A Survey on the Concepts behind Large Language Models

Master Thesis

Daniel Kofler, BSc.
2310876002
June, 2025

MASTER OF SCIENCE
Performed at the
Fachhochschule Kärnten / Carinthia University of Applied Sciences

Degree Program Applied Data Science

Supervised by

Dr. Peter Bachhiesl

Affidavit (Declaration of Originality)

I hereby declare that:

- I have independently written the presented Master thesis by myself;
- I have prepared this Master thesis without outside help and without using any sources or aids other than those cited by me; moreover, I have identified as such any passages taken verbatim or in terms of content from the sources used;
- in addition, I have fully indicated the use of generative AI models (e.g. ChatGPT) by specifying the product name and the reference source (e.g. URL);
- I have not used any other unauthorized aids and have consistently worked independently and when using generative AI models, I realize that I am responsible how the content will be used and to what extent
- I have not yet submitted this Master thesis in the same or similar form to any (other) educational institution as an examination performance or (scientific) thesis.
- I am aware that any violation (”use of unauthorized aids”) violates academic integrity and may result in (academic-related) legal consequences.

Gnesau, June 2, 2025

Place and Date

Student’s signature

Abstract

Following the success of ChatGPT and the exposure of artificial intelligence (AI) capabilities to the general public, this work intends to provide a formal description of the concepts behind said applications. It presents a mathematical description of the Transformer architecture introduced by Vaswani et al. and the training process of large language models (LLMs). It furthermore elaborates on why it was successful in addressing the challenges of recurrent neural networks (RNNs). Throughout this work, each component of a typical decoder-Transformer is detailed with mathematical formalisms, including gradient derivations. To support the theoretical framework, a library was developed and validated against PyTorch. It provides a transparent and readable implementation, while showing sufficient performance to train models with millions of parameters. Using this library, an LLM was successfully pre-trained on the Tiny Shakespeare dataset to demonstrate the learning capabilities of a Transformer model. This work serves as an introductory guide for mathematicians and computer scientists. All code is available on GitHub.

Keywords: Artificial Intelligence, Machine Learning, Natural Language Processing, Transformer, Large Language Model

Acknowledgements

I would like to express my gratitude to my project supervisor, Dr. Peter Bachhiesl. His patience, support and advice played an important role and ultimately led to the success of this work. Furthermore, I want to thank Andrej Karpathy for his influential work in AI and deep learning. The educational resources he provides on his YouTube channel were the initial inspiration for this project.

Contents

List of Figures

List of Tables

Convention

Scalars, Vectors, and Matrices

$x \in \mathbb{R}$	a real-valued scalar
$\boldsymbol{x} \in \mathbb{R}^m$	a column-vector with m real-valued components
$\boldsymbol{x}^\top \in \mathbb{R}^m$	a row-vector with m real-valued components
$\mathbf{1}^m$	a column vector of ones with m components
$\boldsymbol{X} \in \mathbb{R}^{m \times n}$	a matrix with m rows and n columns containing real-valued elements
$\boldsymbol{X}^\top \in \mathbb{R}^{n \times m}$	the transposed matrix $\boldsymbol{X}$
$\mathbf{1}^{m \times n}$	a matrix of ones with m rows and n columns

Indexing

$\boldsymbol{x}_i$	the i-th component of a vector $\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \dots \\ \boldsymbol{x}_m \end{bmatrix}$
$\boldsymbol{X}_{i,:}$	the i-th row of a matrix $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_{1,:} \\ \boldsymbol{X}_{2,:} \\ \vdots \\ \boldsymbol{X}_{m,:} \end{bmatrix}$
$\boldsymbol{X}_{:,j}$	the j-th column of a matrix $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_{:,1} & \boldsymbol{X}_{:,2} & \dots & \boldsymbol{X}_{:,n} \end{bmatrix}$
$\boldsymbol{X}_{i,j}$	the element in the i-th row and j-th column of a matrix $\boldsymbol{X}$

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_{1,1} & \boldsymbol{X}_{1,2} & \dots & \boldsymbol{X}_{1,n} \\ \boldsymbol{X}_{2,1} & \boldsymbol{X}_{2,2} & \dots & \boldsymbol{X}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{X}_{m,1} & \boldsymbol{X}_{m,2} & \dots & \boldsymbol{X}_{m,n} \end{bmatrix}$$

Sets

$\mathbb{A}$	a set
$\mathbb{R}$	the set of real numbers
$\{1, 2, \ldots, n\}$	the set of all integers between and including 1 and n
$[0, 1]$	the real interval including 0 and 1

Information and Probability Theory

X	a random variable
$\mathrm{E}\,(X)$	the expectation value of a random variable X
$\mathrm{Var}\,(X)$	the variance of a random variable X

Functions

$\exp(x)$	the exponential function evaluated at $x \in \mathbb{R}$
$\mathrm{erf}\,(x)$	the gaussian error function evaluated at $x \in \mathbb{R}$
$\log(x)$	the natural logarithmn of $x \in \mathbb{R}$

Calculus

$\frac{\partial y}{\partial x}$	the partial derivative of y with respect to x
$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$	the Kronecker delta

Chapter 1

Introduction

Since ChatGPT took the world by storm in late November 2022 it has transformed the public view of artificial intelligence. For the first time, AI was made available and usable for day-to-day tasks, leading to an increase in hype around the field and its applications. At the heart of large language models, such as GPT-4o by OpenAI[1], lies the Transformer neural network architecture developed by Vaswani et al. and described in their paper ”Attention is all you need” [1].

In their work the authors elaborate on how their newly proposed Scaled Dot-Product Attention mechanism can solve problems that limited recurrent neural networks, such as Long Short-Term Memory (LSTM) [4] models. So far, the Transformer architecture has stood the test of time and has yet to be dethroned.

1.1 Why are Transformers successful?

Many components of the model architecture are not new. Rather, it combines recent developments, such as Attention [5], Normalization [6,7] and Residual Connections [3] with concepts that have existed for decades, such as the Perceptron [8]. Furthermore, the parallelizable architecture of a Transformer leverages the computational power available today, forming a system that is capable of solving a wide range of problems.

Traditionally, when working on data with temporal dependence such as text, RNNs were the model of choice. Such a network processes sequence elements one at a time, passing information learned from previous elements to the next. This implies an inherent restriction of recurrent neural networks, as elements must be processed one after

[1] https://openai.com/

the other, and computations cannot be performed in parallel. Furthermore, because of this step-by-step processing, relevant information must be passed over long temporal distances and is therefore at risk of being lost. This effect increases with the input sequence length. Transformers mitigate these challenges as all input elements are processed simultaneously and in parallel by computing pairwise similarity scores between all elements. The risk of information loss due to long sequences is thereby removed, which allows one to scale such models to billions of parameters. As all operations are differentiable, the well-established backpropagation algorithm is used for training.

Additionally, as will be shown in this work, the matrix multiplication is the dominant operation in Transformer models. Optimizing hard- and software in this regard has thus been a priority of manufacturers. The sharp increase in computational capabilities of graphical processing units (GPUs) has allowed companies and research labs to train these models on a large scale [9].

1.2 Related Work

The number of works describing Transformer models grows by the day. While some books, such as [10, 11], offer a good overview of the architecture and also provide code examples, the mathematical details are often omitted, and existing programming libraries are relied upon to do gradient computation automatically. Papers, such as [12], focus on the description of algorithms, but again disregard mathematical details. Furthermore, there are countless blog posts, YouTube videos and GitHub repositories that aim to provide an intuitive explanation of Transformer models that again do not focus on details. It seems that current descriptions are mainly targeted at engineers and applications, rather than academia.

1.3 Problem Statement

As highlighted, despite the growing volume of material on Transformers and large language models, explanations are often vague or incomplete. Rarely are concepts described on a fundamental level and derivatives essential for training are usually omitted. This work aims to offer a complete formal introduction, supported by a Python reference implementation, to lower the entry barrier for both mathematicians and computer scientists. The focus is on the decoder-Transformer, the most relevant variant for modern LLMs.

Chapter 2

Tokenization

To make use of textual information for machine learning, it is necessary to encode it into a numerical representation. Text does not only include letters of the alphabet, but also numbers, punctuation and other characters. This encoding step forms the basis for extracting features from text, which is useful for natural language processing (NLP) tasks in general (not only large language models). Tokenization describes the idea of dividing text into units of information, called **tokens**, and subsequently replacing them with their unique **token ID** (an integer number). For this, a set of possible tokens (and token IDs), called **vocabulary** $\mathbb{V}$, must be defined first. Once outlined, it can be used to encode arbitrary text.

