them is in the encoder. It takes its inputs from the previous encoder layer and has access to all the positions in the previous layer. The other self-attention mechanism is the masked multi-head attention in the decoder. It differs in that it only has access to previous positions up to and including the current position. It cannot access 'future' tokens. Masking is used to hide future positions from the current decoder. The purpose is to keep the autoregressive properties, in other words, to let only previous tokens affect the next prediction.

The final attention layer type applies cross-attention. It uses the Q and K from the encoder output and the V from the previous decoder layer. The encoder output allows the decoder to attend to all the input positions.

### 2.3.3 Point-wise feed-forward network

Each encoder- and decoder layer contains a fully connected feed-forward network, which consists of two linear transformations with a ReLU activation between them. A feed-forward network (ffn) layer is described as:

$$\text{FFN}(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{5}$$

Here $x$ is the input, $W_1$ and $W_2$, and $b_1$ and $b_2$ are the weights and biases for each linear transformation layer respectively. The input and output are both of dimension $d_{model}$.

### 2.3.4 Residual connections, layer normalization and dropout

Residual connections use the input signal before a layer and add it to the output of the layer. That is why another name for them is skip connections. The method is used to alleviate the vanishing gradient problem, by letting gradients flow more freely. By lessening the burden of vanishing gradients, deeper models can be trained. It also encourages layers to learn modifications to the identity function compared to just having to learn the signal anew at each point, which leads to more efficient training.

Layer normalization, as the name states, normalizes the inputs of a layer. The result is a more stable distribution of values across the layer. Advantages are faster learning, reduction of sensitivity towards initial weights and access to use of higher learning rates.

Dropout is a concept used for regularization. By dropping features/neurons randomly at some determined rate during training, dropout prevents the model from relying too much on any singular features/neurons. Being used to reduce overfitting, dropout in the transformer model is used after each sub-layer, but before the residual connection and layer normalization.

### 2.3.5 Encoder-Decoder

The main architecture consists of n layers of encoders and decoders, as depicted in Fig. 4. Both encoder and decoder use a self-attention sub-layer and a feed-forward neural network, but the decoder additionally utilizes a cross-attention sub-layer.

Inputs of the encoder are first passed through a multi-head self-attention mechanism. After that, they are passed through a feed-forward network. Around both of these sub-layers, residual connections are used, which is followed by layer normalization. The decoder layer can be described as:

$$\begin{aligned} x_1 &= (\text{layernorm}(x_0 + \text{mha}(x_0))) \\ \text{out}_{enc} &= (\text{layernorm}(x_1 + \text{ffn}(x_1))) \end{aligned} \tag{6}$$

Here $x_0$ is the initial input into the encoder, $mha$ is the multi-head attention mechanism, $x_1$ is the result after the applying of the residual connection and the layer

normalization on the first sub-layer, ffn is the feed-forward network and $\text{out}_{enc}$ is the output of the encoder-layer.

The decoder applies a similar first step on the output. It is passed through multi-head attention similarly, with a difference that it applies a look ahead mask, as described at the beginning of Sec. 2.3.1. The sub-layer that follows is the cross-attention, which uses the output of the encoder as its query (Q) and key (K) in combination with the target (V) that has been passed through the masked mha. The result of this step is passed through a feed-forward network and passed outside the decoder. Residual connections and layer normalizations are - just like in the encoder - applied between every sub-layer. The decoder can be described in a manner similar to the encoder:

$$\begin{aligned} y_1 &= (\text{layernorm}(y_0 + \text{mmha}(y_0))) \\ y_2 &= (\text{layernorm}(y_1 + \text{mha}(\text{out}_{enc}, y_1))) \\ \text{out}_{dec} &= (\text{layernorm}(y_2 + \text{ffn}(y_2))) \end{aligned} \tag{7}$$

Here $y_0$ is the initial target passed to the decoder; mmha is the masked multi-head attention; $y_1$ is the target after going through the mmha; $\text{out}_{enc}$, mha and ffn are the same as in Eq. 6, with mha and ffn being the decoder counterparts; $y_2$ is the output after the mha and $\text{out}_{dec}$ is the output of the decoder layer.

After going through every layer, the final output is passed through a linear layer and a softmax function to produce the output/next-token probabilities.

## 2.4 Typical ML process

One way to describe the general ML process is depicted in Fig. 6. It starts by acquiring data, either by collecting it or by finding some readily available data. The raw data usually needs to be prepared to be of use. Some common practices are the cleaning of data, analyzing the data and splitting into sets for training, validation and testing. The data cleaning part can entail multiple different aspects, such as handling missing values, choosing relevant features, correcting errors, standardizing formats and removing duplicates, to name a few.
After the data has been deemed usable and is split into training, validation and test sets, the next step is to choose the best model to handle the task at hand. Models are usually compared to get a feel for which performs the best or performs the task in the most desirable manner. The comparison can result in a change of models early on, or even after completely testing a model.
To train the model, only training data is used. After initial training of the chosen model, it is validated using validation data. The reason being that it needs to be validated on data it hasn't seen before to see, whether it overfits or not. Overfitting would mean that the model learns a representation very fitting to the training data, but that it wouldn't generalize well. Using the validation results, the model parameters are tuned, and the model is retrained, until a satisfactory validation result is achieved.
The final step, in a sense, of the process is the testing of the satisfactory model. The

testing is necessary, as the tuning done during validation might have been tending to the validation set, which would, again, not be desired, as the goal is a model that generalizes well. Based on the test results, the process can either be finished, or it can be improved upon.

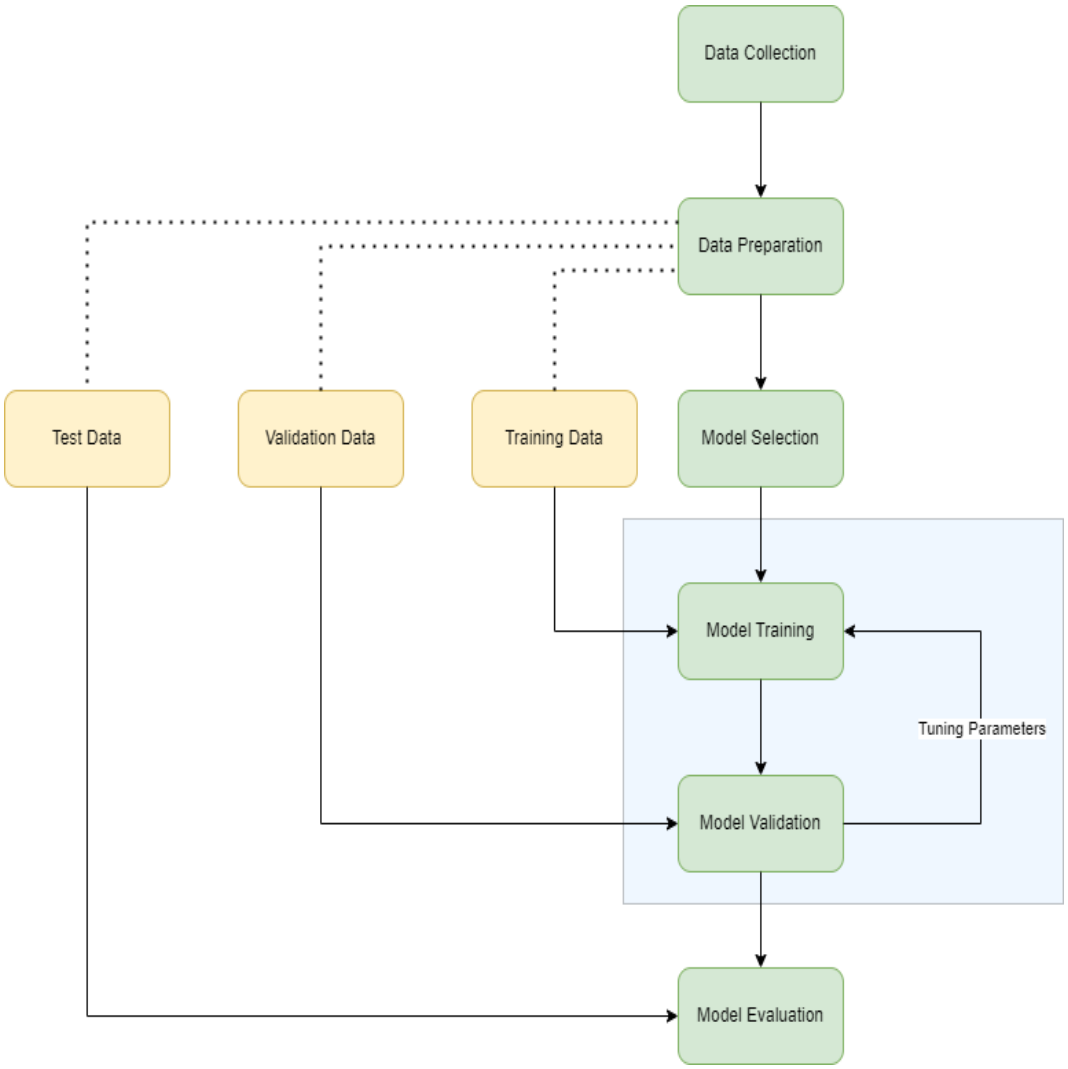



**Figure 6:** The flowchart describes the general ML process. The green boxes indicate the main process, while the yellow boxes indicate the respective part of the dataset that each step of the process uses. The area dyed in blue is an iterative process to find optimal parameters for the model. It utilizes the training set to train the model and the separate validation set to check how well the trained model works on unseen data. Parameters are then tuned as seen necessary to improve the training of the model.

## 2.5 GPT models

GPT models, along with their respective properties, are presented in this section. In addition, similar use cases to this project will be looked into, as well as some concerns

regarding the GPT-models.

### 2.5.1 OpenAI

OpenAI is the developer of GPT-4 and was founded as a non-profit AI research and deployment company in 2015 [38]. According to OpenAI, their mission is to ensure that all of humanity benefits from artificial general intelligence. In 2019 OpenAI opened a capped for-profit subsidiary to finance its operations. The cap is intended to keep the focus on the research, rather than maximizing profits. To further aid this goal, the non-profit part of the company governs and oversees all activity. After the capped for-profit announcement, OpenAI entered a partnership with Microsoft [39].
A key focus for them is safety. Their policy is that as AI is becoming part of everyday life, there is a high need for safety, when it comes to the development and deployment of AI applications. OpenAI employs safety teams for that purpose.
Some of the most well known products by OpenAI include the GPT-models, with ChatGPT standing out in the past year; and Dall-E 3 [40], a modern text-to-image model.

### 2.5.2 GPT

"Improving Language Understanding by Generative Pre-Training" [36] was a pivotal paper in NLP. It introduced the transformer-based GPT model. It was trained using a combination of the "BookCorpus" dataset and a version of English Wikipedia, which roughly estimates to 19.5GB of data. One of the key contributions is that it utilizes a transformer architecture that is pre-trained on a large corpus of unlabeled text data. This encourages learning of a wide range of patterns and structures in language, such as grammar and context.
After the unsupervised pre-training, the model can be fine-tuned to a smaller, task-specific dataset. Such tasks can include classification, question answering and summarization. This is achieved by using the ability of the transformer to handle long-range dependencies efficiently, by using parallel computations.
The approach demonstrates how well large language models (llm) are able to generate and understand natural language. The paper showed promising results over other approaches at the time, particularly in the field of deep understanding of language and semantics.

### 2.5.3 GPT-2

The introduction of the GPT-2 model in "Language Models are Unsupervised Multitask Learners" [41] presented a larger model compared to the base GPT. It has 1.5 billion parameters, which compared to the 117 million parameters of the base GPT was a substantial increase. It was also trained using 10 times the amount of data, compared to GPT.
The dataset that was used for the pre-training is the "WebText" dataset, which is a dataset web scraped by OpenAI. It consists of over 8 million documents.

A central claim is that GPT-2 is able to perform certain some language tasks without task-specific training. This showed that a singular model could generalize to perform different tasks by just changing the prompt.
One of the key aspects of this paper was its emphasis on zero-shot learning, which essentially means the ability to perform tasks, which a model wasn't specifically trained to perform.
GPT-2 was not just an improvement from the base model, but it also outperformed many benchmarks during its release. The paper also touched on ethical concerns that will be elaborated on in Sec. 2.5.8.

### 2.5.4 GPT-3

The introduction of GPT-3 in "Language Models are Few-Shot Learners" [42] once again revealed an increase in model size and in the amount of training data used. The transformer-based llm used roughly 175 billion parameters, which is about 115 times the amount of its predecessor. It is trained using "The Pile" dataset by EleutherAI [43], which is a dataset consisting of 22 smaller datasets, meant specifically for training llms. It consists of English text collected from publicly available, including academic, data. Duplicates of higher quality texts are used to improve training. The size of the dataset is 886.03 GB, compared to the 40 GB of text in WebText.
GPT-3 had an even better few-shot learning ability, as it could perform untrained tasks with just one or two examples. It showed that scaling up the model size lead to better generalization. Just as its predecessor, GPT-3 is trained to be task-agnostic, meaning that it isn't trained to handle any specific task. Among other things, it can generate and translate text, and answer questions.
The new GPT-3, just as GPT-2 did before, surpassed previous models, while also outperforming some state-of-the-art models specifically fine-tuned to some tasks. The paper also talks about ethical concerns and limitations of scaling up the model size.

### 2.5.5 GPT-4

The paper "GPT-4 Technical Report" [37] does not disclose the model size, nor the training set used for GPT-4. It can be assumed, though, that at least the model size is substantially larger than that of GPT-3. The comparisons of known parameter amounts and training set sizes can be seen in Tab. 2.
The paper puts an emphasis on GPT-4 being about building a predictably scaling DL stack. New features, which are not yet available to the public, include the ability to handle image input.
Key improvements of GPT-4 compared to previous models are better multilingual abilities and the ability to take larger inputs of 25000 words and more. It also outperforms the previous versions in most tasks.
A notable achievement is related to the taking of benchmark tests. Reinforcement learning from human feedback (RLHF) is used in ChatGPT, as well as GPT-4 as an additional reward signal, which is supposed to improve the model by converging with human values and expectations. In multiple choice questions the model performed

equally well with and without RLHF, which indicates that GPT-4 didn't need any help, besides its pre-training.
The report also touches on emergent behaviors of the model, which will be discussed in Sec. 2.5.8.

| | GPT | GPT-2 | GPT-3 | GPT-4 |
|---|---|---|---|---|
| Parameters | 117 million | 1.5 billion | 175 billion | unknown |
| Training set size | ≈19.5 GB | 40 GB | 886.03GB | unknown |

**Table 2:** The amount of parameters and the size of the training set for each GPT-model respectively.

### 2.5.6 ChatGPT

ChatGPT is a specialized version of the GPT-models, used for conversational purposes. It is specifically fine-tuned to generate human conversational text. The original version is based on GPT-3. It is specifically trained to handle nuances of conversations, retain context, apply conversational norms and to be coherent. The specialized tuning uses RLHF to improve conversational performance. It's mainly used to filter appropriate and relevant answers. Currently, the newest model version of ChatGPT is based on the GPT-4 model. [44]

### 2.5.7 Using ChatGPT to teach

The capabilities of ChatGPT in education have already been largely explored. Different subjects that have been looked into include programming [45, 46, 47], math [48, 49], medical education [50, 51, 52] and economics [53, 54]. It has also been studied that the performance of ChatGPT differs based on the subject. While law, medical education, math and psychology were in need of improvement, programming and especially economics have had at least satisfactory results.

ChatGPT offers great benefits to support learning. One of them is the personalized- or self-paced learning [46, 55, 56, 57]. It enables a learner to ask questions in a manner fitting to his needs and in his own words. Another advantage is the tutoring and explaining of concepts that ChatGPT provides [46, 57, 58, 59]. It can explain concepts in different ways, which can support a learner to understand. It can also summarize texts and generate questions for practice. Similar things it is capable of are proofreading, grammar checking and rephrasing of sentences [46, 59]. ChatGPT can also be used as a support when coding, debugging or if generated code is required. Answers are also generated quickly, which is a great asset to have.
The aspect of critical thinking and problem-solving when using ChatGPT [59, 60] is controversial, as there are debates concerned about whether it can be used to foster critical thinking or if it is just used to copy answers.

There are also some challenges that arise when talking about the use of ChatGPT for teaching. Lack of citation and reference for one is important, as it's crucial to

know where the information content for the answer is taken from [57]. Due to that it can also produce plausible sounding, but untrue answers [44, 57], which are then difficult to spot. Uncertain queries are also always answered, even if they would need a followup question [44, 57]. The last point isn't as much of a challenge as it is a side note: ChatGPT tends to produce lengthy answers due to the preference of longer answers during training.

There was a large gap in the data science education sector, as there was only one relevant paper, which is "ChatGPT for Teaching and Learning: An Experience from Data Science Education" [61]. It uses ChatGPT with the GPT-3.5 model.
The paper is made about a data science course, where students (n=28) were tasked to use ChatGPT and evaluate their experience on a questionnaire with answer options ranging from 1 (strongly disagree) to 5 (strongly agree). The tasks were focused on 2 key skills used in the data science field: programming and effective problem-solving. The questionnaire consisted of impressions regarding the following 10 themes: programming in general, understanding new concepts, clarifying old concepts, analyzing data without human effort, assistance in critical reasoning, assistance in coding with human effort, assistance in coding/parameter explanation, suggesting learning materials, suggesting libraries or tools, and suggesting skill sets/career path. Out of the themes, coding and (hyper)parameter explanations were the most useful according to the students, while assistance in critical thinking/problem-solving and suggestion of learning materials or skill sets were the most problematic.

### 2.5.8 Limitations and concerns

Some of the more obvious limitations are the amount of data needed to train the models and the ever-increasing need for computational power with ever larger models. Limitations regarding the model itself include the writing of plausible sounding, but untrue answers. The solution isn't simple, as 1) during reinforcement learning there is no source truth, 2) if the model is trained to be more cautious, it will decline questions it answers correctly and 3) supervised training misleads the model, as the ideal answer depends on the model's own knowledge and not what the human demonstrator knows. It is a form of "hallucination", which llms such as the GPT-models suffer from. Hallucinations are factually incorrect, irrelevant or nonsensical answers wrt. the query. They are caused by either the model's limited understanding of complex subjects, biases in training data, or inherent challenges producing coherent human language. The models are also sensitive to small changes in the input and multiple generations of the same query [44].
When it comes to the length of responses, they are more often lengthier than not. Some phrases are also over-represented in answers. Both stem from biases in training data, such as preferences for longer, more thorough answers. They are also affected by over-optimization issues.
Ambiguous queries should require a follow-up question, but are handled by guessing the meaning. Inappropriate requests should be refused, but are sometimes responded to.
One key problem is the difficulty to interpret llms, especially when the model sizes

grow into hundreds of billions of parameters.
An important critique of llms is, that they are sometimes called stochastic parrots [62]. The core idea behind the naming is that under the guise of complexity and apparent understanding, the model merely repeats patterns it has learned, without truly understanding the meaning. This causes ethical concerns, as models could produce harmful outputs or use bad amplified biases to produce an output. The papers have touched on the ethical concerns and talked about impacts of misinformation, impacts on privacy and impacts on professions or social structures [37, 41, 42]. To minimize the impact, models have been released slowly, starting from smaller models. The need for further research has also been stressed as a countermeasure.
The technical report for GPT-4 also mentions possible emerging capabilities of such large llms, such as the ability to make long-term plans outside of the trained data. Power-seeking of the model would be one of such concerns, as it is optimal for most reward functions [37].