2.1 Obtaining a Token-Vocabulary

A token-vocabulary $\mathbb{V}$ is based on a given input text $s_i \in \mathbb{S}$, where $\mathbb{S}$ represents the set of all texts. To obtain tokens from the text, it is first split using a splitting function $f_{\text{split}} : \mathbb{S} \to \mathbb{V}$ into a vector of **tokens** $\boldsymbol{t}$ shown in Eq. (2.1). The set of unique tokens in $\boldsymbol{t}$ then represents the vocabulary $\mathbb{V} = \{v_1, v_2, \ldots, v_n\}$ with **vocabulary size** n.

$$f_{\text{split}}(s_i) = \boldsymbol{t} = \begin{bmatrix} \boldsymbol{t}_1 \\ \boldsymbol{t}_2 \\ \vdots \\ \boldsymbol{t}_m \end{bmatrix} \in \mathbb{V}^m \tag{2.1}$$

The token ID can be obtained from a token via an index function $I : \mathbb{V} \to \mathbb{N}$ shown in Eq. (2.2).

$$I\left(v_j\right) = j \tag{2.2}$$

To summarize, Fig. 2.1 shows the process of obtaining a vocabulary from a text.

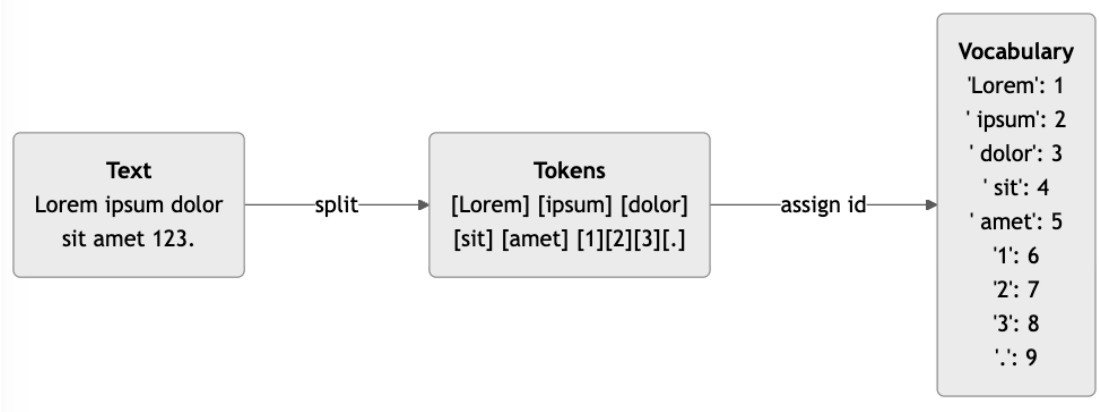

Figure 2.1: Typical tokenization process.

Once the vocabulary is obtained, it can be used to produce a vector of token IDs from any arbitrary text s_j via a tokenization function $f_{\text{token}} : \mathbb{S} \to \mathbb{N}$. This is done by first splitting a text into a vector of tokens using Eq. (2.1) and subsequently replacing each token by its corresponding token ID using Eq. (2.2).

$$\begin{gathered} \boldsymbol{x} = f_{\text{token}}\left(s_j\right) = I\left(f_{\text{split}}\left(s_j\right)\right) = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_m \end{bmatrix} \\ \boldsymbol{x} \in \mathbb{N}^m, s_j \in \mathbb{S} \end{gathered} \tag{2.3}$$

2.2 Text-Splitting Approaches

Different tokenization approaches vary in their strategy on how they split text into tokens. Subsequent machine learning models use the resulting token ID vectors as input. Transformer models scale quadratically with the number of tokens in this input vector (as they compute pairwise token similarities). It is therefore favorable to keep the number of tokens as low as possible and as high as necessary, depending on the application. Assuming that the maximum number of input tokens is given by the model, the splitting strategy of tokens represents an inherent trade-off:

- Choosing fewer characters to represent a token means each token carries less information, which leads to lower input complexity for the model. This reduces the vocabulary size and speeds up processing. However, with this approach, more tokens are required to represent the same text.
- In contrast, using multiple characters per token increases the information content of each token, allowing the model to process more meaning per step. However, this also expands the vocabulary, as it now contains combinations of characters. This may introduce challenges in handling unseen words if a particular combination of characters is not present in the vocabulary.

The trade-off impacts model efficiency, context understanding, and generalization. In the following sections, some of the most common approaches are described.

2.2.1 Character-level Tokenization

With a character-level tokenizer, single characters are considered tokens. This allows for the use of a predefined character set such as ASCII[1] to obtain the vocabulary $\mathbb{V} = \{v_i \mid v_i \in \text{ASCII}\}$.

[1] https://www.ascii-code.com/

2.2.2 Word-level Tokenization

Using entire words as tokens has the benefit of exposing the subsequent model to more information. This, however, implies a large vocabulary as words represent combinations of characters. Taking the Oxford English Dictionary[2], which contains 600,000 words, as an example, this means the vocabulary contains 600,000 unique tokens. Consequently, a machine learning model would have to compute probabilities for each token to predict the next one in a sequence. An alternative approach would be to use the n most frequent words in the English language and handle unknown words with a special token — often called the out-of-vocabulary (OOV) token. This strategy would then cover the English language, but most state-of-the-art models can handle multiple languages.

2.2.3 Sub-Word Tokenization

This approach represents a balance between token information content and vocabulary size. Common algorithms to achieve this include WordPiece [13] and Byte-Pair-Encoding (BPE) [14] (which is commonly used today). The core idea of BPE is to iteratively fuse the most frequently occurring character pairs (or byte pairs) in textual data to create new tokens. Using this approach, the resulting vocabulary contains character-based tokens, as well as tokens representing combinations of characters, depending on their occurrence in the text. A popular implementation using BPE is TikToken[3], which is used with OpenAI's large language models. It also comes with a visualization tool[4] to help understand the tokenization process, as shown in Figure 2.2.

[2] https://languages.oup.com/dictionaries/
[3] https://github.com/openai/tiktoken
[4] https://platform.openai.com/tokenizer

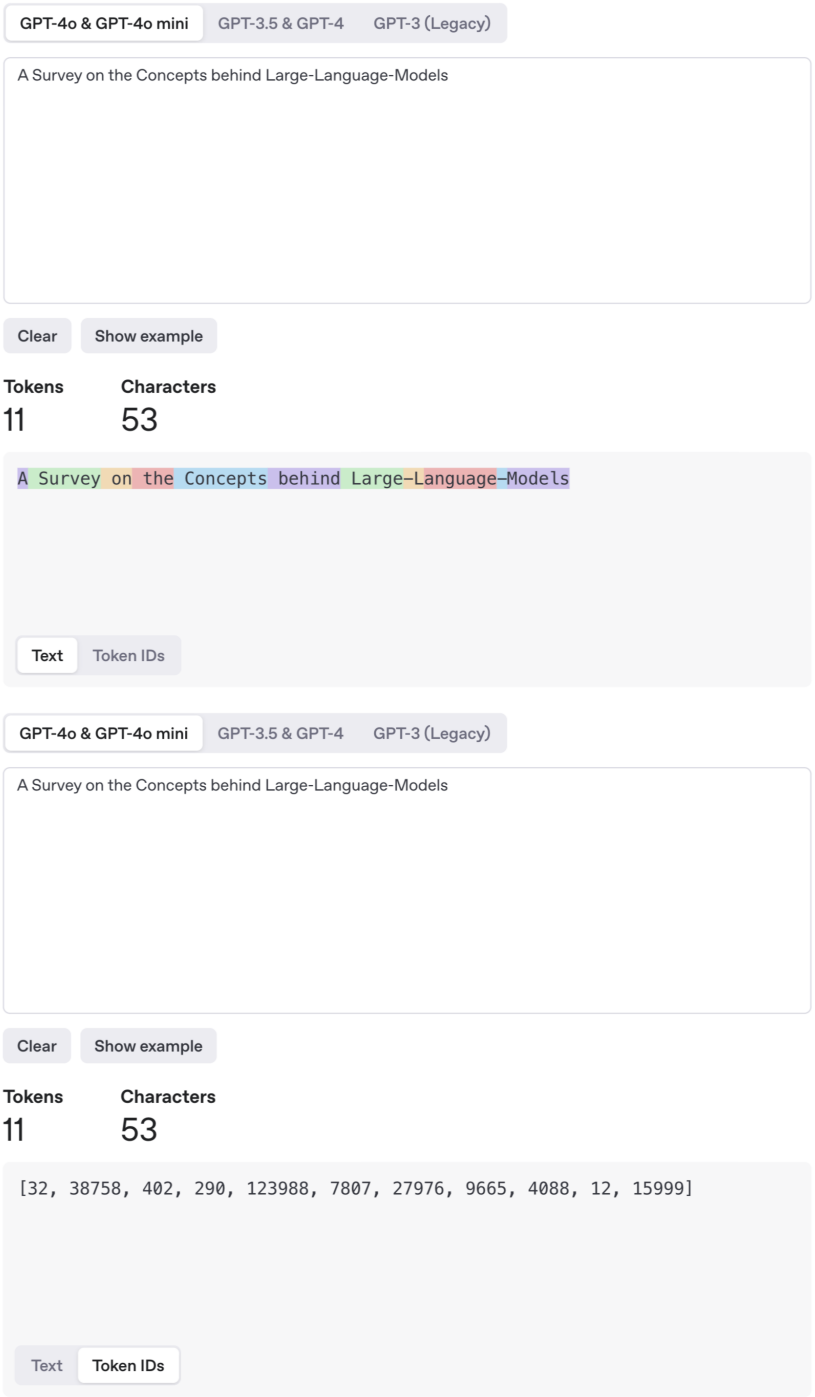

Figure 2.2: Example of how TikToken splits text into tokens. Top: shows how the text is split into tokens by color-coding each token. Bottom: shows the resulting token IDs.

Chapter 3

The Transformer

Vaswani et al. introduced the Transformer in their groundbreaking paper ”Attention is All You Need” [1]. This model architecture was originally invented for machine translation. The authors aimed for a model, which solely relied on the Attention mechanism and did not incorporate recurrent neurons or convolutions. They achieved this by introducing a new variant of the Attention mechanism and combining it with (at the time) novel ideas such as Layer Normalization [7] and Residual Connections [3]. Together with a layer of densely connected neurons this forms a Transformer block. Multiple such blocks are used in sequence, originally proposed as an encoder-decoder model depicted in Fig. 3.1. Furthermore, Dropout [15] may be applied throughout the model to improve generalization.

Since its introduction, different variants of the Transformer architecture have been developed for specific applications:

- **Encoder-Decoder-Transformers**: Following the original architecture, this Transformer variant combines an encoder block, which extracts features from the input, with a decoder block that uses these features for next-token prediction. This architecture has shown promising results in machine translation [1], summarization and speech recognition [16].
- **Encoder-Transformers**: This variant is used to extract features from a token sequence, which are then used for classification and other tasks. The most prominent implementation of an encoder-Transformer is BERT [17].
- **Decoder-Transformers**: The decoder variant is primarily used for generative purposes, such as large language models, and is the main focus of this work. Decoder-Transformers use causal masking to prevent tokens from accessing fu-

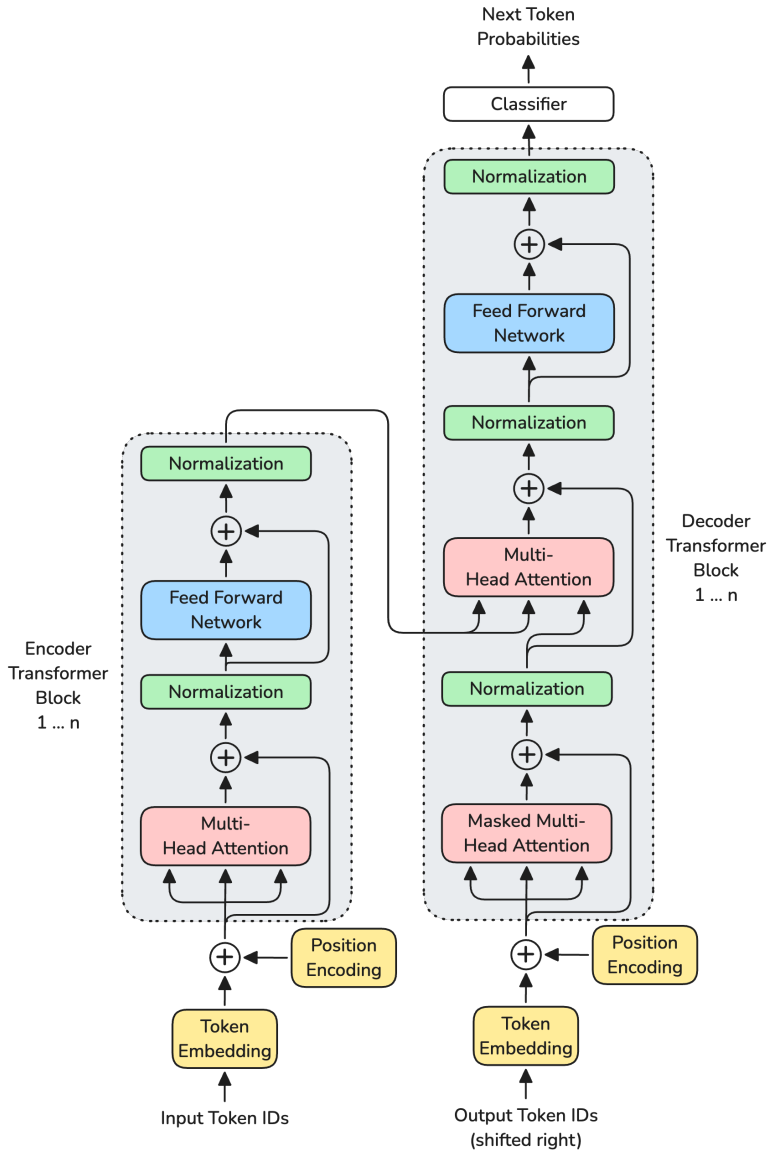

Figure 3.1: The Transformer architecture proposed by Vaswani et al. [1].

ture tokens, making them suitable for next-token prediction tasks, such as text generation. A well-known implementation is the GPT series by OpenAI [18–20]. This general architecture is shown in Fig. 3.2.

The following chapters detail the components of the decoder-Transformer shown in Fig. 3.2. Each section explains how data is transformed through the architecture and presents the corresponding operation derivatives, which are essential for training models, as later discussed in Ch. 4.

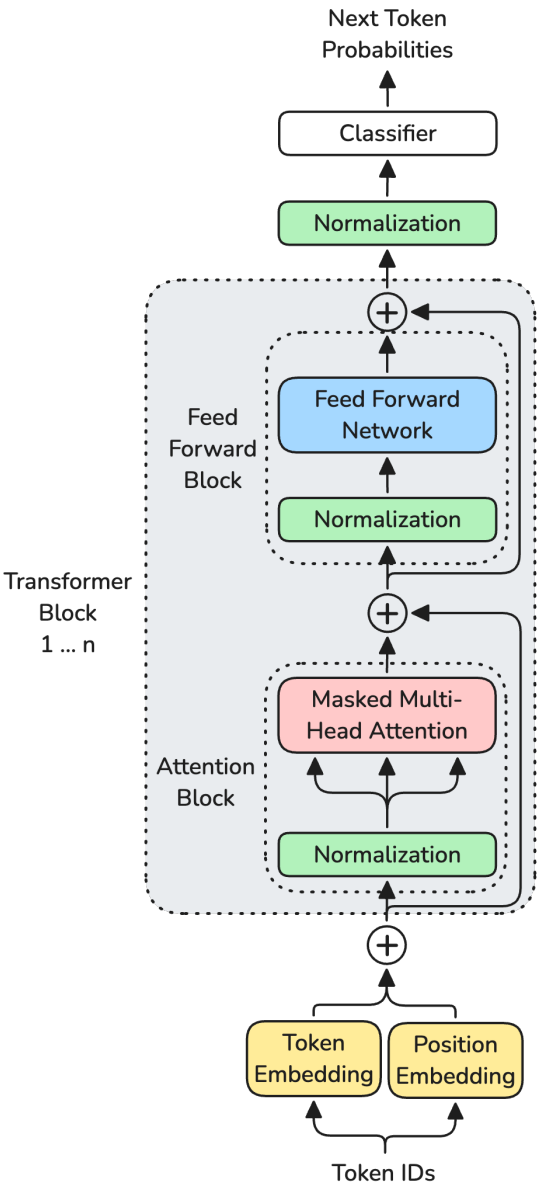

Figure 3.2: The decoder-Transformer architecture. As with the original architecture, multiple **Transformer blocks** can be used sequentially.

3.1 Embedding

The embedding components of Transformer models handle token embedding, as well as the addition of positional information to each token. Both are described in the following sections.