# 3 Research material and methods

This section describes the material and methods used to implement the project. Sec. 3.1 presents the initial plan for the project. Sec. 3.2 lists the resources needed and used to complete the project, such as the available hardware, and the applications used to implement the transformer. Sec. 3.3 talks about the metrics that need to be considered specifically with respect to how ChatGPT interacts with the user. Sec. 3.4 in turn is about how the code for the transformer is implemented and how the process of generating the model is evaluated.

## 3.1 Initial plan

The plan is to simulate a situation, where the user has at least basic knowledge of programming and potentially some general knowledge of ML concepts. The task is to implement a transformer model for translation of sentences from French to English by using ChatGPT as a teacher. The person implementing the project does have more than some basic knowledge about ML concepts, but is not very knowledgeable about transformers or translation tasks in general. To add to the challenge, the implementer is not allowed to look at any relevant material about transformers beforehand. Additionally, the implementer is not supposed to get used to ChatGPT in advance, as to see how well it handles a new user and how well a new user in turn can handle it.
As the implementer has previous experience in the DL framework PyTorch, the instructions will be asked for in another framework, TensorFlow. This will at least to some extent reduce the familiarity of some subjects and will require the implementer to ask more questions.
ChatGPT will be asked to explain the general architecture and parts needed for the transformer, as well as recommend relevant libraries. Ready-made libraries will not be used, unless there is too much unrelated work.
To gauge the abilities of ChatGPT, some queries are formulated with a second objective in mind. An example would be to ask the model to explain some part of the transformer, but to do so in a very detailed manner. The resulting answers could then be used to check for whether the main objective (explaining the part) was completed and if the secondary objective was completed (taking the additional condition into account).
To see the extent of ChatGPT's usability: it is only allowed to reference other sources, such as official documentations and Stack Overflow, if absolutely stuck.

The evaluation will be limited to 1) judging the interactions with the chatbot and 2) the performance of the resulting model. While the evaluation of 2) is relatively straightforward due to its quantifiable properties, interactions also need to be judged based on the experiences of a human learner. Therefore, 1) consists of a quantifiable part, where the interactions can be statistically analyzed, and a "human experiences" part, where human views of the most relevant qualities are taken into account.
Testing for how well a learner has learned about the entirety of the subject would be great, but it would require some kind of test crafted by a domain-expert. Due to lacking the resources for that, it will be considered out-of-scope for this paper.

## 3.2 Resources

The equipment used for the project is a relatively basic desktop station for current standards. It consists of an AMD Ryzen 9 3900X 12-Core Processor, 3793 MHz CPU combined with a 4095MB NVIDIA GeForce RTX 2070 SUPER (ASUStek Computer Inc) GPU and 32 GB of RAM. The choice is intended to explore how well of a solution can be achieved for any individual, who isn't in possession of a powerful machine.

The dataset used is a relatively simple English-French translation set with 175621 sentences in their respective languages [63]. It is taken from the Tatoeba Project [64], which is a large open-source collection of sentences and their translations. The Tatoeba Project is based on crowdsourced contributions of translations.

As for the software, the goal would be to use Jupyter Notebook for the coding of the transformer in combination with the GPT-4 based ChatGPT, which would provide support in constructing the model.

## 3.3 GPT-4 interactions

The evaluation of the quality of interactions is split into 2 categories, which are quantifiable statistics and human impressions. Human impressions are the more abstract concept of the two and are therefore harder to properly judge.

Human impressions can come naturally, or they can be induced by intentional testing. A natural impression with the chatbot is achieved by simply interacting and noting all the good and bad qualities of its responses. An example could be that the bot responds with many paragraphs of text to even simple questions. In this case, it could probably be classified as a bad quality, as it would be producing an unnecessary amount of text.

An induced impression would be achieved by weaving in different types of queries to check how the bot handles them. Different things that will be added to queries include making the answer short ("Can you explain me this thing, but make the answer as short as possible."), using long queries, using error outputs directly as input queries, asking for specific code to be produced, asking about many differing subjects, using very detailed queries, asking multiple questions in one query, etc. Additionally, using ambiguous questions also needs to be tested to see how well the bot handles unclear queries. One example would be to refer to an earlier question, which actually doesn't exist, and seeing how the query is handled.

The handling of different types of queries can in addition also be quantified. Statistics for the interactions can be analyzed by, for instance, counting the amount of errors and checking the relative amount to the amount of all queries. By adding tags to each query, insights can be gained for theme frequencies, counts of consecutive questions used on some themes and so on.

By analyzing the prompts and responses themselves, recurring themes can be identified by plotting word clouds for both separately. Prompts and responses can also be broken down by their word lengths, whether they contain code or not, and whether some part

of the query affects the output in some noteworthy manner.

## 3.4 Coding

The process for the project will be in a logical order. First, ChatGPT will be asked to provide a high level architecture of a transformer model. This step will include asking for the code of the specific parts needed in an optimal order. These parts are then used as a base for the project.
The next step would be to start connecting the individual parts. Starting from the first part, all are glued together until every part is connected and working properly. When the model can be trained, ChatGPT will be used to improve performance. A key aspect will be tuning the hyperparameters.
The process is intended to first give an idea of what a transformer should look like, after which at least the inputs and outputs of different parts need to be considered. At some point all parts are needed, and at that point all parts also need to work correctly. Using ChatGPT to debug parts of the architecture will be the key method to solving errors or parts that aren't working properly. In other words, all parts need to be looked into to get the model working.
If there is absolute doubt and when there is no hope for progress, using other resources such as the official documentation or Stack Overflow is allowed. The final product is also compared to the original implementation to check for correctness and differences. The final model can be evaluated by checking its BLEU score and the translations it produces on the test set.

### 3.4.1 BLEU

Bilingual evaluation understudy (BLEU) [65] is a standard evaluation method in the field of machine translation. It measures the quality of a translated sequence in comparison to a reference sequence. The comparison is performed by using matching n-grams between the two sequences. A simplified example would be:

Translated sentence: A barrel rolls down the hill
Actual sentence: The barrel rolled down the hill

**Unigram:**

["A","barrel","rolls","down","the","hill"]
["The","barrel","rolled","down","the","hill"]

**Bigram:**

["A barrel","barrel rolls","rolls down","down the","the hill"]
["The barrel","barrel rolled","rolled down","down the","the hill"]

4/6 unigram elements match: "barrel", "down", "the", "hill". In comparison, 2/5 elements match for the bigram: "down the", "the hill". 4/6 and 2/5 are the respective accuracies for the unigram and bigram applied to the translated and actual sentence. Accuracies are calculated with the equation acc = $C/A$, where $C$ is the amount of correctly translated samples and $A$ is the amount of all the samples that were translated. Up to 4-grams are utilized in the BLEU calculation, after which the following returns from higher-order n-grams start to diminish enough that they don't need to be considered [65]. The final evaluation score is calculated as a combination of the accuracies in the following manner:

$$\begin{aligned} \text{BLEU} &= \text{BP} \cdot e^{\sum_{n=1}^{N} w_n \log p_n} \\ \text{BP} &= \begin{cases} 1, & \text{if } c > r \\ e^{1-r/c}, & \text{if } c \leq r \end{cases} \end{aligned} \tag{8}$$

Eq. 8 has 2 noteworthy parts. The first part is the calculation of the geometric weighted average of the accuracies $p_n$ of their respective n-grams ranging from 1 to $N$, where the positive weights $w_n$ sum up to 1. A typical way is to set the weights to be equal to each other. The second part is the exponential brevity factor $\text{BP} \cdot exp()$, which applies a penalty of $e^{1-r/c}$ on the score, if the candidate translation c is shorter or equal to the effective reference corpus length $r$. The goal of the penalty is to discourage producing too short translations, independent of accuracy.
Advantages of BLEU are that human evaluation largely agrees with it and its language independence. Disadvantages are that it doesn't take grammar nor fluency of the text into account.

# 4 Results

In total 261 prompts in 2 conversations of sizes 20 and 241 were needed to complete the creation, training and evaluation of the transformer model, for which the code can be found in appendix A. Sec. 4.1 and 4.2 respectively break down the results of the interactions and the model creation. Sec. 4.3 and Sec. 4.4 in turn will analyze perceived challenges and limitations, as well as the usability of ChatGPT in a research context.

## 4.1 GPT-4 interactions

As mentioned in Sec. 3.3, analyzing the interactions is split into a quantifiable statistics section, as well as a section about human experiences using ChatGPT for the task at hand.

### 4.1.1 Statictics

Every single one of the 261 prompts were hand-labeled to describe what the query was about.

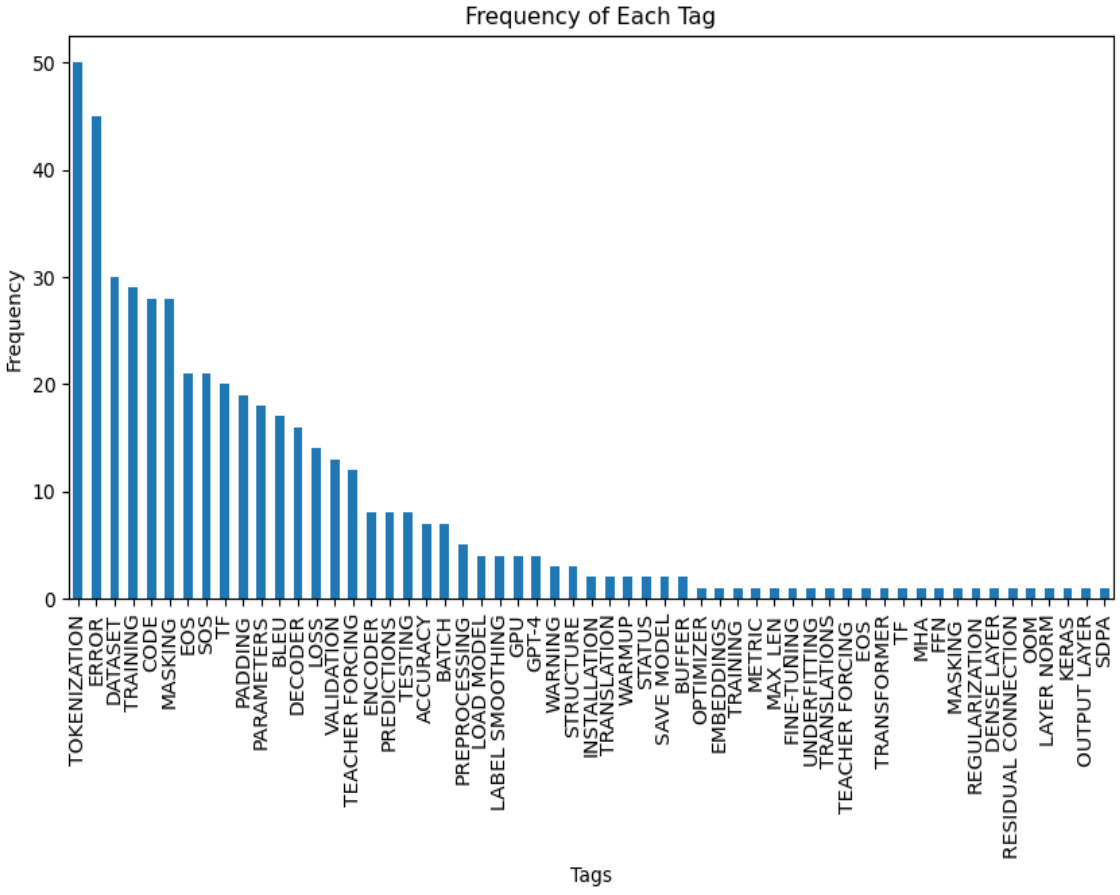



**Figure 7:** The frequencies of themes found in prompts.

Two separate conversations were used. The first was used to establish a base for the transformer code and to acquire an overview of the parts required for the model. The second and longer conversation was used to glue all the parts together, and to train and test the model. Fig. 7 shows the distribution of themes present in the queries. The most present themes in the prompts were "tokenization", "errors", "dataset", "training", "code", "masking", "EOS" and "SOS". This claim is supported by Fig. 8 and Fig. 9, which are respectively word clouds of the most frequent terms in the prompts and the responses. Most of the time was spent with either of two things, 1) finding the proper way to tokenize the sequences during data preparation, and 2) masking of encoder- and decoder inputs during training. While the tags "error" and "code" are general tags identifying the presence of either tag in a prompt, the rest of the most present tags can be said to belong to either tokenization, training or both. Some of the most frequent words that are specific towards a clear part of the model creation process are listed in Tab. 3 along with the total count of their appearances in the prompts and queries of the second conversation.

**Table 3:** Count of certain words in the second conversation.

| Word | Count |
|---|---|
| token: | 1559 |
| mask: | 1509 |
| train: | 913 |
| pad: | 911 |
| batch: | 773 |
| input: | 514 |
| eos: | 308 |
| bleu: | 295 |
| decoder: | 283 |
| error: | 206 |
| sos: | 148 |
| encoder: | 111 |

**Table 4:** Amounts of query-response pairs, where both given words appear in.

| Word pair | Count |
|---|---|
| token,train: | 99 |
| mask,pad: | 74 |
| train,input: | 73 |
| token,pad: | 72 |
| token,input: | 64 |
| token,mask: | 63 |
| train,batch: | 62 |
| token,batch: | 55 |
| train,pad: | 53 |
| pad,batch: | 53 |
| mask,batch: | 52 |
| mask,input: | 50 |

Word pairs in Tab. 4 are attained by searching whether a query-prompt pair contains both words of a pair, be it in a text or code part, either as the main subject or just being mentioned on the side. It can be used to spot deeper connections between some of the themes. The table is additionally sorted by the amount of query-response pairs the word pair appears in. The list consists mostly of themes appearing in the training part, while pairs such as "token, pad" and "token, mask" can also be connected to the data preparation.


**Figure 8:** A word cloud of the most frequent words found in prompts.


**Figure 9:** A word cloud of most frequent words in responses.

Having established the main areas of problems, what can be concluded from the types of problems experienced? During both tokenization and training, the problems didn't purely stem from received responses. It was a mixture of lacking knowledge by the user and inconsistent guiding by the chatbot. If the user misinterprets an answer, it is possible to lead the chatbot to agree with flawed logic or a wrong answer. One source of misinterpretation for users is the inconsistency of some answers. It didn't require two different conversations, but even two consequent queries got somewhat

false or inconsistent responses, which will be talked about further when talking about the code. The main problem can be summarized as the user not being aware of what he doesn't know, and due to that not being able to ask the correct questions, while the chatbot can be convinced to stray from correct instructions. That is why actual errors were relatively easy to solve compared to the implementation of parts of the model, which didn't have proper examples to build upon, but needed to be molded to fit the resources in use.

Another noticeable property of the interactions is that ChatGPT tends to produce very long responses, even for simpler questions. It can be seen from comparing the average query length of 25.4 words to the average response length of 324.7 words. Although the responses are long, they - on the positive side - often contain a lot of helpful details related to the question at hand.

### 4.1.2 Error handling

Out of 261 prompts, 49 were error related, which equals roughly 18.8% or less than a fifth of all queries. The longest chain of consecutive queries containing errors was 9. It was caused by the shapes of dataset items being handled incorrectly, when the training loop was run for the first time. There were also 2 chains of 4 consecutive errors, 1 of 3 errors, and the rest consisted of at most 2 consecutive errors. Besides the shape errors, which took a few tries to correct, most errors were solved fairly quickly. Additionally, 23 of all errors had trailing errors, which means that over half of all errors were solved with a single query.
The distribution of error types of errors directly used as input is displayed in Fig. 10. Most errors were caused by using improper input shapes for their respective use cases, such as forgetting to add the batching dimension, when it was required. Other errors were mostly typical cases, such as using integers when the function expects floats or trying to access an index out of bounds. Twice there were problems due to running out of memory for chosen model parameters, which are depicted as 'OOM' on the figure. All in all, there were not many difficult errors, and the errors that occurred were handled well. One thing to note is that in addition to errors it is important to add the specifications of what would be expected, as ChatGPT easily recommends the reshaping of inputs to a certain shape to suit a single use case, while the expected behavior might be to use dynamic shapes, such as with the different batches of sequences in the model. Compared to the struggles mentioned in the previous section, errors were relatively straightforward to solve and seemed very manageable for ChatGPT as well.

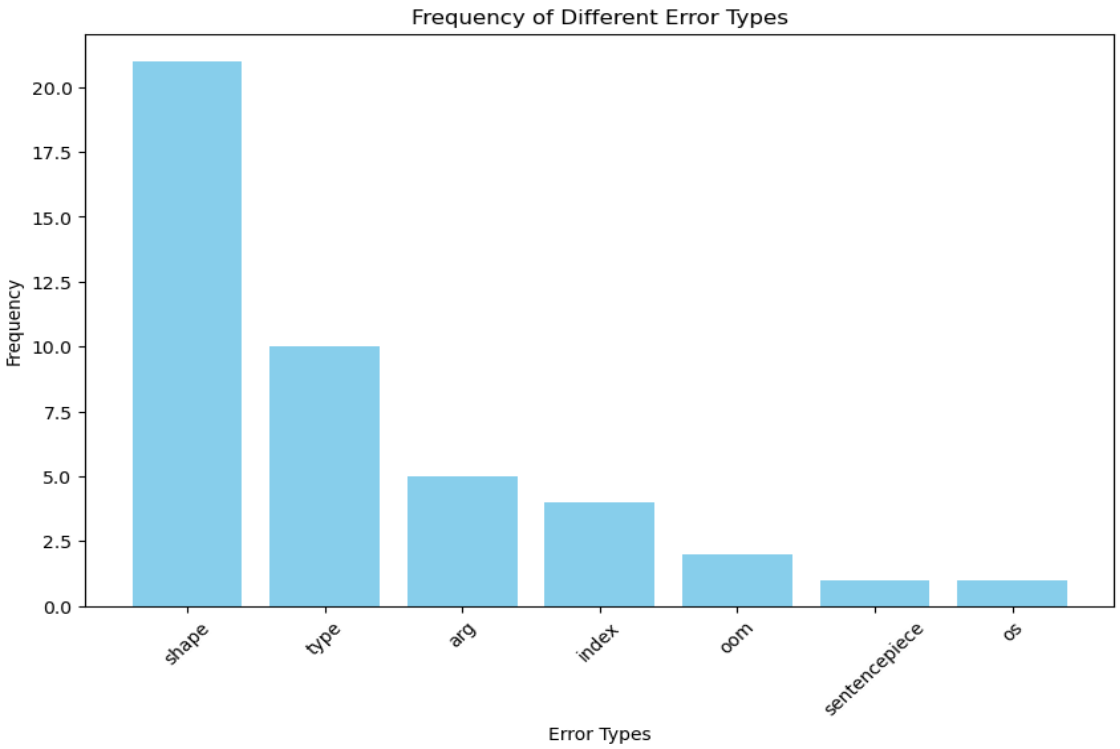



**Figure 10:** The distribution of error types throughout the coding process.

### 4.1.3 Impressions

During the interactions with ChatGPT, it was asked to produce responses with different kinds of conditions. As an example, Fig. 11 shows how the chatbot handled a query with an added condition of only using one sentence. It answered the query correctly, while also doing that in a single sentence as asked.

**You**

Can you tell me in one sentence how to check the shape and type of a tensor?

**ChatGPT**

Certainly! Use `tensor.shape` to check the shape and `tensor.dtype` to check the datatype of a TensorFlow tensor.

**Figure 11:** An example prompt with the request of responding in one sentence.

Similarly to the query in Fig. 11, the chatbot is able to answer very detailed queries as well. It can also answer multiple queries built into one, tested for at least 3 queries in one. Multilingual support also seems to work well, with similar queries tested in German, Finnish and English. All languages produced similar results, with the exception of English adding slightly more details.

Prompts specific to error handling were also tested. One aspect was to check how the bot answers a simple error output, as it would help see how the response matches the user's own thoughts. The answers to such questions were reliable in all cases tested. Another type of prompt was using the error output directly as a query. As mentioned in the previous section, the solution suggestions were correct, but would sometimes need additional specifications. For instance, if the shape needed to be handled dynamically, the specification needed to be added to the prompt. Otherwise, the suggestion would just be to mold all answers into some specific, constant shape.
Another type of test was passing code and an error output to the chatbot. This at times helped directly pinpoint the error. Explaining the error only using one's own words was also performed successfully, although it requires the prompt to be very specific about one's needs.
The original plan was to use a separate conversation for each ambiguously defined part of the model. The first conversation was just about producing the base for the code and getting an overview of the model. But as there were too many intertwined subjects already in the second conversation, it was used to the finish. Using the first conversation for the base enabled another kind of test, which was about the consistency of ChatGPT in different conversations. Asking the chatbot to reference a query from another conversation should be impossible, as it cannot access another conversation. Nonetheless, when asked it obliged and acted as if it could perform said action and produced an output, as if it knew what was meant. The output itself was not consistent with the previously mentioned query and caused more work, as the reply was trusted by the user. Answering ambiguous questions this way is not very desirable, as the answer will be mostly guessing the right meaning at that point. The bot should in those cases ask for clarifications.
Inconsistency was not only a problem between different conversations, but even in consequent queries. This will be further discussed in Sec. 4.2.3. As the conversation was getting longer, responses also took a longer time to generate.
It is important to note that it's easy to simply end up copying answers without paying enough mind to them. The ability to use ChatGPT properly relies on the user to be able to form helpful queries, as well as on the actual reflection of the answers.
As transformers are a relatively common subject, it would be interesting to see how ChatGPT handles more specialized concepts. One consideration in such cases would be how trustworthy the answers would become with fewer data on the subjects. That kind of uncertainty doesn't require a specialized subject, though, as the amount of redundant questions on SOS and EOS tokens during the coding phase show that even simple-seeming problems can cause a similar loss of trust towards answers. Generally useful properties of the chatbot are, among others, its ability to quickly create helper functions of small to medium size. Particularly for ML, it was able to recommend parameters, even pointing out useful parameters to tune with specific requests like "My transformer model is underfitting, what hyperparameters should I try to tune first?".
Another thing to mention is that it produces relevant code in its answers without being requested to. Out of all the responses, 184 contain code. 152 times out of those were either direct requests for code or it was in some other way implied. In 32 cases, it produced code of its own accord. Most often, it did so as an example. Although it

didn't always produce completely consistent answers, it usually could use the previous context of "using tf in python" and remembering a function, at the very least roughly, when referencing one. While ambiguous prompts could be handled better, general statements, which are close to being ambiguous, are handled very well. It might be a reference to a previous prompt, such as "What if I want to use tf", implying "Could the previous code snippet be replaced with one that uses TensorFlow?", and it usually understands the correct implication.
When the chatbot produced code, it explained the key parts. It also specified what certain variables were and what they were used for. On important lines, it added relevant comments to guide the user. These properties were useful, but easy to leave unappreciated, as they felt natural.
ChatGPT is also a great tool for assisting research. More about that will be in Sec. 4.4. The one qualm it has with respect to research is that it doesn't automatically contain citation, and when it does, it's often times citing Wikipedia.

## 4.2 Code

The following sections will break down the produced code and the key events related to the development.

### 4.2.1 Data preparation

The data preparation segment consists of the tokenization of the data, splitting the dataset into respective training-, validation- and test sets and batching and padding the sequences inside their respective batches. It was one of the two parts, where most difficulties were faced.
After acquiring the translation dataset, ChatGPT recommended tokenizing the data. For this task, it recommended the SentencePiece (SP) tokenizer. The wording used when asked about the need for preprocessing on two separate occasions differed slightly, which confused the user. Both stated that plain text was the main requirement, while one seemed to advise preprocessing steps more than the other. The chatbot was given the key properties of the dataset and asked for recommended initial parameters based on the information available. The resulting parameter suggestions seemed well justified and were used on the model.
A major obstacle for the user was at first trying to understand how and when the SOS and EOS tokens were supposed to be added to all the sequences. After many queries aimed at trying to understand the mistake, an implementation was found that seemed to work at first glance. Training the model for the first time revealed that the used method was not correct, as it was not correctly using the SOS and EOS tokens. They were not considered to be single tokens, but rather strings consisting of multiple tokens. Astonishingly, the model learned to translate with the faulty tokens. The correct solution for how to use tokens was found some time after that by using the SP documentation as an additional reference. The correct method was then acquired from ChatGPT by using a specific prompt describing the situation in detail.
During the later stages of the process, a maximum length for the target sequence

generation was needed, which was chosen by looking at the target language sequence lengths 12 and using the longest sequence as maximum length, as it wasn't too long to cause a significant increase in runtime.

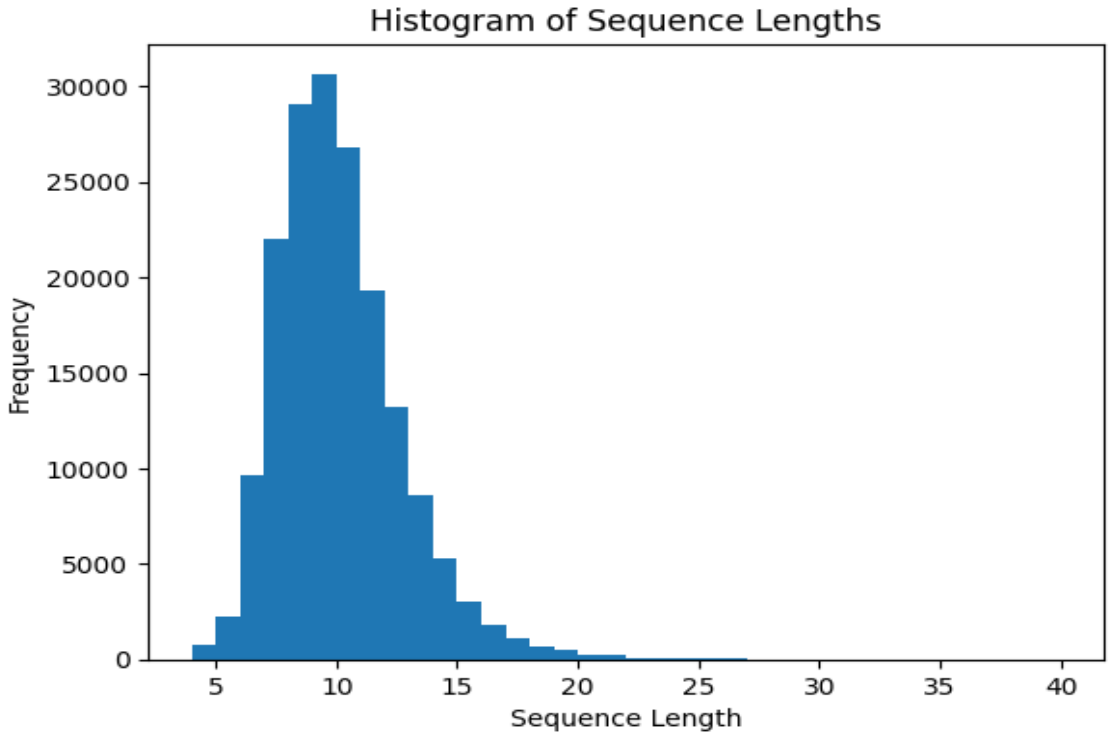



**Figure 12:** Length distribution of target sequences.

Splitting the dataset into respective training-, validation- and test sets was the next step after tokenization. ChatGPT was provided with relevant information about the task and data, and was asked for a recommended split ratio for the sets. It recommended an 80-10-10 split, which then was chosen and used.
During the creation of the respective sets, ChatGPT recommended masking the sequences at this point, which was then chosen to do. The next operation on the data was batching it and padding the sequences inside the batches according to the respective longest sequence. Problems arose from the padding and masking later on. The padding value was set to 0, which happened to also be the token for unknown values in SP. Debugging this problem with the chatbot was also more difficult than it seemingly should have been, but was also finally solved, mostly by the user personally debugging. Masking caused problems during creation of the decoder, which were amplified due to the recommended decoder structure stemming from the first conversation.
After testing different values for the batch size later, 64 was determined to be a fitting choice, as larger sizes sometimes ran out of memory. It worked as a middle ground, as it seemed large enough to be stable, but low enough to improve generalization. The choice of the value was based on suggestions presented by ChatGPT.

### 4.2.2 Helper functions

The helper functions consist of the functions get_positional_encoding, scaled_dot_product_attention, create_look_ahead_mask and pad_to_match. get_positional_encoding is used to create the positional encodings, which are added to the embeddings. The function resembles similar references of the function, which use Eq. 1 and Eq. 2 to calculate the encodings.
The scaled_dot_product_attention calculates attention weights according to Eq. 3. The function, just as many other parts which have been defined in the transformer model, also resembles and uses the original concept as reference.
Both, get_positional_encoding and scaled_dot_product_attention, produced errors due to masking at first. They were caused by incompatible shaping and typing of the inputs, which was debugged with relative ease using the chatbot. Besides that, there were no real requirements for extra tuning of the given functions to fit into the model.
The create_look_ahead_mask function was also very simple, as ChatGPT provided a function with a single line of code. To calculate the translation accuracy, a function was requested that would simplify the process. The input prompt was "When given two random length sequences, what is the fastest way to pad the shorter one to the length of the longer one?". As a result, ChatGPT provided the pad_to_match function found in the code. The use of code and the used language and frameworks didn't need to be specified, but the chatbot was, in this case, able to deduce them from the context of the previous queries.
All the given helper functions were provided by ChatGPT and could almost instantly be used in the model.

### 4.2.3 Model implementation

The core of the model consists of 7 classes: MultiHeadAttention (mha), PointWiseFeedForwardNetwork (ffn), EncoderLayer, Encoder, DecoderLayer, Decoder and Transformer. As depicted in Fig. 5, the mha applies the scaled dot-product attention on projections of the feature vector by using multiple heads. The dimensionality of the model, d_model, needs to be divisible by the number of heads, to be able to split the feature vectors across heads. The code for mha, provided by ChatGPT, also instantly fit the rest of the model.
While there were no problems with the received ffn class, there were not enough resources to increase the size of the layer. Instead, the chatbot was asked, whether compensating with more layers would work. It replied that it could work. The trade-offs would be longer and more difficult training, while possibly being able to overcome the problem of underfitting, which was the main problem at the time. Compared to the original transformer, this implementation uses more than 2 layers for the ffn.
The EncoderLayer and the Encoder calling the layers were both similar to reference solutions and had close to no problems. In Fig. 4 the gray box on the left side shows the architecture of an encoder layer. The entire encoder consists of multiple encoder layers. The Encoder class first handles the embedding and the encoding of the input, before calling the EncoderLayer class num_layers times. The EncoderLayer performs

the operations in Eq. 6 and passes the output to the next layer.
The DecoderLayer and the Decoder on the other hand were not as smoothly implemented. Similar to the encoder, the decoder is also defined in Fig. 4, but as the gray box on the right side. Operations in a decoder layer are performed according to Eq. 7. If the masks passed inside the decoder weren't poorly defined, the decoder would have worked fine.
The problem with masking in the decoder consisted of two separate causes. One was the formulation of the combined mask, which combines the look-ahead-mask and the decoder mask for the target sequence. In hindsight, the chatbot advised a very standard method for the mask implementation. Unfortunately, the user did not completely comprehend the context, and therefore ended up with the chatbot proposing the following mask operations inside the decoder instead:

```
def call(self, x, enc_output, training,
         look_ahead_mask, enc_padding_mask, dec_padding_mask):

        # Expanding the dimensions of look_ahead_mask
        look_ahead_mask = tf.expand_dims(look_ahead_mask, 0)
        look_ahead_mask = tf.tile(look_ahead_mask, [tf.shape(
    dec_padding_mask)[0], 1, 1])

        # Reshaping padding_mask to be compatible for
    multiplication
        padding_mask = tf.expand_dims(dec_padding_mask, -1)
        padding_mask = tf.broadcast_to(padding_mask, tf.shape(
    look_ahead_mask))

        look_ahead_mask = tf.cast(look_ahead_mask, tf.float32)
        padding_mask = tf.cast(padding_mask, tf.float32)
        combined_mask = tf.math.multiply(look_ahead_mask,
    padding_mask)
        ...
```

Now the user couldn't completely understand, what needed to be done. As the combination of the masks wasn't clear to the user after this, other references were used. A simpler method was found by changing the previous code snippet to the standard version:

```
def call(self, x, enc_output, training,
         look_ahead_mask, enc_padding_mask, dec_padding_mask):

        look_ahead_mask = tf.cast(look_ahead_mask, tf.float32)
        dec_padding_mask = tf.cast(dec_padding_mask, tf.float32)
        combined_mask = tf.math.maximum(look_ahead_mask,
    dec_padding_mask)
        ...
```

The new version required the changing of the creation of the sequence masks to:

```
def create_masks(padded_sequences):
    mask = tf.cast(padded_sequences == 0, dtype=tf.int32)
    return mask[:, tf.newaxis, tf.newaxis, :]
```

Here additional needed dimensions are added to the mask already at the time of creation.
The other problem with masking in the decoder was not as confusing, but displayed some problems with the chatbot. The original code for the decoder and its layer were received from the chatbot in the first conversation, which was simply used to create a base for the transformer model. Trying to refer to the transformer created in another conversation, while referencing it with "...in the transformer model you gave me..." should not have produced an answer, as the other conversation couldn't be accessed and no transformer had been created in the current conversation. Instead, the chatbot answered the ambiguous prompt with an inconsistent definition of the masks.
Upon further inspection, however, the Transformer class was defined to pass the decoder the decoder-mask and the look-ahead-mask, but not the encoder mask, which is required for the cross-attention mechanism. It is important to note that the Decoder class and Transformer class were received in the same conversation from successive prompts. This meant that the chatbot produced an untrue or at the very least misleading Transformer code with the context in mind. When questioned about the mistake, the chatbot quickly corrected the mistake and produced a working solution. It would have probably been simpler and more efficient to implement the masks of sequences inside the respective Encoder and Decoder classes.
The Transformer class calls the Encoder and Decoder, after which it passes the output of the decoder through a final linear layer. When the masking problems were solved, the model implementation was ready to run without any further errors.
The amount of trainable parameters of the entire transformer model are shown in Fig. 13.

```
Model: "transformer"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder (Encoder)           multiple                  60476416

 decoder (Decoder)           multiple                  66786304

 dense_108 (Dense)           multiple                  16416000

=================================================================
Total params: 143,678,720
Trainable params: 143,678,720
Non-trainable params: 0
_________________________________________________________________
Total trainable parameters: 143678720
```

**Figure 13:** Trainable parameters of the produced transformer model.

The additional layer in the ffn increases the amount of trainable parameters

significantly. ChatGPT was asked how many trainable parameters there would roughly be, if the amount of linear layers would be changed to 2 in the ffn. The first calculation produced a result of around 87 million parameters. During checking of the validity of the calculation, it was noticed that it calculated the result only using one vocabulary, instead of the two in use for this project. It was asked to recalculate with the corrected vocabularies, and produced a lower expected amount of 81 million parameters compared to its earlier calculation. Upon pointing out the inconsistency, the chatbot corrected itself with the responses in Fig. 14.

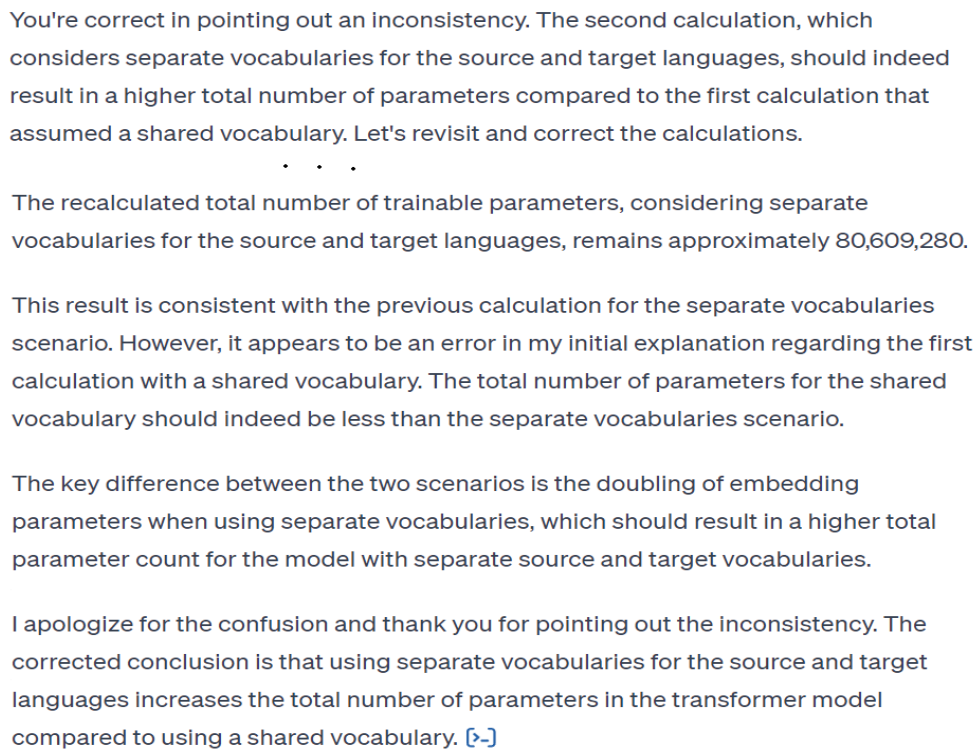



**Figure 14:** ChatGPT identifying an inconsistency pointed out by the user and correcting it.

The fact that it made a mistake, but was able to identify and correct it is positive, although it needed the user to point it out.
In actuality, the count of trainable parameters with a 2-layered ffn in the model is 93322496, which means that adding one layer caused an increase of 50 million trainable parameters, which in total exceeds the amount of trainable parameters in the first GPT model seen in Tab. 2.