3.1.1 Token Embedding

Word embeddings have been used in NLP for decades. However, in 2013 they gained significant attention with the introduction of Word2Vec [21], which used an artificial neural network to learn numerical word representations. In essence, token embeddings are numerical vectors whose relative similarities (e.g. their Euclidean distance or cosine similarity) correlate with semantic similarity of the underlying tokens.

The process of word embedding can be understood as a lookup of embedding vectors from a matrix $\boldsymbol{W}^{\text{TE}} \in \mathbb{R}^{n \times d_{\text{model}}}$ (called **embedding table**) using token IDs as indices. This matrix contains n unique embedding vectors (one for each token in the vocabulary $\mathbb{V}$). Each of the vectors has dimensionality d_{model} (referred to as **model dimension**), which represents a model hyperparameter. Furthermore, the components of the embedding table are trainable model parameters, which are initialized randomly (usually drawing from a standard normal distribution).

For an input vector $\boldsymbol{x} \in \mathbb{N}^m$ containing m token IDs, the token embedding for token ID $\boldsymbol{x}_i$ corresponds to the embedding vector $\boldsymbol{W}^{\text{TE}}_{\boldsymbol{x}_i,:}$ as shown in Eq. (3.1).

$$\boldsymbol{X}^{\text{TE}} = \text{TokenEmbedding}\left(\boldsymbol{x}\right) = \begin{bmatrix} \boldsymbol{W}^{\text{TE}}_{\boldsymbol{x}_1,:} \\ \boldsymbol{W}^{\text{TE}}_{\boldsymbol{x}_2,:} \\ \vdots \\ \boldsymbol{W}^{\text{TE}}_{\boldsymbol{x}_m,:} \end{bmatrix} \tag{3.1}$$

$$\boldsymbol{X}^{\text{TE}} \in \mathbb{R}^{m \times d_{\text{model}}}, \boldsymbol{W}^{\text{TE}} \in \mathbb{R}^{n \times d_{\text{model}}}$$

Because this implies that $\boldsymbol{X}^{\text{TE}}_{i,j} = \boldsymbol{W}^{\text{TE}}_{\boldsymbol{x}_i,j}$, the corresponding derivatives are

$$\frac{\partial \boldsymbol{X}^{\text{TE}}_{i,j}}{\partial \boldsymbol{W}^{\text{TE}}_{p,q}} = \delta_{\boldsymbol{x}_i p} \delta_{jq}$$

for $i, p = 1, \ldots, m$ and $j, q = 1, \ldots, d_{\text{model}}$.

3.1.2 Position-Encoding

By only relying on the Attention mechanism and avoiding recurrence and convolution in the Transformer architecture, the ordering of tokens is not present for the model to process. Vaswani et al. [1] therefore added a fixed "positional encoding" to token embeddings in their original proposal, while referring to the work of Gehring et al. [22]. However, more modern LLMs, such as the GPT series [18–20], make use of a learned positional embedding.

In contrast to word embedding, where embedding vectors are selected using the token ID, in learned positional embedding, the first m rows of a matrix $\boldsymbol{W}^{\text{PE}} \in \mathbb{R}^{c \times d_{\text{model}}}$ are used. The matrix contains c unique positional embedding vectors, where c represents a hyperparameter and corresponds to the maximum number of tokens the Transformer can process (referred to as **maximum context size**). For positional embeddings, the same number of embedding dimensions d_{model} is used, such that word- and positional embedding vectors can be summed. The components of $\boldsymbol{W}^{\text{PE}}$ again represent learnable model parameters, which are initialized randomly.

For an input vector $\boldsymbol{x} \in \mathbb{N}^m$ containing m token IDs with $m \leq c$, the positional embedding for the i-th token corresponds to the embedding vector $\boldsymbol{W}^{\text{PE}}_{i,:}$, as shown in Eq. (3.2).

$$\boldsymbol{X}^{\text{PE}} = \text{PositionEmbedding}\left(\boldsymbol{x}\right) = \begin{bmatrix} \boldsymbol{W}^{\text{PE}}_{1,:} \\ \boldsymbol{W}^{\text{PE}}_{2,:} \\ \vdots \\ \boldsymbol{W}^{\text{PE}}_{m,:} \end{bmatrix} \tag{3.2}$$

$$\boldsymbol{X}^{\text{PE}} \in \mathbb{R}^{m \times d_{\text{model}}}, \boldsymbol{W}^{\text{PE}} \in \mathbb{R}^{c \times d_{\text{model}}}$$

Again, since $\boldsymbol{X}^{\text{PE}}_{i,j} = \boldsymbol{W}^{\text{PE}}_{i,j}$, the corresponding derivatives are

$$\frac{\partial \boldsymbol{X}^{\text{PE}}_{i,j}}{\partial \boldsymbol{W}^{\text{PE}}_{p,q}} = \delta_{ip}\delta_{jq}$$

for $i, p = 1, \ldots, m$ and $j, q = 1, \ldots, d_{\text{model}}$.

3.1.3 Embedding Addition

Once both embeddings are computed, their matrices are added as described in Eq. (3.3), resulting in the final token embeddings $\boldsymbol{X}^{\mathrm{E}}$.

$$\boldsymbol{X}^{\mathrm{E}} = \boldsymbol{X}^{\mathrm{TE}} + \boldsymbol{X}^{\mathrm{PE}} \qquad \boldsymbol{X}^{\mathrm{E}} \in \mathbb{R}^{m \times d_{\text{model}}} \tag{3.3}$$

Since embeddings are simply added, the derivatives are

$$\frac{\partial \boldsymbol{X}^{\mathrm{E}}}{\partial \boldsymbol{X}^{\mathrm{TE}}} = \mathbf{1}^{m \times d_{\text{model}}}$$
$$\frac{\partial \boldsymbol{X}^{\mathrm{E}}}{\partial \boldsymbol{X}^{\mathrm{PE}}} = \mathbf{1}^{m \times d_{\text{model}}}$$

.

3.2 Normalization

Normalization layers occur multiple times throughout the Transformer and play a vital role in the training of these models. They allow one to train deep neural networks by avoiding the problem of vanishing and exploding gradients [23]. There exist several approaches for implementing normalization in neural networks, such as Batch Normalization [6] and **Layer Normalization** [7]. The GPT series and the original Transformer architecture use the latter to scale embedding vectors and will therefore be covered in this section.

Vaswani et al. applied Layer Normalization after each Attention- and feed-forward block. This approach is referred to as **Post-Layer Normalization**. Later, it has become more common to use a normalization layer before applying the Attention mechanism and the feed-forward block as depicted in Fig. 3.3. This so-called **Pre-Layer Normalization** was shown to benefit the training process by Xiong et al. [2].

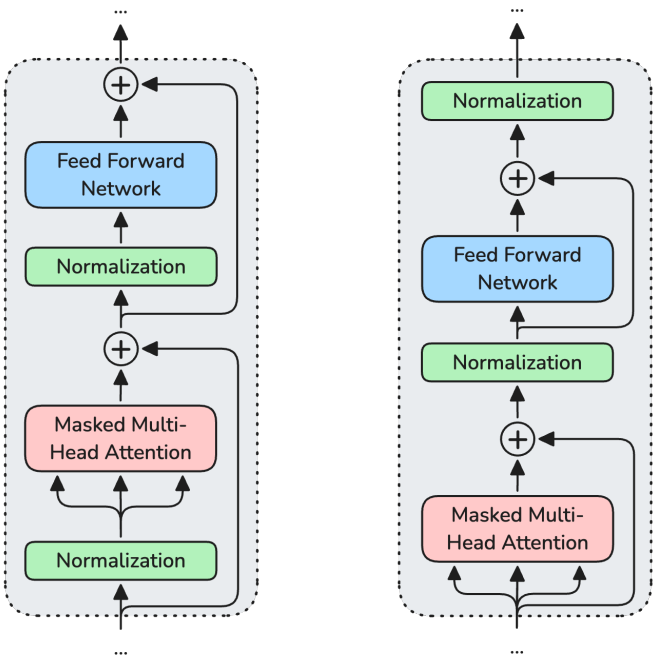

Figure 3.3: According to Xiong et al. [2], Pre-Layer Normalization (left) outperforms Post-Layer Normalization (right) by accelerating Transformer training through *"well-behaved gradients"*.