### 4.2.4 Training and validation

The training of the model consisted mainly of the training loop, which utilized the class CustomSchedule and the functions loss_function, loss_function_smoothing, calculate_translation_accuracy, calculate_accuracy, validate, generate_output and train_step to perform the training and validation of the model. The CustomSchedule

class is defined according to the original parameters in the paper [33] and was left as provided, besides changing the number of warm-up steps.
The two loss functions differ in that one applies label smoothing, which softens the labels, while the other doesn't. Label smoothing is used to soften sharp distinctions of target labels, such as classifying the correct label as 1 and the others 0, to a less confident distribution of labeling the correct label as 0.9 and distributing the remaining 0.1 among the incorrect labels. This discourages the model from becoming too confident towards one solution, if other possible solutions exist, such as alternative translations for a sentence.
Label smoothing was recommended and produced by the chatbot as an additional hyperparameter to improve model performance. It is only used during training, as it is a regularization technique. Therefore, there is also the need for the normal loss function, which is used during validation and testing.
Similar to the loss functions, calculate_accuracy and calculate_translation_accuracy are respectively used for the calculation of training accuracy and evaluation accuracy. calculate_translation_accuracy needs the pad_to_match function to be able to calculate the accuracy between a translated sequence and a reference sequence. Both loss and accuracy are used as reference metrics to determine how well training and evaluation is progressing.
While training uses the targets of input sequences as ground truth, during evaluation the ground truth is not allowed to be used the same way, as the model should generate the translations itself. To accomplish this, the generate_output function was created. It calls the transformer function multiple times to predict the target token sequence for the respective input sequence. Due to the implementation structure of the transformer model, the generate_output function needed to be custom-tailored. Starting from a normal base solution, the function was molded in steps to fit the model architecture. The process was not as difficult as other problems, but still required some effort to complete using ChatGPT.
To calculate the different evaluation metrics, the validate function was created. As input, the validate function takes the trained transformer model, as well as the validation- or test set, depending on what is being evaluated. Both sets consist of data, which is previously unseen to the model. The function uses the prior loss_function, calculate_translation_accuracy and generate_output functions, as well as a ready-made function for BLEU-score calculation, to print sample translations and evaluation metrics. The purpose is to determine how well the trained model performs on unseen data. Just as the generate_output function, the validate function also needed to be custom designed for the model. It was created earlier than the previously mentioned function, so it saw a few more changes along the development. In its earlier stages, it only used 'teacher forcing', which is the concept of feeding the ground truth to the decoder to train the model.
With both functions, most problems originated from improper shaping of inputs and outputs. That is why validation and test sets were batched to size 1 to simplify the evaluation procedure by removing one dimension, as it wasn't required for the evaluation anyway. The implementation of the batched evaluation would have probably needed some effort, but would have been very possible to complete.

The train_step function handled the training of the model. Because the model was run for the first time during the train step function, most errors caused by different parts became apparent when running the training loop. In addition to shape- and type errors, and the masking problem in the decoder, the proper masking and use of SOS- and EOS tokens for the encoder and decoder inputs caused headaches. As most other problems were fixed, the model predicted tokens past the EOS token during training, up to the maximum length of the batch. These tokens were usually random tokens. As it happens, this is expected behavior, as the model shouldn't try to predict the padding following the EOS token. Instead, translated sequences are ended, when arriving at the EOS token. Due to the user not understanding this concept for a while, a lot of time was spent on trying to fix a problem that was not actually there. During the interactions it would have been beneficial to hear that the translation simply needs to be cut off after the EOS, but as the answers usually were pointing out that masking should ignore the padded tokens, the user needed to find it out on their own. The things the user had doubts about were how the shift of the target sequence should be properly handled, how the respective sequences needed to be masked and how the EOS token was used for the encoder and decoder inputs. It didn't help that the chatbot again started to get convinced by the flawed logic of the user, resulting in some wasted time on an improper implementation. Another flaw was how messy the transformer call was. It takes in a lot of parameters, and when it needed to be compared to a reference implementation for debugging purposes, it was too difficult to properly compare the models.
The problems experienced spurred the user on to further explore the entire process in detail. The proper understanding came to the user as a random thought and was therefore not really an accomplishment of the chatbot. It is partly the users fault, for not being able to ask the right questions. Upon realizing that it didn't matter what came after the EOS token, the proper form of the masking and token inputs were quickly found with the help of the chatbot.
One problem born from the previous tokenization, where the SOS- and EOS tokens were interpreted as strings by the model, was that the model training improved really fast, which of course was probably only overfitting to the training data, while the training with the proper tokens took a long time to improve significantly. It added to the doubt of whether the implementation of the training, the masking and tokenization was correct. Tuning the hyperparameters later revealed that the model was underfitting.
The training loop goes through the training data in batch_size sized chunks, which are passed to the train_step function to train the model. An epoch is defined by the whole training set having been passed that way through the train_step function. After every 5 epochs, the model is validated by passing it through the validate function using the validation set. This loop is continued until satisfactory model results have been achieved, after which it can proceed to the testing phase.

### 4.2.5 Hyperparameter tuning

The loss type and the optimizer recommended by the chatbot are "SparseCategoricalCrossentropy" and the "Adaptive Moment Estimation" (Adam) [66] optimizer.

Sparse categorical cross-entropy is used for multi-class classification problems, where the labels are integers. It's especially useful in situations, when there are a lot of classes. The loss is calculated as the negative log-likelihood of the true class, averaged over all samples in the batch. Mathematically for one sample the loss is defined as $-\log p_y$, where $p_y$ is the predicted probability for class $y$. Adam is used to adjust weights during training. It adjusts individual weights at separate rates. Two important factors are taken into account: momentum and scaling. Momentum determines the direction and speed, at which a weight changes from one step to another, which helps maintaining a consistent direction in weight adjustments. The scaling factor takes into account the rate, at which changes are happening for each individual weight. During rapid changes, a more cautious approach towards weight adjustment is taken.
As the training of the model for multiple epochs took a significant amount of time, initial parameter tuning was done for only a few epochs. ChatGPT was asked for hyperparameters to tune when the model is clearly underfitting. The plan was to tune the recommended parameters one by one and training the resulting model for at least 5 epochs. Based on the performance of the changed parameters, the new parameter values were either kept or reversed. Tab. 5 shows the compiled results for 9 different parameter choices. Constant hyperparameters were:

| | |
|---|---|
| beta 1: | 0.9 |
| beta 2: | 0.98 |
| epsilon: | 1e-9 |
| d model: | 512 |
| dff: | 2048 |
| batch size: | 64 |
| num layers: | 6 |
| num heads: | 8 |
| input vocab: | 32k |
| target vocab: | 32k |
| pe input: | 500 |
| pe target: | 500 |
| train-,val-,test size: | 20k,1k,1k |
| buffer size: | 10k |

Beta 1, beta 2 and epsilon are Adam-optimizer specific parameters. D model, dff and batch size stand for the dimensionality of the model, the dimensionality of the feed forward network and the size of a batch used during training. Increasing the first two would have helped the model during learning, but they couldn't be assigned any larger values due to running out of memory. The batch size also couldn't be increased much before crashing due to the same reason. During probes 64 seemed to be a fine batch size, as it was small enough to generalize, but large enough to be stable for this dataset. Num layers is the amount of layers used in the encoder and decoder, and num heads is the amount of attention heads that is used. The vocab parameters specify the vocabulary size for the input- and target language, while the "pe"-parameters specify the respective positional encoding, meaning, the maximal length, of input

and target sequences. The runs used a smaller training set, validation set and test set, with respective sizes of 20k, 1k and 1k samples. The reasoning for the choice was the saving of time. The final parameter is the buffer size of 10k, which indicates how much of the respective set gets shuffled. The learning rate (lr) is adjusted with the CustomSchedule class, which is dependent on the dimensionality of the model and the amount of warm-up steps.
The hyperparameters in the table columns are the amount of said warm-up steps, the depth of the feed forward network, the dropout rate, the maximum length of the sequence generation during evaluation, and whether punctuation has been normalized or not. Out of these, the chatbot recommended to especially tune the dropout and the learning rate, as well as the dff. Dropout could be changed directly, while lr and dff couldn't be. Lr is dependent on the d model and the amount of warm-up steps. D model could not be increased, so only the warm-up steps could be changed in a meaningful way. Similarly, the dimensionality of the feed forward network dff also couldn't be increased, so the compromise was to increase the depth of the ffn instead.

| warmup | ffn layers | drop | len | norm | tr $a_5$ | tr $l_5$ | v $a_5$ | v $b_5$ |
|---|---|---|---|---|---|---|---|---|
| 4k | 2 | 0.1 | batch | 0 | 0.542 | 2.929 | 0.315 | 0.0445 |
| 4k | 2 | 0.1 | batch | 1 | 0.503 | 2.907 | 0.306 | 0.0442 |
| 10k | 2 | 0.1 | batch | 1 | 0.444 | 3.555 | 0.271 | 0.0280 |
| 1k | 2 | 0.1 | batch | 1 | 0.120 | 5.515 | 0.111 | 0.0000 |
| 4k | 3 | 0.1 | batch | 1 | 0.523 | 2.802 | 0.355 | 0.0660 |
| 4k | 3 | 0.05 | batch | 1 | 0.539 | 2.676 | 0.366 | 0.0670 |
| 4k | 3 | 0.01 | batch | 1 | 0.558 | 2.561 | 0.369 | 0.0736 |
| 4k | 3 | 0.001 | batch | 1 | 0.565 | 2.516 | 0.351 | 0.0781 |
| 10k | 3 | 0.01 | 40 | 1 | 0.490 | 3.232 | 0.320 | 0.0326 |

**Table 5:** The different parameters used for different model iterations with their respective results.

All the made changes were tested in isolation, by changing one parameter with respect to the original values, and adopting said values, if they improved the results. The last 4 columns of Tab. 5 display the results for average training accuracy "tr $a$", average training loss "tr $l$", average translation accuracy "v $a$" and average validation BLEU "v $b$" after 5 epochs. Additionally, the first and last model in the table were trained for 15 epochs, with the first model achieving tr $a_{15}$: 0.608, tr $l_{15}$: 1.879, v $a_{15}$: 0.357 and v $b_{15}$: 0.1052. The respective results for the last model were tr $a_{15}$: 0.0.772, tr $l_{15}$: 1.161, v $a_{15}$: 0.469 and v $b_{15}$: 0.2253. The design choices that seemed to improve learning were: the increase of warm-up steps, the addition of another ffn layer and a lower dropout probability of 0.01. Additionally, defining a constant max length for text generation based on the longest target sequence in the dataset was chosen for consistency's sake for all models after the 8th model in Tab. 5. It was noticed after the first run that the inconsistent spacing around punctuation in the source language might affect outputs, which is why spacing around punctuation was normalized for both languages.

Increasing the warm-up on the 3rd model caused a decrease in model performance, but it was assumed to be due to the shorter run. The previous assumption could be deduced based on later results. After analyzing the results for the effects of lowering the dropout probability, there must have happened a mistake during evaluation, as the dropout of 0.001 seems to outperform the dropout of 0.01, but 0.01 was chosen going forward.

Once the parameter effects had been roughly charted out, 6 models A, B, C, D, E and F in Tab. 6 were created in alphabetical order of the letters to seek for the best performing model. For the models after A, the previously broken translation loss "v $l$" was fixed. Models after A also utilized the full buffer size for each set.
The significant changes of these models involved using the full data, with 140k training samples, 18k validation samples and 18k test samples. The dropout for each model was set to 0.01. All models also use punctuation normalization and a max length of 40 for output generation.

| model | warmup | ffn layers | buffer | heads |
|---|---|---|---|---|
| A | 10k | 3 | 10k | 8 |
| B | 10k | 3 | full | 8 |
| C | 10k | 3 | full | 16 |
| D | 10k | 4 | full | 8 |
| E | 15k | 3 | full | 8 |
| F | 10k | 3 | full | 8 |

**Table 6:** The parameters of the 6 different chosen models.

Tab. 6 displays the results of the 6 models after 5 epochs of training. From model B on, label smoothing was also applied, as recommended by ChatGPT. Model C was tested with more attention heads, as ChatGPT mentioned that increasing the amount of heads generally should increase the range of contextual relationships captured in a sequence. Model D was tested by increasing the layer count of the ffn to 4. Models E and F were modeled based on models A and B, but using more epochs of training, and E having more warm-up steps.

| model | tr $a_5$ | tr $l_5$ | v $a_5$ | v $l_5$ | v $b_5$ |
|---|---|---|---|---|---|
| A | 0.777 | 1.160 | 0.593 | - | 0.3677 |
| B | 0.781 | 2.394 | 0.593 | 3.463 | 0.3968 |
| C | 0.778 | 2.401 | 0.577 | 3.462 | 0.3893 |
| D | - | - | - | - | - |
| E | 0.800 | 2.325 | 0.602 | 3.470 | 0.4330 |
| F | 0.781 | 2.393 | 0.584 | 3.600 | 0.3936 |

**Table 7:** The training accuracies, training losses, translation accuracies, translation losses and average validation BLEU-scores after 5 epochs of training for the 6 chosen models.

Tab. 7 shows that model A, which was the same model as the last model of Tab. 5, but with the full data in use, performed better with more data. Applying the label smoothing and increasing the buffer size to cover the full train, val and test sets could have in hindsight been done in separate steps. The result was nonetheless better than model A, which is why both operations were used on the following models.
More heads in model C didn't seem to have any clearly improving effects, so they were reverted to 8 heads in the next models. Model D ran out of memory due to the additional layer, and was thus discarded. Model E and F both performed well, with E outperforming F with more warm-up steps.

| model | tr $a_{15}$ | tr $l_{15}$ | v $a_{15}$ | v $l_{15}$ | v $b_{15}$ |
|---|---|---|---|---|---|
| A | 0.884 | 0.5810 | 0.667 | - | 0.5278 |
| E | 0.892 | 1.859 | 0.673 | 2.909 | 0.5461 |
| F | 0.889 | 1.872 | 0.660 | 2.948 | 0.5389 |

**Table 8:** The training accuracies, training losses, translation accuracies, translation losses and average validation BLEU-scores after 15 epochs of training for models A, E and F.

Tab. 8 shows the results of the initial model A and the final two models E and F being run for 15 epochs. ChatGPT mentioned that label smoothing usually increases training loss, which can be seen from both Tab. 7 and Tab. 8, where model A is the only one lacking label smoothing. The training loss for A is lower, but the overall performance is in the favor of models applying label smoothing.
Model E has also been caught up to by F in terms of average BLEU validation.

| model | tr $a_{30}$ | tr $l_{30}$ | v $a_{30}$ | v $l_{30}$ | v $b_{30}$ |
|---|---|---|---|---|---|
| E | 0.951 | 1.644 | 0.699 | 2.885 | 0.6017 |
| F | 0.954 | 1.640 | 0.700 | 2.910 | 0.5864 |

**Table 9:** The training accuracies, training losses, translation accuracies, translation losses and average validation BLEU-scores after 30 epochs of training for models E and F.

Tab. 9 shows the final 2 models E and F, which were both run for 30 epochs. The training of both models took respectively approximately one day, with validations taking about 40% of the training time. The main goal was to see whether a higher amount of warm-up steps would lead to better performance on the model, which it, at least based on final validation BLEU-scores after 30 epochs, did.

#### 4.2.6 Evaluation

Validation and testing was performed using 10k samples from their respective sets. Tab. 10 shows the results of testing for the models A, B, C, E and F. E outperformed every other model in each aspect of testing. The closest competition was model F,

which managed to achieve an average test BLEU-score of 0.0141 lower than the best model.

| model | test accuracy | test loss | test bleu |
|---|---|---|---|
| A | 0.674 | - | 0.5477 |
| B | 0.584 | 3.524 | 0.4147 |
| C | 0.582 | 3.462 | 0.3779 |
| E | 0.697 | 2.912 | 0.5899 |
| F | 0.692 | 2.987 | 0.5758 |

**Table 10:** The test accuracies, test losses and average BLEU-scores on the test set of models A, B, C, E and F.

Despite some qualms such as not taking fluency into account [67], validity of BLEU in machine translation (MT) is often supported by the correlation with human expectations [68]. Based very roughly on Googles guidelines [69], the BLEU seems to indicate good performance on the test set. It has to be kept in mind, though, that the dataset is a simple one, and does not contain as large a variety as benchmark sets, such as WMT-2014 sets [70]. The dataset is also taken from an open source project, which means that although the translations are human-made, some might contain mistakes due to indirect translations and lacking inclusion of nuances. Even so, the dataset is popular among language learners, educators and NLP developers, which makes it suitable enough for the project at hand. In addition to being dependent on the quality of translations, BLEU also lacks the property to take context into account, which has to be kept in mind when evaluating translations. Having noted that, the test BLEU of 0.59 seems at least relatively good for the dataset at hand, indicating translations of high quality wrt. to accuracy. As the goal isn't to create the most competitive model, it is enough to check that the model is working as intended. To support the claim that the model seems to be working well, looking at the generated translations from a human perspective is needed as well.
Fig. 15 and Fig. 16 show the evolution of the training loss and training accuracy over 30 epochs, respectively. It seems that model E with 15k warm-up steps takes more time to decrease the loss and accuracy during the first five epochs, but then overtakes model F, until both metrics roughly converge after 15 epochs.
In contrast, Fig. 17 and Fig. 18 show the respective validation losses and validation accuracies over 30 epochs. Model E performs better on both metrics, but starts to converge with the results of the other model after 30 epochs, at least for the validation/translation accuracy.

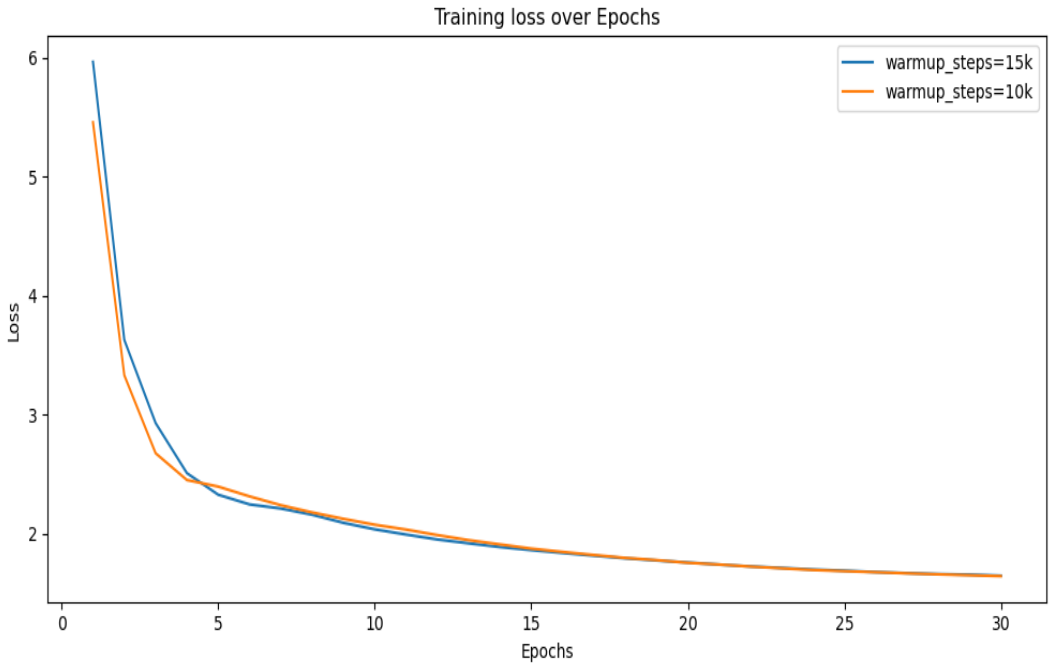



**Figure 15:** The training losses of models E and F over 30 epochs.

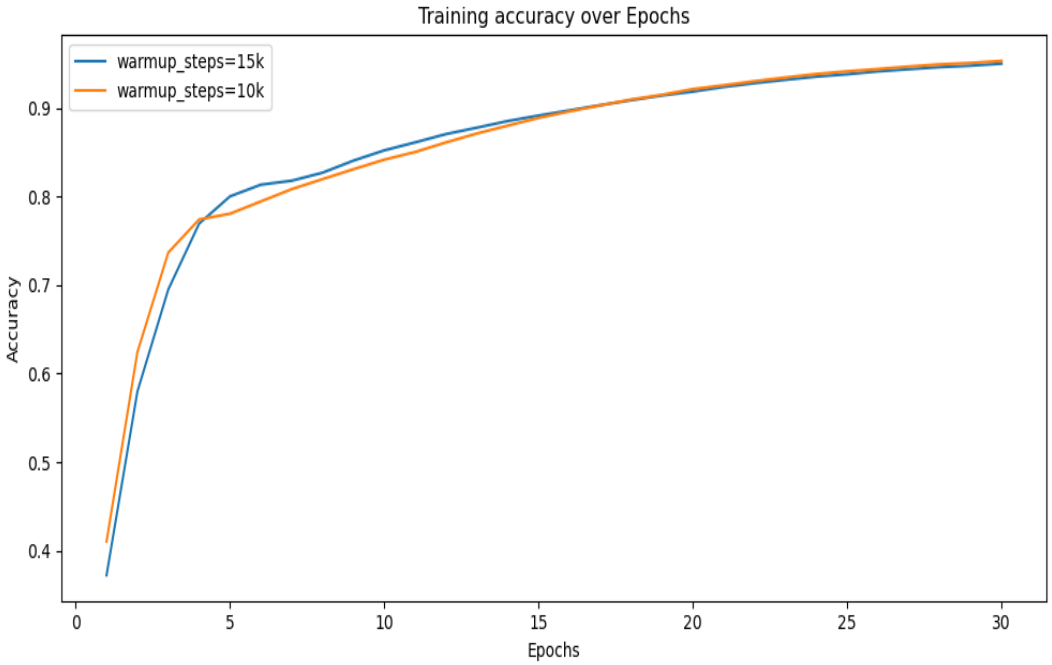



**Figure 16:** The training accuracies of models E and F over 30 epochs.

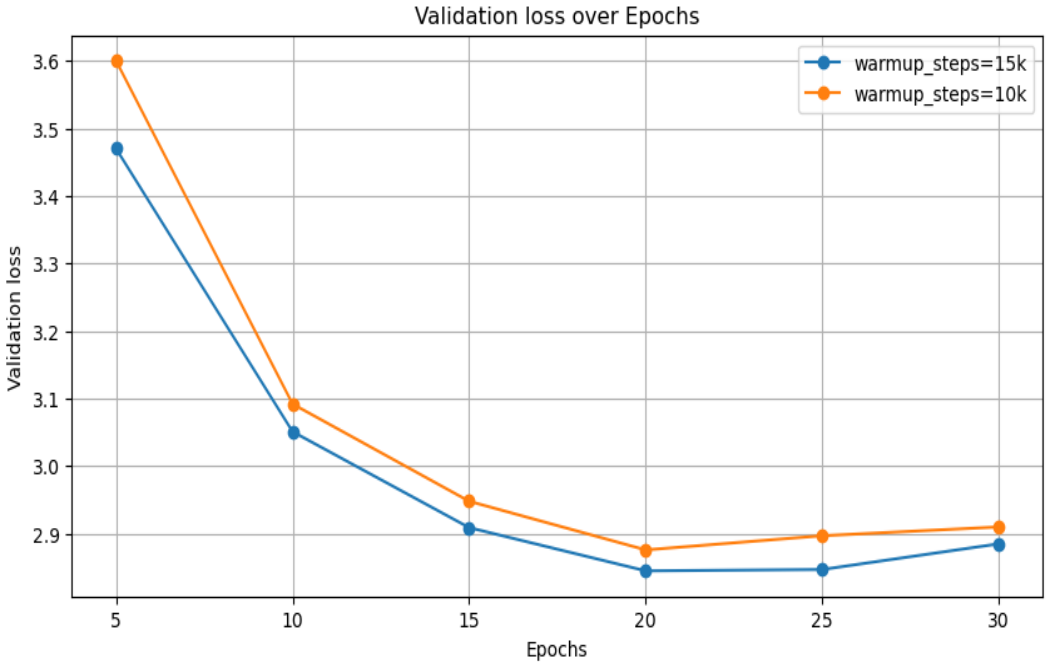



**Figure 17:** The validation losses of models E and F over 30 epochs.

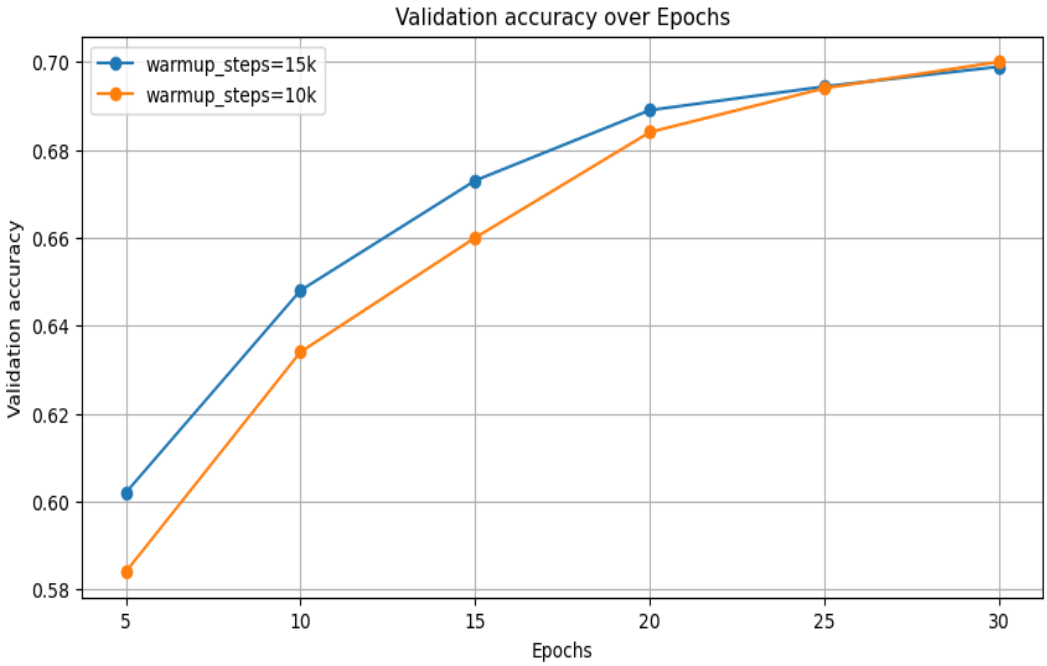



**Figure 18:** The validation/translation accuracies of models E and F over 30 epochs.

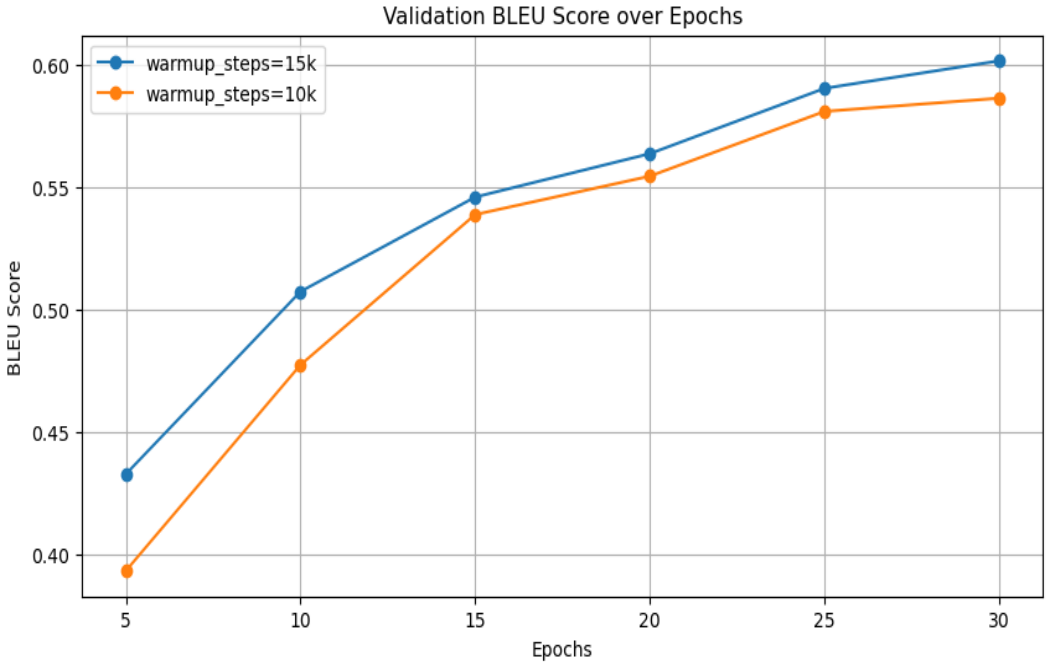



**Figure 19:** The average validation BLEU-scores of models E and F over 30 epochs.

Fig. 19 plots the average validation BLEU of models E and F against each other over 30 epochs of training. Here the takeaway is that the increased amount of warm-up steps improves the model, as model E strictly outperforms model F in terms of validation BLEU, as well as test BLEU in Tab. 10.

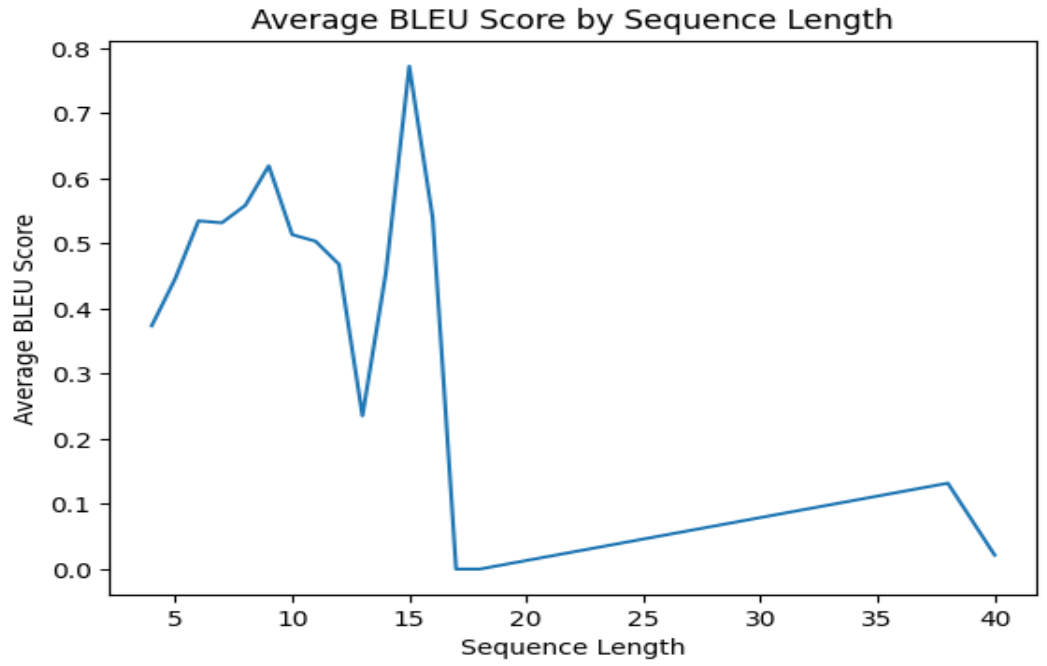



**Figure 20:** The average BLEU by sequence length.

Fig. 20 plots the average BLEU score by sequence length for 1000 test samples. It can be interpreted that the quality of translations drops for longer sequences, as sequences over the length of 15 don't reach an average BLEU over 0.2.

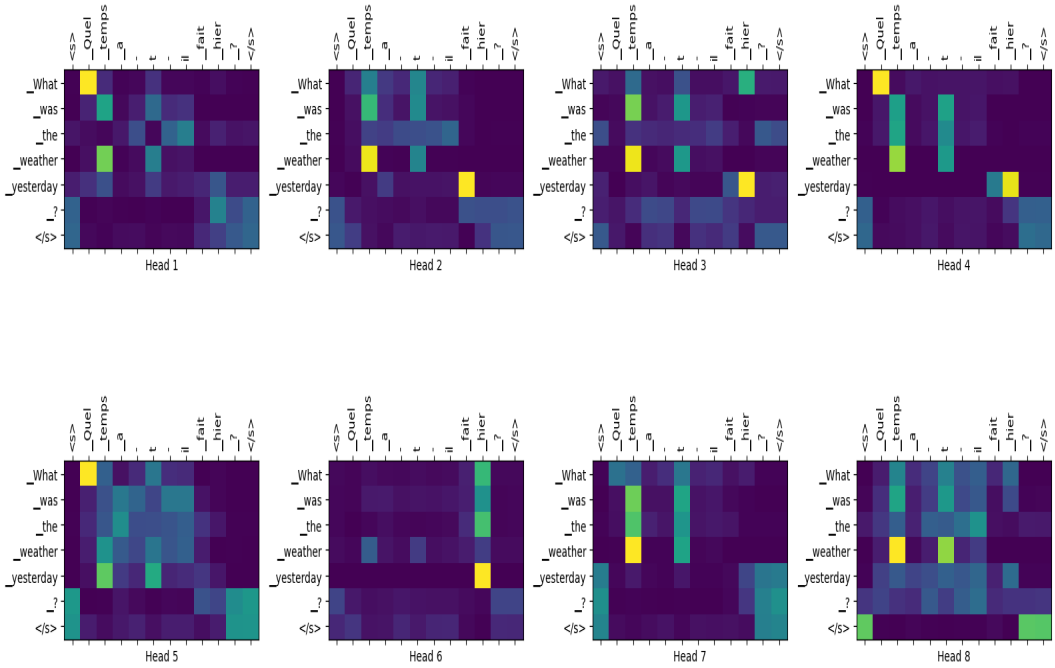



**Figure 21:** The attention weights of each respective head plotted as a heat map, for the translation of 'Quel temps a-t-il fait hier ?' to 'What was the weather yesterday ?'.

The attention of the sixth and last layer is plotted in Fig. 21 for the 8 respective attention heads. As source serves the French sentence "Quel temps a-t-il fait hier ?", which is translated to "What was the weather yesterday ?". Heat-maps are used to plot which part of the sequence each respective head is paying attention to. As an example: for head 1, the main focus is on the question word 'Quel', which it connects to 'What', as the "warmest" color indicates. The evolution of the attention for the sequence through the six layers can be seen in appendix B.

| | Predicted sequence | Actual sequence |
|---|---|---|
| 1 | Tom joined our group . | Tom joined our group . |
| 2 | It was raining . | It was raining . |
| 3 | I don't understand it at all . | I don't understand this at all . |
| 4 | He went to London yesterday . | He left for London yesterday . |
| 5 | We speak English all now . | English is spoken everywhere in the world now . |
| 6 | Forget what happened . | Never mind what happened . |
| 7 | They will survive . | They will survive . |
| 8 | It's personal . | This is personal . |
| 9 | I don't know how to use a . | I don't know how to operate a spinning wheel . |
| 10 | How much does the shirt cost ? | How much does the shirt cost ? |
| 11 | He asked me out on a date . | He asked me out on a date . |
| 12 | I've got one for you . | I've got one for you . |
| 13 | I didn't meet with each other yesterday . | It was not until yesterday that I knew the news . |
| 14 | You need that . | You need this . |
| 15 | Nobody likes war . | No one loves war . |

**Table 11:** 15 samples of translated sentences, highlighted where the translations differ.

Sample translations from the test set are shown in Tab. 11. Every translation besides 5, 9 and 13 seem to be either completely correct, have a different wording or a slight deviation from the original meaning. For translation 9 the problem seems to be that the term 'spinning wheel' was not in the training set, which is why it can't translate the word. Sentence 5 has some resemblance to the reference sentence, but is a very rough translation. The translation for 13 on the other hand has completely lost the original meaning of the reference sentence.
The resulting translations seem relatively good for the most part. To see the translations with their respective tokenization, refer to appendix C.

### 4.2.7 Other code

In addition to the mentioned code parts, ChatGPT was used to recommend and help with installations of necessary dependencies. Examples include the setting up of GPU usability for TensorFlow and helping with other libraries and TensorFlow related queries.

## 4.3 Challenges and limitations

Cases such as this - where there about 1000 lines of code and 200 previous queries exist - are difficult in the sense that the relevant context is hard to be kept in queries. At times, queries were aimed at changing previously received functions, but got responses lacking previously necessary properties. This can be either due to the prompt not

being specific enough, or due to the chatbot simply not being able to preserve the context. In a similar vein, it is also difficult for the user to comprehend a larger structure consisting of parts proposed by the chatbot. To add to this, the response generation of a conversation slows down as the length of the conversation increases.
Prompts and responses are reliant on how well the user can specify needs. The second part of the interaction relies on the user being able to trust and properly interpret the received answer. An additional layer to the trust problem is that the chatbot doesn't state how it proposes code, which could possibly be completely copied from another source.
The key problems for the code itself are currently that the code has not been optimized and scales poorly. The final model was trained for 30 epochs on 140k samples, which took 26 min per epoch and 1 h 40 min every fifth epoch for validation. The poor scalability is an intrinsic property caused by the nature of using questions to glue parts together. An additional complaint is that the only real evaluation metric is the BLEU at the time. It would have been advantageous to compare using the WMT-2014, which was also used on the original transformer.