In Transformer models, each token embedding vector $\boldsymbol{X}_{i,:}$ of a matrix $\boldsymbol{X} \in \mathbb{R}^{m \times d_{\text{model}}}$ is normalized independently using Eq. (3.4). Two learnable parameter vectors $\boldsymbol{\gamma}$ (initialized as $\mathbf{1}$) and $\boldsymbol{\beta}$ (initialized as $\mathbf{0}$) are introduced. These are shared between tokens and allow the model to learn a scaling and shifting of the output distribution.

$$\begin{gathered}\boldsymbol{X}_{i,j}^{\mathrm{LN}}=\operatorname{Layernorm}\left(\boldsymbol{X}_{i,j}\right)=\boldsymbol{\gamma}_j\cdot\frac{\boldsymbol{X}_{i,j}-\boldsymbol{\mu}_i}{\boldsymbol{\sigma}_i}+\boldsymbol{\beta}_j\\ \boldsymbol{\gamma},\boldsymbol{\beta}\in\mathbb{R}^{d_{\text{model}}},\boldsymbol{X}^{\mathrm{LN}}\in\mathbb{R}^{m\times d_{\text{model}}}\end{gathered}\tag{3.4}$$

The statistics for the i-th token embedding vector are computed as

$$\boldsymbol{\mu}_i=\frac{1}{d_{\text{model}}}\sum_{j=1}^{d_{\text{model}}}\boldsymbol{X}_{i,j}\qquad\boldsymbol{\sigma}_i=\sqrt{\frac{1}{d_{\text{model}}}\sum_{j=1}^{d_{\text{model}}}\left(\boldsymbol{X}_{i,j}-\boldsymbol{\mu}_i\right)^2}$$

.

The corresponding derivatives are

$$\begin{aligned}\frac{\partial\boldsymbol{X}_{i,j}^{\mathrm{LN}}}{\partial\boldsymbol{X}_{p,q}}&=\delta_{ip}\frac{\boldsymbol{\gamma}_j}{\boldsymbol{\sigma}_i}\\ \frac{\partial\boldsymbol{X}_{i,j}^{\mathrm{LN}}}{\partial\boldsymbol{\mu}_p}&=-\delta_{ip}\frac{\boldsymbol{\gamma}_j}{\boldsymbol{\sigma}_i}\\ \frac{\partial\boldsymbol{X}_{i,j}^{\mathrm{LN}}}{\partial\boldsymbol{\sigma}_p}&=-\delta_{ip}\boldsymbol{\gamma}_j\cdot\frac{\boldsymbol{X}_{i,j}-\boldsymbol{\mu}_i}{\boldsymbol{\sigma}_i^2}\\ \frac{\partial\boldsymbol{X}_{i,j}^{\mathrm{LN}}}{\partial\boldsymbol{\gamma}_p}&=\delta_{jp}\frac{\boldsymbol{X}_{i,j}-\boldsymbol{\mu}_i}{\boldsymbol{\sigma}_i}\\ \frac{\partial\boldsymbol{X}_{i,j}^{\mathrm{LN}}}{\partial\boldsymbol{\beta}_p}&=\delta_{jp}\end{aligned}$$

and using $\boldsymbol{S}=\frac{1}{d_{\text{model}}}\sum_{j=1}^{d_{\text{model}}}\left(\boldsymbol{X}_{i,j}-\boldsymbol{\mu}_i\right)^2$, the statistics derivatives are

$$\begin{aligned}\frac{\partial\boldsymbol{\mu}_i}{\partial\boldsymbol{X}_{p,q}}&=\frac{\delta_{ip}}{d_{\text{model}}}\\ \frac{\partial\boldsymbol{\sigma}_i}{\partial\boldsymbol{X}_{p,q}}&=\delta_{ip}\cdot\frac{\partial\boldsymbol{\sigma}_i}{\partial\boldsymbol{S}}\cdot\frac{\partial\boldsymbol{S}}{\partial\boldsymbol{X}_{p,q}}\\ &=\delta_{ip}\cdot\frac{1}{2\sqrt{\boldsymbol{S}}}\cdot\frac{1}{d_{\text{model}}}\sum_{j=1}^{d_{\text{model}}}2\left(\boldsymbol{X}_{i,j}-\boldsymbol{\mu}_i\right)\cdot\left(\delta_{jq}-\frac{\partial\boldsymbol{\mu}_i}{\partial\boldsymbol{X}_{p,q}}\right)\\ &=\frac{\delta_{ip}}{\boldsymbol{\sigma}_i d_{\text{model}}}\sum_{j=1}^{d_{\text{model}}}\left(\boldsymbol{X}_{i,j}-\boldsymbol{\mu}_i\right)\cdot\left(\delta_{jq}-\frac{\partial\boldsymbol{\mu}_i}{\partial\boldsymbol{X}_{p,q}}\right)\end{aligned}$$

for $i,p=1,\ldots,m$ and $j,q=1,\ldots,d_{\text{model}}$.

3.3 Residual Connections

Besides Layer Normalization, another measure to improve the learning procedure in Transformers is the use of **residual connections** (also called **skip connections**). The idea behind these was introduced by He et al. [3]. The authors showed, how the use of skip connections can improve the training time and performance of deep neural networks.

This is achieved by adding the input to a block of layers to its output, as illustrated in Fig. 3.4. In Transformers, this block of layers corresponds to the Mulit-Head Attention block and the feed-forward block, as depicted earlier in Fig. 3.2. By using residual connections, layers do not need to preserve useful information from their input and can therefore emphasize on learning new, residual information (He et al. call this *"residual learning"*). Furthermore, if a layer turns out to be redundant during optimization, it can be eliminated by shrinking its weights towards zero.

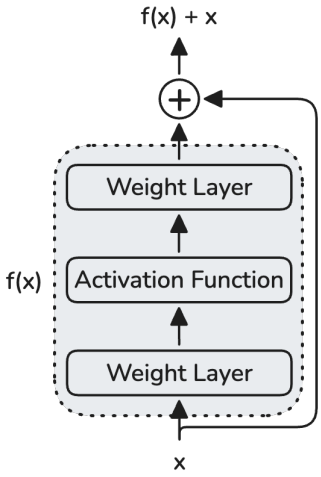

Figure 3.4: The residual learning block used in the work by He et al. [3]. Adding the input of a weight layer stack to its output allows it to effectively learn residual information.

Because Transformers process token embeddings, by adding the input to a layer back to its output, the semantic meaning of a token embedding is updated depending on the context of other tokens. This can be thought of as tokens following a path in $\mathbb{R}^{d_{\text{model}}}$ as they are processed in each layer.

In Transformers, the input $\boldsymbol{X} \in \mathbb{R}^{m \times d_{\text{model}}}$ to the Attention block and feed-forward block (as denoted in Fig. 3.2) is added back to their output. This idea is described in Eq. (3.5) where blocks are represented by a function $f_{\text{Block}} : \mathbb{R}^{m \times d_{\text{model}}} \rightarrow \mathbb{R}^{m \times d_{\text{model}}}$.

$$\boldsymbol{X}^{\mathrm{RC}} = f_{\mathrm{Block}}\left(\boldsymbol{X}\right) + \boldsymbol{X} \qquad \boldsymbol{X}^{\mathrm{RC}} \in \mathbb{R}^{m \times d_{\mathrm{model}}} \tag{3.5}$$

The corresponding derivatives are

$$\frac{\partial \boldsymbol{X}^{\mathrm{RC}}}{\partial \boldsymbol{X}} = \frac{\partial}{\partial \boldsymbol{X}} f_{\mathrm{Block}}\left(\boldsymbol{X}\right) + \mathbf{1}^{m \times d_{\mathrm{model}}}$$

.

3.4 Masked Multi-Head Self-Attention (MHSA)

The idea of the Attention mechanism, as introduced by Bahdanau et al. [5] in 2014, lies in allowing input elements to interact and learn which other elements are relevant in a given context. The authors suspected that *"the use of a fixed-length vector is a bottleneck in improving the performance of a basic encoder–decoder architecture"* and therefore proposed to *"allow a model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word"*.

For Transformer models, Vaswani et al. proposed an Attention variant, which they called **Masked Multi-Head Self-Attention**. It enables a Transformer to learn, which inputs are relevant to make predictions for the next token. By computing pairwise similarities between all input tokens, the model simultaneously generates predictions for **all** m **tokens in the input**. This mechanism is explained in the following sections.

3.4.1 Using Multiple Attention Heads

In Multi-Head Attention, the computations happen in multiple independent **Attention heads** in parallel. Vaswani et al. [1] *"found it to be beneficial"* and claim that *"Multi-Head Attention allows the model to jointly attend to information from different representation subspaces at different positions"*.