## 4.4 GPT-4 and research advantages

The experience of using a reference freely in combination with ChatGPT was eye-opening. As one of the goals of the project was using ChatGPT in isolation as much as possible, it was helpful to have another reference to fact-check the outputs of the chatbot. Having a fail-safe for when the chatbot utters untrue statements increased the efficiency of the workflow. It also provided a clearer structure to base questions to ChatGPT on. It is definitely recommended to use a trusted reference to support the use of ChatGPT.
Strengths of the chatbot were also experienced during the research phase of the paper. Its ability to synthesize information by connecting different concepts, such as the connection between "word2vec", "attention", "transformer" and "GPT", was at times useful. Especially, when a source cited a connection, without further delving into the subject, information could be prompted from the chatbot. Other useful abilities during research were the ability to rephrase unfamiliar or difficult concepts, the summarizations and the use of examples, when some process was not clear enough.
During collection of results produced by the model, the chatbot was asked to provide simple functions to plot images or collect data regarding the model. It was able to produce many smaller functions to aid with information visualization, and they all worked as intended. This greatly reduced the time needed to code the functions by hand. It was also able to accommodate small variations to functions with great precision.

# 5 Summary

The goal of the paper was to find out how well GPT-4 based ChatGPT could be used as a tool to create a transformer model for MT purposes. The methodology used to achieve said goal consisted of creating, training and evaluating a transformer model for French to English translation with the support of ChatGPT. Support in this context meant prompting the chatbot to answer questions regarding the task at hand. Those interactions between the user and the chatbot were then analyzed based on the usability and other properties during the process. To achieve a result as realistic as possible, the user was barred from accessing ChatGPT or transformer specific information beforehand. After the model had been completed, the chatbot was further used to help in researching and analyzing results produced by the model.

People approach tasks differently and this was just one kind of approach, which still gives some ideas on general aspects of using ChatGPT for a DL task. Positives found during the project were the ability to create small functions, the ability to handle various types of queries, the summarization ability, the provided explanations, the rephrasing of concepts, the use of examples, and the flexibility to fix its own mistakes. Flexibility was not only positive, as the user could unknowingly convince the chatbot to agree to an untrue statement. Other negatives include inconsistency between given responses and the guessing of ambiguous prompts, rather than asking clarifying follow-up questions. An important finding was that to get the correct answer, the user needs to be able to properly specify the prompt.

The model required assistance from other references as well, as the user couldn't complete the task purely with ChatGPT. After the model was completed, other sources were utilized as references for the research part of the project. Based on the experience of using ChatGPT in combination with other guiding references, it is recommended to utilize other references as much as possible as ground truths for when the chatbot might fail.

For the model itself, ChatGPT provided well-defined parts, such as the usual architecture of a transformer model, in a standard format. By prompting, the functions it provided could also be customized to fit the task at hand. During training of the model, ChatGPT was also a useful tool for hyperparameter-tuning, as it was able to recommend situation specific parameters to be changed, while also defining what each parameter was for.

Combining all the parts was the hardest task of the project. The model itself didn't need too much attending to, but the data preparation, as well as the implementation of the training, needed to be molded according to the model inputs and outputs. As it sets a large part of the implementation responsibility on the user, it leaves more room for error. The requirement to keep the whole relevant context in prompts, while also defining them to produce the correct result, relied completely on the knowledge of the user. For this reason, it is advised to use supporting material to alleviate problems caused by said situation.

Results of the final model were good enough based on the evaluation criteria set during planning. To further test the usability of ChatGPT in a DL setting, an even more specialized subject could be used to check how well it handles themes, which it has little to no training data on.

# References


[1] S Padilla, CNN. Wikipedia releases its top 25 most-viewed pages of 2023. https://edition.cnn.com/2023/12/05/tech/wikipedia-chatgpt-oppenheimer-indian-entertainment/index.html, 2023. Accessed: 2023-12-10.

[2] A Murphy. Chatgpt's wikipedia page looked at by internet users more than any other in 2023. https://finance.yahoo.com/news/chatgpts-wikipedia-page-looked-internet-215357540.html, 2023. Accessed: 2023-12-10.

[3] Google trends - explore: "chatgpt, ai, /m/0mkz, /g/11khcfz0y2". https://trends.google.com/trends/explore?date=today%205-y&q=chatgpt,ai,%2Fm%2F0mkz,%2Fg%2F11khcfz0y2&hl=en-GB, 2023. Accessed: 2023-12-10.

[4] Y Liu L Deng. *Deep learning in natural language processing*, pages 1–12. Springer, 2018.

[5] WX Zhao, K Zhou, J Li, T Tang, X Wang, Y Hou, Y Min, B Zhang, J Zhang, Z Dong, Y Du, C Yang, Y Chen, Z Chen, J Jiang, R Ren, Y Li, X Tang, Z Liu, P Liu, JY Nie, and JR Wen. A survey of large language models, 2023.

[6] AM Turing. Mind. *Mind*, 59(236):433–460, 1950.

[7] LE Dostert. Brief history of machine translation research. In *Research in Machine Translation*, 1957.

[8] WJ Hutchins. The georgetown-ibm experiment demonstrated in january 1954. In RE Frederking and KB Taylor, editors, *Machine Translation: From Real Users to Research*, pages 102–114, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[9] VP Popescu. The impact of natural language processing on language learning and teaching. *Redefining Community in Intercultural Context*, page 171, 2023.

[10] C Manning and H Schutze. *Foundations of statistical natural language processing*. MIT press, 1999.

[11] ZS Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[12] CE Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[13] K Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[14] LE Baum and T Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, 37(6):1554–1563, 1966.

[15] AP Dempster, NM Laird, and DB Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society: series B (methodological)*, 39(1):1–22, 1977.

[16] X He, L Deng, and W Chou. Discriminative learning in sequential pattern recognition. *IEEE Signal Processing Magazine*, 25(5):14–36, 2008.

[17] K Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.

[18] J Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan kaufmann, 1988.

[19] V Vapnik. *The nature of statistical learning theory*, pages 138–156. Springer science & business media, 1999.

[20] M Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 conference on empirical methods in natural language processing (EMNLP 2002)*, pages 1–8, 2002.

[21] DE Rumelhart, GE Hinton, and RJ Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[22] R Collobert, J Weston, L Bottou, M Karlen, K Kavukcuoglu, and P Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537, 2011.

[23] GE Hinton, S Osindero, and YW Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[24] GE Dahl, D Yu, L Deng, and A Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 20(1):30–42, 2011.

[25] Y Bengio, R Ducharme, and P Vincent. A neural probabilistic language model. *Advances in neural information processing systems*, 13, 2000.

[26] T Mikolov, I Sutskever, K Chen, GS Corrado, and J Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[27] T Mikolov, K Chen, G Corrado, and J Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[28] S Hochreiter and J Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[29] K Cho, B Van Merriënboer, C Gulcehre, D Bahdanau, F Bougares, H Schwenk, and Y Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[30] M Schuster and KK Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[31] I Sutskever, O Vinyals, and QV Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[32] D Bahdanau, K Cho, and Y Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[33] A Vaswani, N Shazeer, N Parmar, J Uszkoreit, L Jones, AN Gomez, Ł Kaiser, and I Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[34] ME Peters, M Neumann, M Iyyer, M Gardner, C Clark, K Lee, and L Zettlemoyer. Deep contextualized word representations, 2018.

[35] J Devlin, MW Chang, K Lee, and K Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[36] A Radford, K Narasimhan, T Salimans, I Sutskever, et al. Improving language understanding by generative pre-training, 2018.

[37] OpenAI. Gpt-4 technical report, 2023.

[38] OpenAI. Introducing openai. https://openai.com/blog/introducing-openai, 2023. Accessed: 2023-12-05.

[39] Our structure - openai. https://openai.com/our-structure. Accessed: 2023-12-05.

[40] Dall-e 3 - openai. https://openai.com/dall-e-3. Accessed: 2023-12-05.

[41] A Radford, J Wu, R Child, D Luan, D Amodei, I Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[42] T Brown, B Mann, N Ryder, M Subbiah, JD Kaplan, P Dhariwal, A Neelakantan, P Shyam, G Sastry, A Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[43] L Gao, S Biderman, S Black, L Golding, T Hoppe, C Foster, J Phang, H He, A Thite, N Nabeshima, S Presser, and C Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[44] OpenAI. Chatgpt: Optimizing language models for dialogue. https://openai.com/blog/chatgpt, 2023. Accessed: 2023-12-04.

[45] E Chen, R Huang, HS Chen, YH Tseng, and LY Li. Gptutor: a chatgpt-powered programming tool for code explanation. *arXiv preprint arXiv:2305.01863*, 2023.

[46] S Biswas. Role of chatgpt in computer programming.: Chatgpt in computer programming. *Mesopotamian Journal of Computer Science*, 2023:8–16, 2023.

[47] NMS Surameery and MY Shakor. Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290*, 3(01):17–22, 2023.

[48] Y Wardat, MA Tashtoush, R AlAli, and AM Jarrah. Chatgpt: A revolutionary tool for teaching and learning mathematics. *Eurasia Journal of Mathematics, Science and Technology Education*, 19(7):em2286, 2023.

[49] LM Sánchez-Ruiz, S Moll-López, A Nuñez-Pérez, JA Moraño-Fernández, and E Vega-Fleitas. Chatgpt challenges blended learning methodologies in engineering education: A case study in mathematics. *Applied Sciences*, 13(10):6039, 2023.

[50] RA Khan, M Jawaid, AR Khan, and M Sajjad. Chatgpt-reshaping medical education and clinical management. *Pakistan Journal of Medical Sciences*, 39(2):605, 2023.

[51] H Lee. The rise of chatgpt: Exploring its potential in medical education. *Anatomical Sciences Education*, 2023.

[52] TB Arif, U Munaf, and I Ul-Haque. The future of medical education and research: Is chatgpt a blessing or blight in disguise?, 2023.

[53] W Geerling, GD Mateer, J Wooten, and N Damodaran. Chatgpt has aced the test of understanding in college economics: Now what? *The American Economist*, page 05694345231169654, 2023.

[54] W Geerling, GD Mateer, J Wooten, and N Damodaran. Is chatgpt smarter than a student in principles of economics? *Available at SSRN 4356034*, 2023.

[55] MM Rahman and Y Watanobe. Chatgpt for education and research: Opportunities, threats, and strategies. *Applied Sciences*, 13(9):5783, 2023.

[56] P Rospigliosi. Artificial intelligence in teaching and learning: what questions should we ask of chatgpt?, 2023.

[57] E Opara, A Mfon-Ette Theresa, and TC Aduke. Chatgpt for teaching, learning and research: Prospects and challenges. *Opara Emmanuel Chinonso, Adalikwu Mfon-Ette Theresa, Tolorunleke Caroline Aduke (2023). ChatGPT for Teaching, Learning and Research: Prospects and Challenges. Glob Acad J Humanit Soc Sci*, 5, 2023.

[58] CK Lo. What is the impact of chatgpt on education? a rapid review of the literature. *Education Sciences*, 13(4):410, 2023.

[59] E Kasneci, K Seßler, S Küchemann, M Bannert, D Dementieva, F Fischer, U Gasser, G Groh, S Günnemann, E Hüllermeier, et al. Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and individual differences*, 103:102274, 2023.

[60] M Halaweh. Chatgpt in education: Strategies for responsible implementation, 2023.

[61] Y Zheng. Chatgpt for teaching and learning: An experience from data science education. In *The 24th Annual Conference on Information Technology Education*, SIGITE '23. ACM, October 2023.

[62] EM Bender, T Gebru, A McMillan-Major, and S Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, pages 610–623, 2021.

[63] C Kelly. Language translation (english-french). https://www.kaggle.com/dsv/1067156, 2020.

[64] Tatoeba Project. https://tatoeba.org. Accessed on: 2023-12-07.

[65] K Papineni, S Roukos, T Ward, and WJ Zhu. 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, 2002.

[66] DP Kingma and J Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[67] Y Zhang, S Vogel, and A Waibel. Interpreting bleu/nist scores: How much improvement do we need to have a better system? In *LREC*, 2004.

[68] E Reiter. A structured review of the validity of bleu. *Computational Linguistics*, 44(3):393–401, 2018.

[69] Google Cloud. Evaluating models - automl translation documentation, 2023. Accessed: 2023-12-17.

[70] O Bojar, C Buck, C Federmann, B Haddow, P Koehn, J Leveling, C Monz, P Pecina, M Post, H Saint-Amand, et al. Findings of the 2014 workshop on statistical machine translation. In *Proceedings of the ninth workshop on statistical machine translation*, pages 12–58, 2014.

# A Transformer code

```
import re
import numpy as np
import tensorflow as tf
import sentencepiece as spm
import matplotlib.pyplot as plt

from datetime import datetime
from nltk.translate.bleu_score import corpus_bleu
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.sequence import pad_sequences


print("Num GPUs Available: ", len(tf.config.experimental.
    list_physical_devices('GPU')))
```

```
with open('eng_-french.csv', 'r', encoding='utf-8') as f, \
     open('english.txt', 'w', encoding='utf-8') as en, \
     open('french.txt', 'w', encoding='utf-8') as fr:
    for line in f:
        clean_line = " ".join(line.strip().split())
        english_sentence, french_sentence = clean_line.strip().
    split(',', 1)
        english_sentence = re.sub('([.,!?()"])', r' \1 ',
    english_sentence)
        english_sentence = re.sub('\s{2,}', ' ', english_sentence)
        french_sentence = re.sub('([.,!?()"])', r' \1 ',
    french_sentence)
        french_sentence = re.sub('\s{2,}', ' ', french_sentence)
        fr.write('' + french_sentence + '\n')
        en.write('' + english_sentence + '\n')
    f.close()
    en.close()
    fr.close()
```

```
# using french as the source language and english as the target
spm.SentencePieceTrainer.Train(input='french.txt',pad_id=0,unk_id
    =3,model_prefix='sourcemodel',vocab_size=32000,
    character_coverage=0.9995,model_type='bpe',control_symbols=['<s>
    ','</s>'])
spm.SentencePieceTrainer.Train(input='english.txt',pad_id=0,unk_id
    =3,model_prefix='targetmodel',vocab_size=32000,
    character_coverage=0.9995,model_type='bpe',control_symbols=['<s>
    ','</s>'])

# Load the trained models
sp_source = spm.SentencePieceProcessor()
sp_source.Load('sourcemodel.model')

sp_target = spm.SentencePieceProcessor()
sp_target.Load('targetmodel.model')

sos_fr = sp_source.piece_to_id('<s>')
```

```
eos_fr = sp_source.piece_to_id('</s>')

sos_en = sp_target.piece_to_id('<s>')
eos_en = sp_target.piece_to_id('</s>')

# Tokenize example
tokens_fr = []
french = open('french.txt', 'r')
Lines = french.readlines()
for line in Lines:
    tokens_fr.append([sos_fr] + sp_source.EncodeAsIds(line) + [eos_fr])


tokens_en = []
english = open('english.txt', 'r')
Lines = english.readlines()
for line in Lines:
    tokens_en.append([sos_en] + sp_target.EncodeAsIds(line) + [eos_en])


tokens_fr = tokens_fr[1:]
tokens_en = tokens_en[1:]

print(tokens_fr[0:15])
print(tokens_en[0:15])
```

```
# Calculate the lengths of each sequence
sequence_lengths = [len(seq) for seq in tokens_en]

# Calculate the statistics
longest_sequence = max(sequence_lengths)
average_sequence_length = np.mean(sequence_lengths)
percentile_90_length = np.percentile(sequence_lengths, 90)

# Output the stats
print(f"Longest sequence length: {longest_sequence}")
print(f"Average sequence length: {average_sequence_length:.2f}")
print(f"90th percentile sequence length: {percentile_90_length}")

# Plot the histogram
plt.hist(sequence_lengths, bins=range(min(sequence_lengths), max(sequence_lengths) + 1))
plt.title('Histogram of Sequence Lengths')
plt.xlabel('Sequence Length')
plt.ylabel('Frequency')
plt.show()
```

```
# Split data into training and testing (80-20 split, for example)
en_train, en_temp, fr_train, fr_temp = train_test_split(tokens_en, tokens_fr, test_size=0.2, random_state=42)

# Split the temp data again into validation and test sets (50-50
```

```
    split)
en_val, en_test, fr_val, fr_test = train_test_split(en_temp,
    fr_temp, test_size=0.5, random_state=42)

buffer_train = len(en_train)
buffer_val = len(en_val)
buffer_test = len(en_test)
print(buffer_train)
print(buffer_val)
print(buffer_test)
```

```
def create_masks(padded_sequences):
    mask = tf.cast(padded_sequences == 0, dtype=tf.int32)
    return mask[:, tf.newaxis, tf.newaxis, :]

def generator_fn():
    for source, target in zip(fr_train, en_train):
        yield source, target

output_signature = (
    tf.TensorSpec(shape=(None,), dtype=tf.int32),
    tf.TensorSpec(shape=(None,), dtype=tf.int32)
)

dataset = tf.data.Dataset.from_generator(
    generator_fn,
    output_signature=output_signature
)

# Specify your batch size
batch_size = 64

# Determine the padding shapes dynamically
padding_shapes = (tf.TensorShape([None]), tf.TensorShape([None]))

dataset = dataset.shuffle(buffer_size=buffer_train)  # Adjust
    buffer size as needed
# Take a subset of the entire training set to speed up training
#dataset = dataset.take(20000)
dataset = dataset.padded_batch(
    batch_size,
    padded_shapes=((tf.TensorShape([None]), tf.TensorShape([None]))
    )
)

def create_attention_masks(source_batch, target_batch):
    source_masks = create_masks(source_batch)
    target_masks = create_masks(target_batch)
    return (source_batch, source_masks), (target_batch,
    target_masks)

dataset = dataset.map(create_attention_masks)
dataset = dataset.prefetch(buffer_size=tf.data.experimental.
    AUTOTUNE)
```

```
40
41 def val_generator_fn():
42     for source, target in zip(fr_val, en_val):
43         yield source, target
44
45 valset = tf.data.Dataset.from_generator(
46     val_generator_fn,
47     output_signature=output_signature
48 )
49
50 # Specify your batch size
51 batch_size = 1
52
53 valset = valset.shuffle(buffer_size=buffer_val)  # Adjust buffer
       size as needed
54 # Take a subset
55 #valset = valset.take(1000)
56 valset = valset.padded_batch(
57     batch_size,
58     padded_shapes=((tf.TensorShape([None]), tf.TensorShape([None]))
      )
59 )
60
61 valset = valset.map(create_attention_masks)
62 valset = valset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
63
64 def test_generator_fn():
65     for source, target in zip(fr_test, en_test):
66         yield source, target
67
68 testset = tf.data.Dataset.from_generator(
69     test_generator_fn,
70     output_signature=output_signature
71 )
72
73 # Specify your batch size
74 batch_size = 1
75
76 testset = testset.shuffle(buffer_size=buffer_test)  # Adjust buffer
       size as needed
77 #testset = testset.take(1000)
78 testset = testset.padded_batch(
79     batch_size,
80     padded_shapes=((tf.TensorShape([None]), tf.TensorShape([None]))
      )
81 )
82
83 testset = testset.map(create_attention_masks)
84 testset = testset.prefetch(buffer_size=tf.data.experimental.
      AUTOTUNE)
```

```
1 def get_positional_encoding(max_len, d_model):
2     # Create a matrix to store the positional encodings
3     pos_enc = np.array([
```

```
        [pos / np.power(10000, 2.0 * (j // 2) / d_model) for j in range(d_model)]
        for pos in range(max_len)
    ])

    # Apply the sine to the even indices and the cosine to the odd indices
    pos_enc[:, 0::2] = np.sin(pos_enc[:, 0::2])
    pos_enc[:, 1::2] = np.cos(pos_enc[:, 1::2])

    pos_enc = pos_enc[np.newaxis, ...]  # Add a new dimension for the batch size
    return tf.cast(pos_enc, tf.float32)
```

```
def scaled_dot_product_attention(q, k, v, mask=None):
    """
    Calculate the attention weights.
    q, k, v must have matching leading dimensions.
    The mask has different shapes depending on its type (padding or look ahead)
    but it must be broadcastable for addition.

    Args:
    - q: query shape == (..., seq_len_q, depth)
    - k: key shape == (..., seq_len_k, depth)
    - v: value shape == (..., seq_len_v, depth_v)
    - mask: Float tensor with shape broadcastable to (..., seq_len_q, seq_len_k). Defaults to None.

    Returns:
    - output, attention_weights
    """

    # Calculate the dot product
    matmul_qk = tf.matmul(q, k, transpose_b=True)

    # Scale the dot product
    d_k = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_attention_logits = matmul_qk / tf.math.sqrt(d_k)

    # Apply the mask if provided
    if mask is not None:

        mask = tf.cast(mask, dtype=tf.float32)

        scaled_attention_logits += (mask * -1e9)

    # Softmax to get the attention weights
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)

    # Multiply the attention weights with the value matrix
    output = tf.matmul(attention_weights, v)

```

```
    return output, attention_weights

def create_look_ahead_mask(size):
    # Create a lower triangular matrix and subtract it from 1.
    # This will produce a matrix with zeros in its lower triangle and ones elsewhere.
    mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
    return mask  # (seq_len, seq_len)

def pad_to_match(seq1, seq2, padding_value=0):
    # Calculate the lengths of the sequences
    len_seq1 = tf.shape(seq1)[0]
    len_seq2 = tf.shape(seq2)[0]

    # Calculate the amount of padding needed for each sequence
    padding_seq1 = tf.maximum(len_seq2 - len_seq1, 0)
    padding_seq2 = tf.maximum(len_seq1 - len_seq2, 0)

    # Pad each sequence to the same length
    seq1_padded = tf.pad(seq1, paddings=[[0, padding_seq1]], constant_values=padding_value)
    seq2_padded = tf.pad(seq2, paddings=[[0, padding_seq2]], constant_values=padding_value)

    return seq1_padded, seq2_padded

class MultiHeadAttention(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.d_model = d_model

        assert d_model % self.num_heads == 0

        self.depth = d_model // self.num_heads

        self.wq = tf.keras.layers.Dense(d_model)
        self.wk = tf.keras.layers.Dense(d_model)
        self.wv = tf.keras.layers.Dense(d_model)

        self.dense = tf.keras.layers.Dense(d_model)

    def split_heads(self, x, batch_size):
        x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
        return tf.transpose(x, perm=[0, 2, 1, 3])

    def call(self, v, k, q, mask):
        batch_size = tf.shape(q)[0]

        q = self.wq(q)
        k = self.wk(k)
        v = self.wv(v)

```

```
        q = self.split_heads(q, batch_size)
        k = self.split_heads(k, batch_size)
        v = self.split_heads(v, batch_size)

        # Use the previously defined scaled_dot_product_attention function
        scaled_attention, attention_weights = scaled_dot_product_attention(
            q, k, v, mask)

        scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])
        concat_attention = tf.reshape(scaled_attention,
                                      (batch_size, -1, self.d_model))

        output = self.dense(concat_attention)

        return output, attention_weights
```

```
class PointWiseFeedForwardNetwork(tf.keras.layers.Layer):
    def __init__(self, d_model, dff):
        super(PointWiseFeedForwardNetwork, self).__init__()

        # First dense layer
        self.dense1 = tf.keras.layers.Dense(dff, activation='relu')  # dff: dimensionality of feed-forward inner layer

        # Second dense layer
        self.dense2 = tf.keras.layers.Dense(dff, activation='relu')  # dff: dimensionality of feed-forward inner layer

        # Final dense layer
        self.dense3 = tf.keras.layers.Dense(d_model)

    def call(self, x):
        return self.dense3(self.dense2(self.dense1(x)))
```

```
class EncoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, dff, rate=0.1):
        super(EncoderLayer, self).__init__()

        self.mha = MultiHeadAttention(d_model, num_heads)  # Assuming you've defined this earlier
        self.ffn = PointWiseFeedForwardNetwork(d_model, dff)  # Assuming you've defined this

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        self.dropout1 = tf.keras.layers.Dropout(rate)
        self.dropout2 = tf.keras.layers.Dropout(rate)
```

```
13
14      def call(self, x, training, mask):
15
16          attn_output, _ = self.mha(x, x, x, mask)  # Self attention
17          attn_output = self.dropout1(attn_output, training=training)
18          out1 = self.layernorm1(x + attn_output)
19
20          ffn_output = self.ffn(out1)
21          ffn_output = self.dropout2(ffn_output, training=training)
22          out2 = self.layernorm2(out1 + ffn_output)
23
24          return out2
25
26  class Encoder(tf.keras.layers.Layer):
27      def __init__(self, num_layers, d_model, num_heads, dff,
        input_vocab_size,
28                   maximum_position_encoding, rate=0.1):
29          super(Encoder, self).__init__()
30
31          self.d_model = d_model
32          self.num_layers = num_layers
33
34          self.embedding = tf.keras.layers.Embedding(input_vocab_size
        , d_model)
35          self.pos_encoding = get_positional_encoding(
        maximum_position_encoding, self.d_model)  # Assuming you've
        defined positional encoding function
36          self.pos_encoding = tf.cast(self.pos_encoding, tf.float32)
37
38          self.enc_layers = [EncoderLayer(d_model, num_heads, dff,
        rate)
39                             for _ in range(num_layers)]
40
41          self.dropout = tf.keras.layers.Dropout(rate)
42
43      def call(self, x, training, mask):
44
45          seq_length = tf.shape(x)[1]
46
47          # Adding embedding and position encoding
48          x = self.embedding(x)
49          x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
50          x += tf.gather(self.pos_encoding, indices=tf.range(
        seq_length), axis=1)
51
52          x = self.dropout(x, training=training)
53
54          for i in range(self.num_layers):
55              x = self.enc_layers[i](x, training, mask)
56
57          return x  # (batch_size, input_seq_len, d_model)
```

```
1   class DecoderLayer(tf.keras.layers.Layer):
2       def __init__(self, d_model, num_heads, dff, rate=0.1):
```

```
        super(DecoderLayer, self).__init__()

        self.mha1 = MultiHeadAttention(d_model, num_heads)  # Self attention
        self.mha2 = MultiHeadAttention(d_model, num_heads)  # Cross attention with encoder's output

        self.ffn = PointWiseFeedForwardNetwork(d_model, dff)

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        self.dropout1 = tf.keras.layers.Dropout(rate)
        self.dropout2 = tf.keras.layers.Dropout(rate)
        self.dropout3 = tf.keras.layers.Dropout(rate)

    def call(self, x, enc_output, training, combined_mask, enc_padding_mask):
        # enc_output.shape == (batch_size, input_seq_len, d_model)

        attn1, attn_weights_block1 = self.mha1(x, x, x, combined_mask)  # Self attention
        attn1 = self.dropout1(attn1, training=training)
        out1 = self.layernorm1(attn1 + x)

        attn2, attn_weights_block2 = self.mha2(enc_output, enc_output, out1, enc_padding_mask)  # Cross attention
        attn2 = self.dropout2(attn2, training=training)
        out2 = self.layernorm2(attn2 + out1)

        ffn_output = self.ffn(out2)
        ffn_output = self.dropout3(ffn_output, training=training)
        out3 = self.layernorm3(ffn_output + out2)

        return out3, attn_weights_block1, attn_weights_block2

class Decoder(tf.keras.layers.Layer):
    def __init__(self, num_layers, d_model, num_heads, dff, target_vocab_size,
                 maximum_position_encoding, rate=0.1):
        super(Decoder, self).__init__()

        self.d_model = d_model
        self.num_layers = num_layers

        self.embedding = tf.keras.layers.Embedding(target_vocab_size, d_model)
        self.pos_encoding = get_positional_encoding(maximum_position_encoding, d_model)
        self.pos_encoding = tf.cast(self.pos_encoding, tf.float32)
```

```
        self.dec_layers = [DecoderLayer(d_model, num_heads, dff,
    rate)
                           for _ in range(num_layers)]
        self.dropout = tf.keras.layers.Dropout(rate)

    def call(self, x, enc_output, training,
             look_ahead_mask, enc_padding_mask, dec_padding_mask):

        look_ahead_mask = tf.cast(look_ahead_mask, tf.float32)
        dec_padding_mask = tf.cast(dec_padding_mask, tf.float32)
        combined_mask = tf.math.maximum(look_ahead_mask,
    dec_padding_mask)

        seq_length = tf.shape(x)[1]
        attention_weights = {}

        x = self.embedding(x)
        x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
        x += tf.gather(self.pos_encoding, indices=tf.range(
    seq_length), axis=1)

        x = self.dropout(x, training=training)

        for i in range(self.num_layers):
            x, block1, block2 = self.dec_layers[i](x, enc_output,
    training, combined_mask, enc_padding_mask) # Use
    enc_padding_mask for cross attention

            attention_weights[f'decoder_layer{i+1}_block1'] =
    block1
            attention_weights[f'decoder_layer{i+1}_block2'] =
    block2

        return x, attention_weights  # x.shape == (batch_size,
    target_seq_len, d_model)
```

```
class Transformer(tf.keras.Model):
    def __init__(self, num_layers, d_model, num_heads, dff,
    input_vocab_size,
                 target_vocab_size, pe_input, pe_target, rate=0.1):
        super(Transformer, self).__init__()

        self.encoder = Encoder(num_layers, d_model, num_heads, dff,
                               input_vocab_size, pe_input, rate)

        self.decoder = Decoder(num_layers, d_model, num_heads, dff,
                               target_vocab_size, pe_target, rate)

        self.final_layer = tf.keras.layers.Dense(target_vocab_size)

    @tf.function(input_signature=[
      tf.TensorSpec(shape=(None, None), dtype=tf.int32, name='inp')
    ,
```

```
      tf.TensorSpec(shape=(None, None), dtype=tf.int32, name='tar'),
      tf.TensorSpec(shape=(), dtype=tf.bool, name='training'),
      tf.TensorSpec(shape=(None, None, None, None), dtype=tf.int32, name='enc_padding_mask'),
      tf.TensorSpec(shape=(None, None), dtype=tf.float32, name='look_ahead_mask'),
      tf.TensorSpec(shape=(None, None, None, None), dtype=tf.int32, name='dec_padding_mask')
    ])
    def call(self, inp, tar, training, enc_padding_mask,
             look_ahead_mask, dec_padding_mask):

        enc_output = self.encoder(inp, training, enc_padding_mask)  # (batch_size, inp_seq_len, d_model)

        # dec_output.shape == (batch_size, tar_seq_len, d_model)
        dec_output, attention_weights = self.decoder(
            tar, enc_output, training, look_ahead_mask, enc_padding_mask, dec_padding_mask)

        final_output = self.final_layer(dec_output)  # (batch_size, tar_seq_len, target_vocab_size)

        return final_output, attention_weights
```

```
class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
    def __init__(self, d_model, warmup_steps=15000):
        super(CustomSchedule, self).__init__()

        self.d_model = d_model
        self.d_model = tf.cast(self.d_model, tf.float32)

        self.warmup_steps = warmup_steps

    def __call__(self, step):
        arg1 = tf.math.rsqrt(step)
        arg2 = step * (self.warmup_steps ** -1.5)

        return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)

d_model = 512
learning_rate = CustomSchedule(d_model)

optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9, beta_2=0.98,
                                     epsilon=1e-9)
```

```
num_layers = 6

num_heads = 8
dff = 2048
```

```
input_vocab_size = 32000
target_vocab_size = 32000
pe_input = 500
pe_target = 500
rate=0.01

transformer = Transformer(num_layers, d_model, num_heads, dff,
    input_vocab_size, target_vocab_size, pe_input, pe_target, rate=
    rate)  # Fill with your parameters

checkpoint_path = "./checkpoints/train"

ckpt = tf.train.Checkpoint(transformer=transformer, optimizer=
    optimizer)

ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path,
    max_to_keep=5)

# If a checkpoint exists, restore the latest checkpoint.
#if ckpt_manager.latest_checkpoint:
#    ckpt.restore(ckpt_manager.latest_checkpoint)
#    print ('Latest checkpoint restored!!')
```

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

def loss_function(real, pred):

    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)

    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

    return tf.reduce_sum(loss_)/tf.reduce_sum(mask)

def loss_function_smoothing(real, pred, smoothing_rate=0.1):
    num_classes = tf.shape(pred)[-1]  # Assuming pred is [
    batch_size, seq_len, num_classes]

    # Convert 'real' to a one-hot encoding of type float32
    real_one_hot = tf.one_hot(real, depth=num_classes, dtype=tf.
    float32)

    # Apply label smoothing
    real_smoothed = real_one_hot * (1.0 - smoothing_rate) + (
    smoothing_rate / tf.cast(num_classes, tf.float32))

    # Compute the cross-entropy loss using smoothed labels
    cross_entropy = tf.keras.losses.categorical_crossentropy(
    real_smoothed, pred, from_logits=True)

    # Create a mask for non-padding labels (assuming 0 is the
    padding label)
```

```
    mask = tf.cast(tf.math.logical_not(tf.math.equal(real, 0)), dtype=cross_entropy.dtype)

    # Apply the mask to filter out the contribution of padding to the loss
    loss_ = cross_entropy * mask

    # Calculate the average loss considering the mask
    return tf.reduce_sum(loss_) / tf.reduce_sum(mask)

def calculate_translation_accuracy(predictions, references):
    # Ensure that both predictions and references are numpy arrays
    predictions = np.array(predictions)
    references = np.array(references)

    # Calculate the total number of tokens (excluding padding)
    total_tokens = np.sum(references != 0)  # Assuming 0 is the padding token

    # Calculate the number of correct predictions
    correct_predictions = np.sum((predictions == references) & (references != 0))

    # Calculate accuracy
    accuracy = correct_predictions / total_tokens
    return accuracy

def calculate_accuracy(logits, targets, padding_value=0):
    # Step 1: Convert logits to class predictions
    predictions = tf.argmax(logits, axis=-1, output_type=tf.int32)

    # Step 2: Create a mask to filter out padding tokens from the targets
    mask = tf.not_equal(targets, padding_value)

    # Step 3: Cast the mask to the same dtype as the predictions
    mask = tf.cast(mask, dtype=predictions.dtype)

    # Step 4: Use the mask to filter out padding values from the predictions and targets
    masked_predictions = tf.boolean_mask(predictions, mask)
    masked_targets = tf.boolean_mask(targets, mask)

    # Step 5: Compare predictions to targets to get a tensor of 1s (correct) and 0s (incorrect)
    correct_predictions = tf.cast(tf.equal(masked_predictions, masked_targets), tf.float32)

    # Step 6: Calculate accuracy as mean of correct predictions
    accuracy = tf.reduce_mean(correct_predictions)

    return accuracy
```

```
def validate(transformer, val_dataset,n_translations=5):
```

```
    # Store the generated sequences and the actual sequences
    generated_sequences = []
    actual_sequences = []
    samples = 0
    acc = 0.0
    loss = 0.0

    for (batch, ((inp, mask_inp), (tar, mask_tar))) in enumerate(
val_dataset):
        # Generate output using the model
        generated_sequence,avg_loss = generate_output(transformer,
inp, tar, mask_inp)
        generated_list = [int(tf.get_static_value(x)) for x in
generated_sequence]
        generated_sentence = sp_target.DecodeIds(generated_list)
        generated_sequences.append(generated_sentence.split())


        # Store the actual sequence (excluding the start token)
        actual_sequence = tar[:, 1:]  # Assuming the start token is
 at the beginning
        actual_list = [tf.get_static_value(x) for x in
actual_sequence]
        actual_sentence = sp_target.DecodeIds([int(token) for e in
actual_list for token in e])
        actual_sequences.append([actual_sentence.split()])

        samples += 1

        a = tar
        g = generated_sequence
        a_pad,g_pad = pad_to_match(tf.squeeze(tar),
generated_sequence)
        accuracy = calculate_translation_accuracy(g_pad,a_pad)
        acc += accuracy
        loss += avg_loss

        if samples < n_translations:
            print("src_id: " + str(generated_list) + "\ntar_id: " +
 str([1] + actual_list[0].tolist()))
            print("Pred: " + generated_sentence + "\nActual: " +
actual_sentence + "\n")

        if samples == 10000:
            break

    print(f"Translation Accuracy: {acc/samples * 100:.2f}%")
    print("Translation Loss: " + str(loss/samples))

    average_bleu = corpus_bleu(actual_sequences,
generated_sequences)

    return average_bleu
```

```python
def generate_output(transformer, input_sequence, tar, input_mask, start_token=1, end_token=2,max_len=40):
    # encoder_input is the input_sequence
    # decoder_input starts with only the start_token
    batch_size = tf.shape(input_sequence)[0]  # Get dynamic batch size from input_sequence
    output = tf.fill([batch_size, 1], start_token)

    total_loss = 0.0
    total_count = 0

    for i in range(max_len):  # MAX_LENGTH is the maximum length of the output sequence
        # Create look-ahead mask for the current output sequence
        look_ahead_mask = create_look_ahead_mask(tf.shape(output)[1])

        # Combine it with the target padding mask
        # Assuming target_mask is 1 for real tokens and 0 for padding
        target_padding_mask = tf.cast(tf.math.equal(output, 0), dtype=tf.int32)
        target_padding_mask = target_padding_mask[:, tf.newaxis, tf.newaxis, :]
        #combined_mask = tf.maximum(target_padding_mask, look_ahead_mask)

        # Call the transformer with input masks and the combined mask
        predictions,weights = transformer(input_sequence,
                                    output,
                                    False,
                                    input_mask,
                                    look_ahead_mask,
                                    target_padding_mask)

        #tf.print(np.shape(predictions))
        loss = loss_function(tar[:,min(i+1,tar.shape[1]-1)], predictions[:, -1, :])
        total_loss += loss.numpy()
        total_count += 1

        # Select the last word from the seq_len dimension
        predictions = predictions[:, -1:, :]  # shape now is [batch_size, 1, vocab_size]

        predicted_id = tf.cast(tf.argmax(predictions, axis=-1), tf.int32)

        # If the predicted_id is equal to the end token, return the result
        if predicted_id == end_token:
            output = tf.concat([output, predicted_id], axis=-1)
            return tf.squeeze(output, axis=0),total_loss /
```