The normalized input $\boldsymbol{X}^{\text{LN}} \in \mathbb{R}^{m \times d_{\text{model}}}$ is split into h smaller, equally sized matrices, where h represents the number of Attention heads (a hyperparameter). This partitioning is done along the d_{model}-dimension, splitting the $m \times d_{\text{model}}$ input matrix into multiple $m \times \frac{d_{\text{model}}}{h}$ chunks, as shown in Eq. (3.6) and Fig. 3.5. This new, smaller dimension is often called **head dimension** $d_{\text{head}} = \frac{d_{\text{model}}}{h}$. This also implies that d_{model} must be divisible by h.

$$\begin{gathered} \boldsymbol{X}^{\text{H},i} = \begin{bmatrix} \boldsymbol{X}^{\text{LN}}_{:,(i-1)\cdot d_{\text{head}}+1} & \boldsymbol{X}^{\text{LN}}_{:,(i-1)\cdot d_{\text{head}}+2} & \cdots & \boldsymbol{X}^{\text{LN}}_{:,i\cdot d_{\text{head}}} \end{bmatrix} \\ \boldsymbol{X}^{\text{H},i} \in \mathbb{R}^{m \times d_{\text{head}}}, i = 1, \ldots, h \end{gathered} \tag{3.6}$$

When training the Transformer model, the matrices containing the derivatives for each head are concatenated, which is structurally analogous to Eq. (3.14).

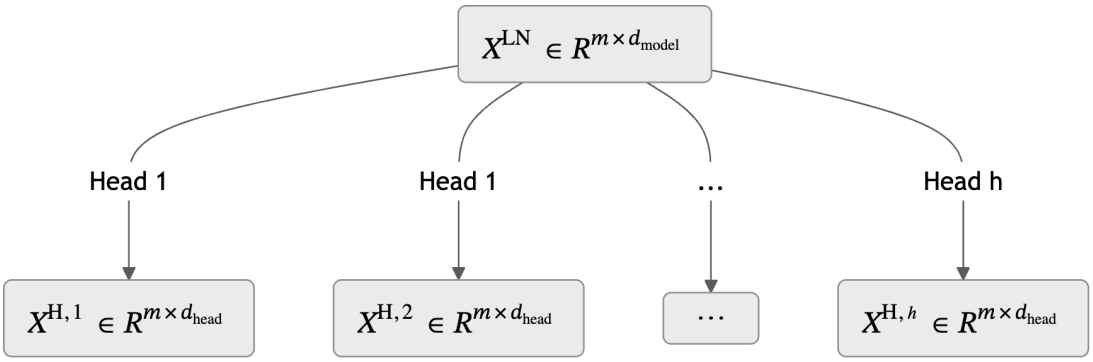

Figure 3.5: In **Multi-Head Attention** the normalized input is split into h equally sized matrix chunks for each head.

3.4.2 Query, Key and Value Vectors

For the Attention mechanism, the computation of pairwise similarities requires **query, key and value** vectors for each token. In this context, **self-Attention** means, all three of these vectors are created from the same input token embedding vector, as depicted in Fig. 3.6. This is not the case in encoder-decoder-Transformers. These vectors are obtained through linear transformations of token embeddings using the weight matrices $\boldsymbol{W}^{\mathrm{Q},i}$, $\boldsymbol{W}^{\mathrm{K},i}$ and $\boldsymbol{W}^{\mathrm{V},i}$ (for the i-th Attention head). The elements of these matrices represent learnable parameters which are initialized randomly.

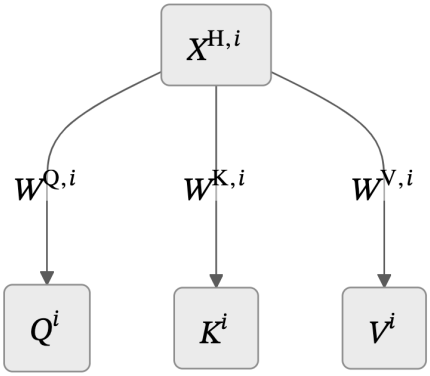

Figure 3.6: In **self-Attention**, query, key, and value vectors are generated from the same token embedding vector.

Eqs. (3.7) to (3.9) describe the linear transformations of a head input matrix $\boldsymbol{X}^{\mathrm{H},i} \in \mathbb{R}^{m \times d_{\text{head}}}$ to query, key and value matrices for the i-th Attention head.

$$\boldsymbol{Q}^i = \boldsymbol{X}^{\mathrm{H},i}\boldsymbol{W}^{\mathrm{Q},i} \qquad \boldsymbol{Q}^i \in \mathbb{R}^{m \times d_{\text{head}}}, \boldsymbol{W}^{\mathrm{Q},i} \in \mathbb{R}^{d_{\text{head}} \times d_{\text{head}}} \tag{3.7}$$

$$\boldsymbol{K}^i = \boldsymbol{X}^{\mathrm{H},i}\boldsymbol{W}^{\mathrm{K},i} \qquad \boldsymbol{K}^i \in \mathbb{R}^{m \times d_{\text{head}}}, \boldsymbol{W}^{\mathrm{K},i} \in \mathbb{R}^{d_{\text{head}} \times d_{\text{head}}} \tag{3.8}$$

$$\boldsymbol{V}^i = \boldsymbol{X}^{\mathrm{H},i}\boldsymbol{W}^{\mathrm{V},i} \qquad \boldsymbol{V}^i \in \mathbb{R}^{m \times d_{\text{head}}}, \boldsymbol{W}^{\mathrm{V},i} \in \mathbb{R}^{d_{\text{head}} \times d_{\text{head}}} \tag{3.9}$$

The corresponding derivatives are

$$\begin{aligned}
\frac{\partial \boldsymbol{Q}^i_{j,k}}{\partial \boldsymbol{X}^{\mathrm{H},i}_{p,q}} &= \delta_{jp}\boldsymbol{W}^{\mathrm{Q},i}_{q,k} \\
\frac{\partial \boldsymbol{K}^i_{j,k}}{\partial \boldsymbol{X}^{\mathrm{H},i}_{p,q}} &= \delta_{jp}\boldsymbol{W}^{\mathrm{K},i}_{q,k} \\
\frac{\partial \boldsymbol{V}^i_{j,k}}{\partial \boldsymbol{X}^{\mathrm{H},i}_{p,q}} &= \delta_{jp}\boldsymbol{W}^{\mathrm{V},i}_{q,k} \\
\frac{\partial \boldsymbol{Q}^i_{j,k}}{\partial \boldsymbol{W}^{\mathrm{Q},i}_{p,q}} = \frac{\partial \boldsymbol{K}^i_{j,k}}{\partial \boldsymbol{W}^{\mathrm{K},i}_{p,q}} &= \frac{\partial \boldsymbol{V}^i_{j,k}}{\partial \boldsymbol{W}^{\mathrm{V},i}_{p,q}} = \delta_{kq}\boldsymbol{X}^{\mathrm{H},i}_{j,p}
\end{aligned}$$

for $j, p = 1, \ldots, m$ and $k, q = 1, \ldots, d_{\text{head}}$.

3.4.3 Scaled Dot-Product Attention

To actually compute similarities, Vaswani et al. proposed **Scaled Dot-Product Attention** (with optional masking for decoder-Transformers) to compute **Attention scores**. In the following section, the ideas behind the proposed mechanism are described.

3.4.3.1 Token-Similarities

First, token similarities are computed via a dot product between every token's query and key vectors.

$$\boldsymbol{Q}^i \boldsymbol{K}^{i,\top} = \begin{bmatrix} \boldsymbol{Q}^i_{1,:} \\ \boldsymbol{Q}^i_{2,:} \\ \vdots \\ \boldsymbol{Q}^i_{m,:} \end{bmatrix} \begin{bmatrix} \boldsymbol{K}^{i,\top}_{1,:} & \boldsymbol{K}^{i,\top}_{2,:} & \dots & \boldsymbol{K}^{i,\top}_{m,:} \end{bmatrix} \qquad \boldsymbol{Q}^i \boldsymbol{K}^{i,\top} \in \mathbb{R}^{m \times m}$$

The derivatives computed as

$$\frac{\partial \left[\boldsymbol{Q}^i \boldsymbol{K}^{i,\top} \right]_{j,k}}{\partial \boldsymbol{Q}^i_{p,q}} = \delta_{jp} \boldsymbol{K}^i_{k,q}$$
$$\frac{\partial \left[\boldsymbol{Q}^i \boldsymbol{K}^{i,\top} \right]_{j,k}}{\partial \boldsymbol{K}^i_{p,q}} = \delta_{jp} \boldsymbol{Q}^i_{j,q}$$

for $j, k, p = 1, \dots, m$ and $q = 1, \dots, d_{\text{head}}$.