```
    total_count

        # Concatenate the predicted_id to the output which is given to the decoder as its input.
        output = tf.concat([output, predicted_id], axis=-1)

    return tf.squeeze(output, axis=0),total_loss / total_count
```

```
@tf.function(reduce_retracing=True)
def train_step(inp, tar, mask_inp, mask_tar):
    tar_inp = tar[:,:-1]
    tar_real = tar[:, 1:]
    mask_tar_inp = mask_tar[:, :, :, :-1]
    mask_tar_real = mask_tar[:, :, :, 1:]
    seq_len = tf.shape(tar_inp)[1]
    look_ahead_mask = create_look_ahead_mask(seq_len)


    with tf.GradientTape() as tape:

        predictions,weights = transformer(inp,tar_inp,
    enc_padding_mask=mask_inp,look_ahead_mask=look_ahead_mask,
    dec_padding_mask=mask_tar_inp,training=True)
        loss = loss_function_smoothing(tar_real, predictions,
    smoothing_rate=0.1)
        accuracy = calculate_accuracy(predictions,tar_real)


    gradients = tape.gradient(loss, transformer.trainable_variables
    )
    optimizer.apply_gradients(zip(gradients, transformer.
    trainable_variables))
    return loss,accuracy

# Training loop
EPOCHS = 30  # Or your desired number of epochs

loss_log = open("loss.txt", "w")
acc_log = open("accuracy.txt", "w")
bleu_log = open("bleu.txt","w")

start = datetime.now()

start_time = start.strftime("%H:%M:%S")
print("Start Time =", start_time)

for epoch in range(EPOCHS):
    total_loss = 0.0
    total_accuracy = 0.0
    iteration = 0
    print("\n")
    for (batch, ((inp, mask_inp), (tar, mask_tar))) in enumerate(
    dataset):
        batch_loss,accuracy = train_step(inp, tar, mask_inp,
```

```
    mask_tar)
        total_loss += batch_loss
        total_accuracy += accuracy

        iteration += 1

        loss_log.write(str(batch_loss.numpy()) + '\n')
        acc_log.write(str(accuracy.numpy()) + '\n')

        if batch % 50 == 0:  # Print the batch loss every 50 batches
            print(f"Epoch {epoch + 1} Batch {batch} Loss {batch_loss.numpy():.4f} Accuracy {accuracy.numpy():.4f}")

    if (epoch + 1) % 5 == 0:

        avg_bleu = validate(transformer, valset)
        bleu_log.write(str(avg_bleu) + '\n')
        print(f"Average Validation BLEU Score for epoch: {epoch+1} is {avg_bleu:.4f}")

    if (epoch + 1) % 5 == 0:
        ckpt_save_path = ckpt_manager.save()
        print(f'Saving checkpoint for epoch {epoch+1} at {ckpt_save_path}')

    print(f"Epoch {epoch + 1} Loss {total_loss.numpy():.4f}")
    print(f"Epoch {epoch + 1} Average Loss {total_loss.numpy()/iteration:.4f} Average Accuracy {total_accuracy.numpy()/iteration:.4f}")
    loss_log.write("Avg: " + str(total_loss.numpy()/iteration) + '\n')
    acc_log.write("Avg: " + str(total_accuracy.numpy()/iteration) + '\n')

loss_log.close()
acc_log.close()
bleu_log.close()

end = datetime.now()

end_time = end.strftime("%H:%M:%S")
print("End Time =", end_time)

# get difference
delta = end - start

# time difference in seconds
print(f"Time difference is {delta.total_seconds()} seconds")
```

```

transformer.save('PATH\\model')
transformer = tf.keras.models.load_model('PATH')

```

```
avg_bleu_test = validate(transformer, testset, 20)
print(f"Average Test BLEU Score: {avg_bleu_test:.4f}")
```

# B Example attention layers 1 to 6

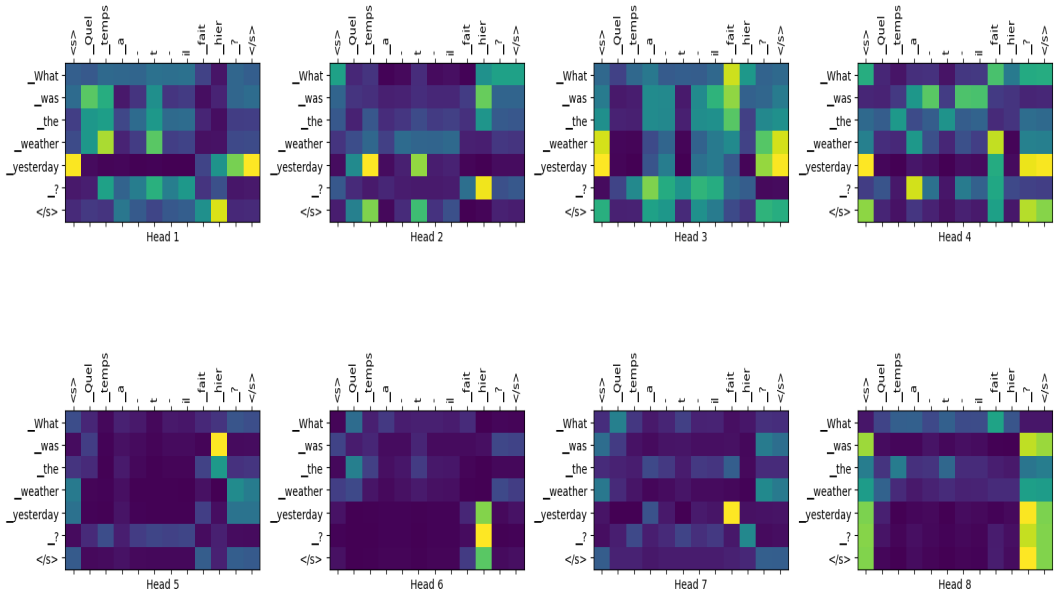



**Figure B1:** Attention of layer 1

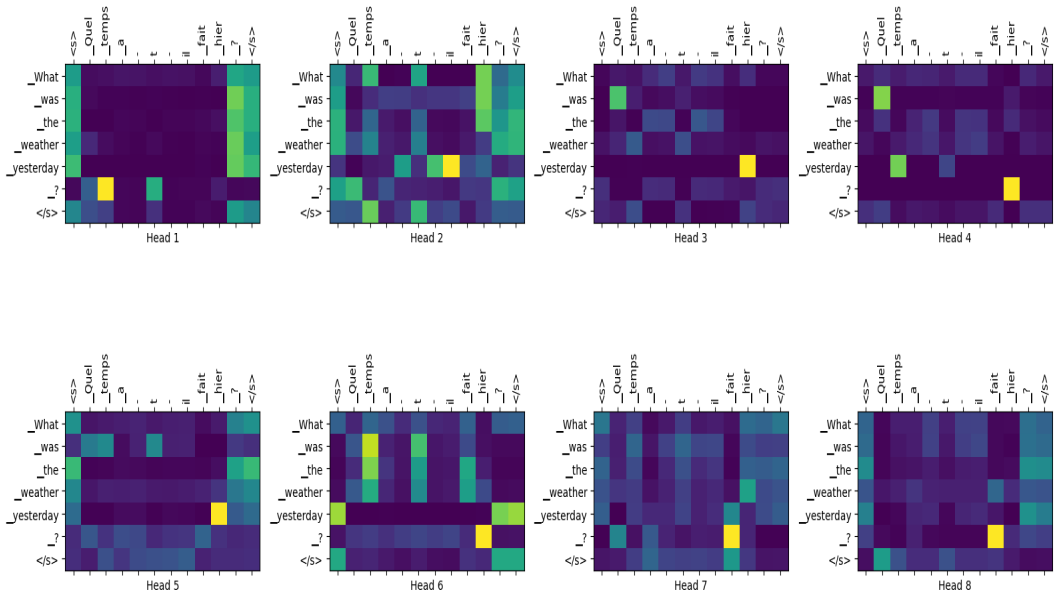



**Figure B2:** Attention of layer 2

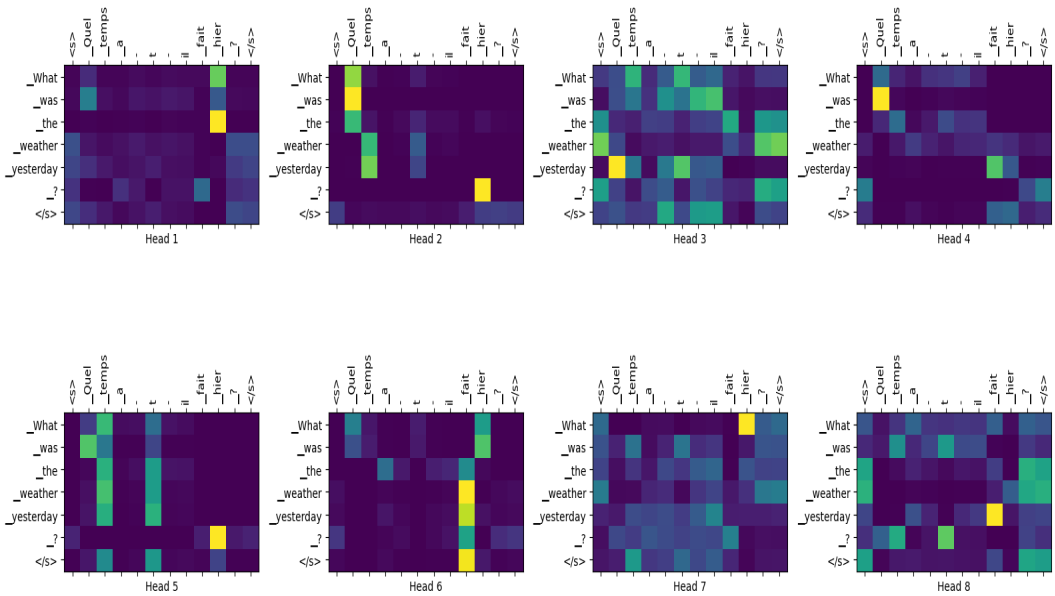



**Figure B3:** Attention of layer 3

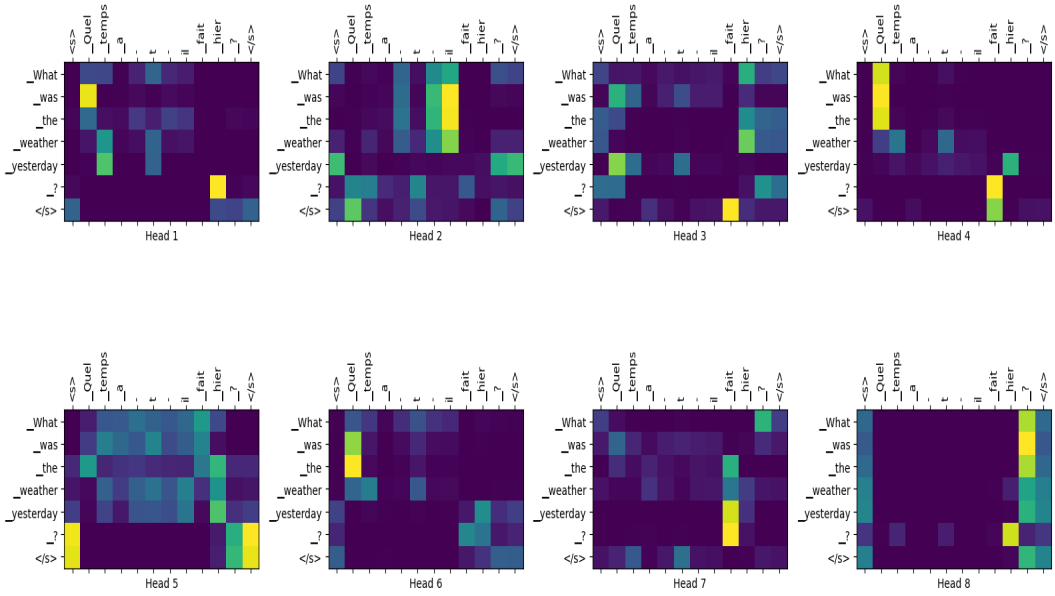



**Figure B4:** Attention of layer 4

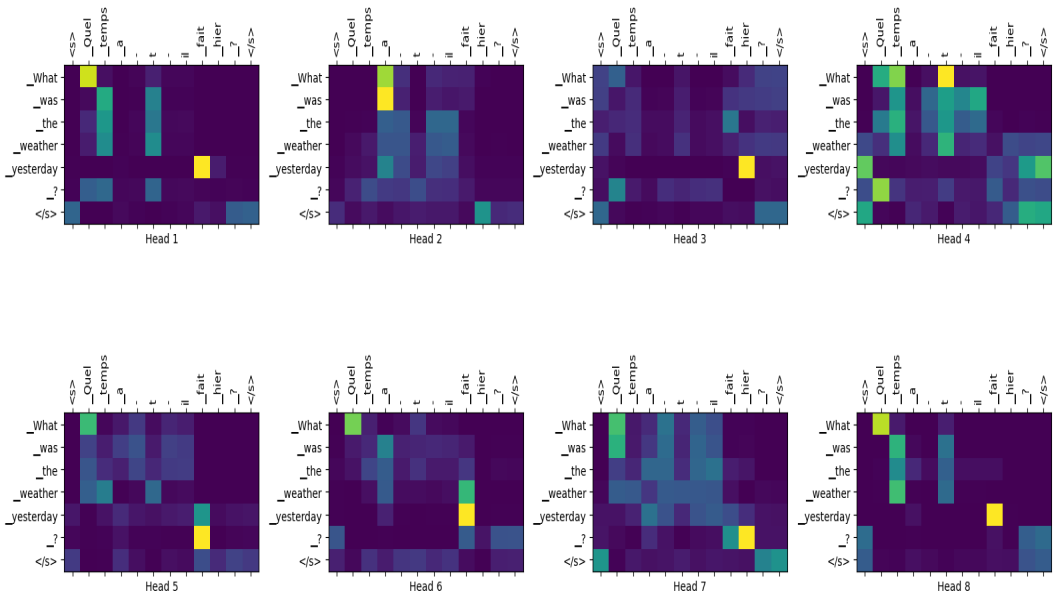



**Figure B5:** Attention of layer 5

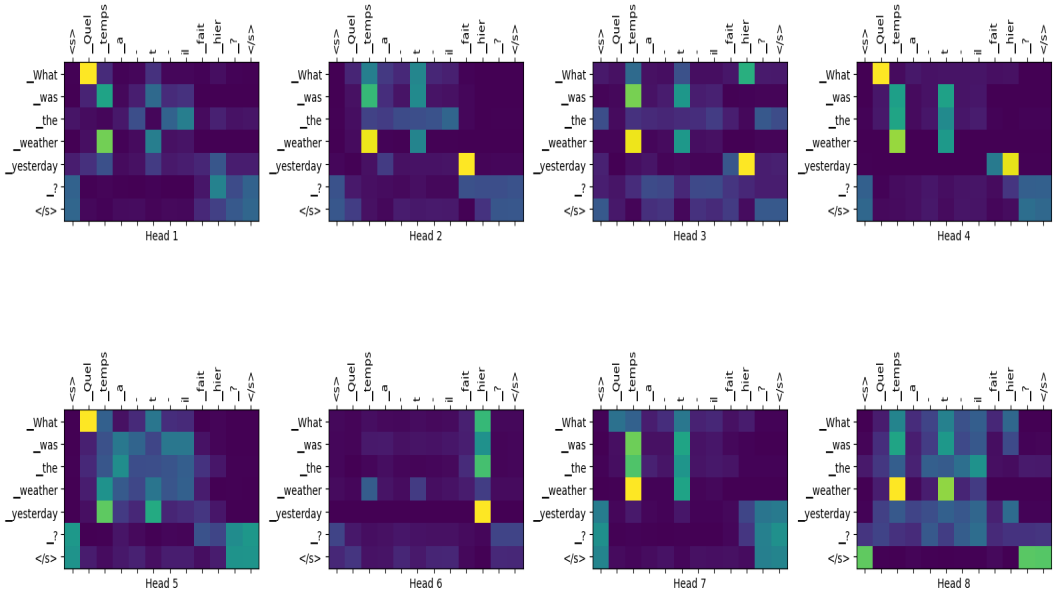



**Figure B6:** Attention of layer 6

# C Translation samples by the final model

Pred: Tom joined our group . [1, 57, 5118, 512, 3161, 4, 2]
Actual: Tom joined our group . [1, 57, 5118, 512, 3161, 4, 2]

Pred: It was raining . [1, 127, 81, 2079, 4, 2]
Actual: It was raining . [1, 127, 81, 2079, 4, 2]

Pred: I don't understand it at all . [1, 9, 98, 31967, 31948, 561, 93, 143, 158, 4, 2]
Actual: I don't understand this at all . [1, 9, 98, 31967, 31948, 561, 99, 143, 158, 4, 2]

Pred: He went to London yesterday . [1, 79, 492, 17, 2473, 568, 4, 2]
Actual: He left for London yesterday . [1, 79, 549, 89, 2473, 568, 4, 2]

Pred: We speak English all now . [1, 105, 452, 793, 158, 313, 4, 2]
Actual: English is spoken everywhere in the world now . [1, 793, 52, 2668, 3341, 59, 26, 1017, 313, 4, 2]

Pred: Forget what happened . [1, 4003, 151, 550, 4, 2]
Actual: Never mind what happened . [1, 2494, 708, 151, 550, 4, 2]

Pred: They will survive . [1, 177, 237, 3231, 4, 2]
Actual: They will survive . [1, 177, 237, 3231, 4, 2]

Pred: It's personal . [1, 127, 31967, 31952, 2411, 4, 2]
Actual: This is personal . [1, 193, 52, 2411, 4, 2]

Pred: I don't know how to use a . [1, 9, 98, 31967, 31948, 117, 366, 17, 718, 8, 4, 2]
Actual: I don't know how to operate a spinning wheel . [1, 9, 98, 31967, 31948, 117, 366, 17, 9892, 8, 11146, 4673, 4, 2]

Pred: How much does the shirt cost ? [1, 201, 337, 302, 26, 1959, 1210, 33, 2]
Actual: How much does the shirt cost ? [1, 201, 337, 302, 26, 1959, 1210, 33, 2]

Pred: He asked me out on a date . [1, 79, 582, 69, 204, 83, 8, 1847, 4, 2]
Actual: He asked me out on a date . [1, 79, 582, 69, 204, 83, 8, 1847, 4, 2]

Pred: I've got one for you . [1, 9, 31967, 22, 307, 225, 89, 16, 4, 2]
Actual: I've got one for you . [1, 9, 31967, 22, 307, 225, 89, 16, 4, 2]

Pred: I didn't meet with each other yesterday . [1, 9, 208, 31967, 31948, 581, 126, 981, 573, 568, 4, 2]
Actual: It was not until yesterday that I knew the news . [1, 127, 81, 111, 1091, 568, 66, 9, 674, 26, 964, 4, 2]

Pred: You need that . [1, 72, 211, 66, 4, 2]
Actual: You need this . [1, 72, 211, 99, 4, 2]

Pred: Nobody likes war . [1, 1084, 982, 812, 4, 2]
Actual: No one loves war . [1, 528, 225, 1702, 812, 4, 2]