3.4.3.2 Scaling

Next, each component of the token-similarity matrix $\boldsymbol{Q}^i \boldsymbol{K}^{i,\top}$ is scaled by $\frac{1}{\sqrt{d_{\text{head}}}}$. This is done, because large absolute values can cause problems with vanishing and exploding gradients [23]. Vaswani et al. [1] argue their choice for the scaling factor by first assuming that for a query vector $\boldsymbol{q}$ and key vector $\boldsymbol{k}$, their components $\boldsymbol{q}_i, \boldsymbol{k}_i$ are realizations of random variables

$$Q, K \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, 1) .$$

Then the expectation of the dot product remains

$$\mathrm{E}(\boldsymbol{q} \cdot \boldsymbol{k}) = \mathrm{E}\left(\sum_{i=1}^{d_{\text{head}}} \boldsymbol{q}_i \cdot \boldsymbol{k}_i \right) = \sum_{i=1}^{d_{\text{head}}} \mathrm{E}(\boldsymbol{q}_i) \cdot \mathrm{E}(\boldsymbol{k}_i) = 0$$

but the variance grows to

$$\mathrm{Var}(\boldsymbol{q} \cdot \boldsymbol{k}) = \mathrm{Var}\left(\sum_{i=1}^{d_{\text{head}}} \boldsymbol{q}_i \cdot \boldsymbol{k}_i \right) = \sum_{i=1}^{d_{\text{head}}} \mathrm{Var}(\boldsymbol{q}_i \cdot \boldsymbol{k}_i) = d_{\text{head}}$$

with the variance of the products described by Goodman [24] as

$$\mathrm{Var}\left(\boldsymbol{q}_i \cdot \boldsymbol{k}_i\right) = \underbrace{\left[\mathrm{E}(\boldsymbol{q}_i)\right]^2 \mathrm{Var}\left(\boldsymbol{k}_i\right)}_{0} + \underbrace{\left[\mathrm{E}\left(\boldsymbol{k}_i\right)\right]^2 \mathrm{Var}\left(\boldsymbol{q}_i\right)}_{0} + \underbrace{\mathrm{Var}\left(\boldsymbol{q}_i\right) \mathrm{Var}\left(\boldsymbol{k}_i\right)}_{1} = 1$$

The scaling can therefore be interpreted as a normalization by subtracting the mean 0 and dividing by the standard deviation $\sqrt{d_{\text{head}}}$. The corresponding derivatives of the scaling operation are

$$\frac{\partial \boldsymbol{X}^{\mathrm{S},i}_{j,k}}{\partial \left[\boldsymbol{Q}^i \boldsymbol{K}^{i,\top}\right]_{p,q}} = \frac{1}{\sqrt{d_{\text{head}}}} \cdot \mathbf{1}^{m \times m}$$

for $j, k, p, q = 1, \ldots, m$.

3.4.3.3 Mask

As mentioned, decoder-Transformers produce predictions for all m tokens of the input. Therefore, it is necessary to apply a **causal mask** to the scaled token-similarity matrix, which prevents tokens from aggregating information from other tokens that occur **later** in the input (future tokens). This can be achieved by first setting the corresponding scaled token-similarity to $-\infty$ according to Eq. (3.10).

$$\mathrm{Mask}\left(\boldsymbol{X}_{i,j}\right) = \begin{cases} -\infty & \text{if } i < j \\ \boldsymbol{X}_{i,j} & \text{otherwise} \end{cases} \qquad i, j = 1, \ldots, m \tag{3.10}$$

By applying the mask, only scaled token-similarities in the lower triangular half of the matrix remain.

$$\boldsymbol{X}^{\mathrm{M},i}_{j,k} = \mathrm{Mask}\left(\boldsymbol{X}^{\mathrm{S},i}_{j,k}\right)$$

$$\boldsymbol{X}^{\mathrm{M},i} = \begin{bmatrix} \boldsymbol{X}^{\mathrm{S},i}_{1,1} & -\infty & -\infty & \ldots & -\infty \\ \boldsymbol{X}^{\mathrm{S},i}_{2,1} & \boldsymbol{X}^{\mathrm{S},i}_{2,2} & -\infty & \ldots & -\infty \\ \boldsymbol{X}^{\mathrm{S},i}_{3,1} & \boldsymbol{X}^{\mathrm{S},i}_{3,2} & \boldsymbol{X}^{\mathrm{S},i}_{3,3} & \ldots & -\infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{X}^{\mathrm{S},i}_{m,1} & \boldsymbol{X}^{\mathrm{S},i}_{m,2} & \boldsymbol{X}^{\mathrm{S},i}_{m,3} & \ldots & \boldsymbol{X}^{\mathrm{S},i}_{m,m} \end{bmatrix} \in \mathbb{R}^{m \times m}, \boldsymbol{X}^{\mathrm{S},i} = \frac{\boldsymbol{Q}^i \boldsymbol{K}^{i,\top}}{\sqrt{d_{\text{head}}}}$$

Because the masking is equivalent to adding $-\infty$ to matrix elements, the derivatives become

$$\frac{\partial}{\partial \boldsymbol{X}^{\mathrm{S},i}_{p,q}} \mathrm{Mask}\left(\boldsymbol{X}^{\mathrm{S},i}_{j,k}\right) = \delta_{jp}\delta_{kq}$$

for $j, k, p, q = 1, \ldots, m$.

3.4.3.4 Softmax

Subsequently applying the Softmax function described in Eq. (3.11) to **each row** of the scaled token-similarities yields a right stochastic matrix.

$$\text{Softmax}\left(\boldsymbol{x}_i\right) = \varsigma\left(\boldsymbol{x}_i\right) = \frac{\exp\left(\boldsymbol{x}_i\right)}{\sum_{j=1}^{|\boldsymbol{x}|}\exp\left(\boldsymbol{x}_j\right)} \tag{3.11}$$

For the Softmax function in general, derivatives are computed using $s = \sum_{k=1}^{|\boldsymbol{x}|}\exp\left(\boldsymbol{x}_k\right)$ for two distinct cases.

- for $i = j$

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{x}_j}\varsigma\left(\boldsymbol{x}_i\right) &= \frac{\exp\left(\boldsymbol{x}_i\right)\cdot s - \exp\left(\boldsymbol{x}_i\right)\cdot\exp\left(\boldsymbol{x}_j\right)}{s^2}\\ &= \frac{\exp\left(\boldsymbol{x}_i\right)}{s} - \frac{\exp\left(\boldsymbol{x}_i\right)}{s}\cdot\frac{\exp\left(\boldsymbol{x}_j\right)}{s}\\ &= \varsigma\left(\boldsymbol{x}_i\right) - \varsigma\left(\boldsymbol{x}_i\right)\varsigma\left(\boldsymbol{x}_j\right)\\ &= \varsigma\left(\boldsymbol{x}_i\right)\cdot\left[1-\varsigma\left(\boldsymbol{x}_j\right)\right]\end{aligned}$$

- for $i \neq j$

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{x}_j}\varsigma\left(\boldsymbol{x}_i\right) &= -\frac{\exp\left(\boldsymbol{x}_i\right)}{s^2}\cdot\exp\left(\boldsymbol{x}_j\right)\\ &= -\frac{\exp\left(\boldsymbol{x}_i\right)}{s}\cdot\frac{\exp\left(\boldsymbol{x}_j\right)}{s}\\ &= -\varsigma\left(\boldsymbol{x}_i\right)\varsigma\left(\boldsymbol{x}_j\right)\\ &= \varsigma\left(\boldsymbol{x}_i\right)\cdot\left[0-\varsigma\left(\boldsymbol{x}_j\right)\right]\end{aligned}$$

Both cases can be combined using the Kronecker delta.

$$\frac{\partial}{\partial \boldsymbol{x}_j}\varsigma\left(\boldsymbol{x}_i\right) = \varsigma\left(\boldsymbol{x}_i\right)\cdot\left(\delta_{ij} - \varsigma\left(\boldsymbol{x}_j\right)\right)$$

In the Transformer context, this means

$$\frac{\partial \boldsymbol{X}^{\text{SM},i}_{j,k}}{\partial \boldsymbol{X}^{\text{S},i}_{p,q}} = \delta_{jp}\boldsymbol{X}^{\text{SM},i}_{j,k}\left(\delta_{kq} - \boldsymbol{X}^{\text{SM},i}_{p,q}\right)$$

for $j, k, p, q = 1, \ldots, m$.

Elements of the resulting matrix are non-negative real numbers in the range $[0, 1]$ with matrix rows summing to 1. Furthermore, similarity scores previously set to $-\infty$ vanish as $e^{-\infty} := 0$, resulting in a lower triangular matrix. The elements of this matrix are termed **Attention weights**, and represent the contextual relevance between tokens - specifically, $\boldsymbol{X}^{\text{SM},i}_{j,k}$ measures how relevant the k-th token is to the j-th token.

$$\boldsymbol{X}^{\text{SM},i}_{j,k} = \varsigma\left(\boldsymbol{X}^{\text{M},i}_{j,k}\right)$$

$$\boldsymbol{X}^{\text{SM},i} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \boldsymbol{X}^{\text{SM},i}_{2,1} & \boldsymbol{X}^{\text{SM},i}_{2,2} & 0 & \dots & 0 \\ \boldsymbol{X}^{\text{SM},i}_{3,1} & \boldsymbol{X}^{\text{SM},i}_{3,2} & \boldsymbol{X}^{\text{SM},i}_{3,3} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{X}^{\text{SM},i}_{m,1} & \boldsymbol{X}^{\text{SM},i}_{m,2} & \boldsymbol{X}^{\text{SM},i}_{m,3} & \dots & \boldsymbol{X}^{\text{SM},i}_{m,m} \end{bmatrix} \in [0,1]^{m\times m}, \sum_{k=1}^{m} \boldsymbol{X}^{\text{SM},i}_{j,k} = 1$$

3.4.3.5 Attention Scores

To obtain the final Attention scores for the i-th head, the weights are multiplied by the value vectors following Eq. (3.9). This can be interpreted as the weighted aggregation of information from previous tokens in the input. No information from succeeding tokens is aggregated, since their Attention weights are 0.

$$\boldsymbol{X}^{\text{A},i} = \boldsymbol{X}^{\text{SM},i}\boldsymbol{V}^{i} \qquad \boldsymbol{X}^{\text{A},i} \in \mathbb{R}^{m\times d_{\text{head}}} \tag{3.12}$$

The corresponding derivatives are

$$\frac{\partial \boldsymbol{X}^{\text{A},i}_{j,k}}{\partial \boldsymbol{X}^{\text{SM},i}_{p,q}} = \delta_{jp}\boldsymbol{V}^{i}_{q,k}$$

for $j, p, q = 1, \dots, m$ and $k = 1, \dots, d_{\text{head}}$ and

$$\frac{\partial \boldsymbol{X}^{\text{A},i}_{j,k}}{\partial \boldsymbol{V}^{i}_{p,q}} = \delta_{jp}\boldsymbol{X}^{\text{SM},i}_{j,p}$$

for $j, p = 1, \dots, m$ and $k, q = 1, \dots, d_{\text{head}}$.

Putting all parts together, Scaled Dot-Product Attention is described by Eq. (3.13) for the i-th Attention head.

$$\text{Attention}\left(\boldsymbol{Q}^{i}, \boldsymbol{K}^{i}, \boldsymbol{V}^{i}\right) = \varsigma\left(\text{Mask}\left(\frac{\boldsymbol{Q}^{i}\boldsymbol{K}^{i,\top}}{\sqrt{d_{\text{head}}}}\right)\right)\boldsymbol{V}^{i} \tag{3.13}$$

3.4.4 Concatenation and Output Projection

As shown in Eq. (3.14), the results computed independently in each Attention head are concatenated along their head dimension, yielding $h \cdot d_{\text{head}} = d_{\text{model}}$-dimensional vectors. The final step of Masked Multi-Head Self-Attention involves a linear transformation of these vectors using a weight matrix $\boldsymbol{W}^{\text{O}}$ (containing randomly initialized, learnable parameters).

$$\begin{gathered}\boldsymbol{X}^{\text{MHA}} = \text{Concat}\left(\boldsymbol{X}^{\text{A},1}, \boldsymbol{X}^{\text{A},2}, ..., \boldsymbol{X}^{\text{A},h}\right) \boldsymbol{W}^{\text{O}} \\ \boldsymbol{X}^{\text{MHA}} \in \mathbb{R}^{m \times d_{\text{model}}}, \boldsymbol{W}^{\text{O}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}\end{gathered} \tag{3.14}$$

The derivatives, using $\boldsymbol{X}^{\text{Concat}} = \text{Concat}\left(\boldsymbol{X}^{\text{A},1}, \boldsymbol{X}^{\text{A},2}, ..., \boldsymbol{X}^{\text{A},h}\right)$, are

$$\frac{\partial \boldsymbol{X}^{\text{MHA}}_{i,j}}{\partial \boldsymbol{X}^{\text{Concat}}_{p,q}} = \delta_{ip} \boldsymbol{W}^{\text{O}}_{q,j}$$

for $i, p = 1, \ldots, m$ and $j, q = 1, \ldots, d_{\text{model}}$ and

$$\frac{\partial \boldsymbol{X}^{\text{MHA}}_{i,j}}{\partial \boldsymbol{W}^{\text{O}}_{p,q}} = \delta_{jq} \boldsymbol{X}^{\text{Concat}}_{i,p}$$

for $i = 1, \ldots, m$ and $j, p, q = 1, \ldots, d_{\text{model}}$.

To obtain the concatenation derivatives, the matrix is split into equally sized chunks analogous to Eq. (3.6).

3.5 Feed-Forward Network (FFN)

Tokens have so far aggregated contextual information from other tokens. In the feed-forward network, this accumulated information is processed. This block handles tokens individually and in parallel using linear transformations and activation functions. Tokens do not interact as they do in the Attention block, and for each token, weight matrices and bias vectors are shared. These weights can be interpreted as the model's information memory.

The original feed-forward network proposed by Vaswani et al. [1] uses three layers: an input layer and two hidden layers with trainable parameters, as depicted in Fig. 3.7. The first hidden layer (abbreviated as L_1) contains $4 \cdot d_{\text{model}}$ neurons and the second hidden layer (abbreviated as L_2) has d_{model} neurons (4 can be seen as a hyperparameter, although Transformer variants usually adopt this value). Besides the weight matrices $\boldsymbol{W}^{\mathrm{L}_1}, \boldsymbol{W}^{\mathrm{L}_2}$, the network also uses the bias vectors $\boldsymbol{b}^{\mathrm{L}_1}, \boldsymbol{b}^{\mathrm{L}_2}$ for each hidden layer, with all parameters initialized randomly. These hidden layers in the feed-forward network turn out to contain most of a Transformer's parameters.

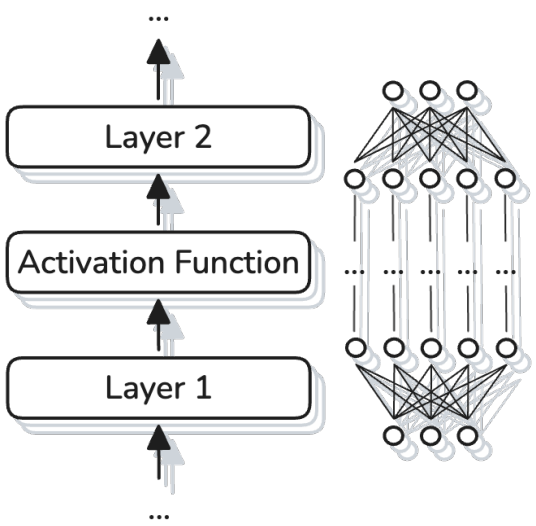

Figure 3.7: The feed-forward network is applied to each token individually and in parallel (represented by the depth).

In between the hidden layers, an activation function is applied to each neuron. While Vaswani et al. [1] used the ReLU activation function described in Eq. (3.15), later models, such as the GPT series, used GELU [25] (Eq. (3.16)) for improved performance.

$$\mathrm{ReLU}\left(x\right) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{3.15}$$

$$\mathrm{GELU}\left(x\right) = x \cdot \Phi\left(x\right) = x \cdot \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \tag{3.16}$$

For all m normalized embedding vectors within the matrix $\boldsymbol{X}^{\mathrm{LN}} \in \mathbb{R}^{m \times d_{\text{model}}}$, the output of the feed-forward network is computed as described in Eq. (3.17), using an activation function f_{act} applied element-wise.

$$\begin{gathered} \boldsymbol{X}^{\mathrm{L}^1} = \boldsymbol{X}^{\mathrm{LN}}\boldsymbol{W}^{\mathrm{L}^1} + \boldsymbol{1}^m \boldsymbol{b}^{\mathrm{L}^1,\top} \\ \boldsymbol{X}^{\text{act}} = f_{\text{act}}\left(\boldsymbol{X}^{\mathrm{L}^1}\right) \\ \boldsymbol{X}^{\text{FFN}} = \boldsymbol{X}^{\text{act}}\boldsymbol{W}^{\mathrm{L}^2} + \boldsymbol{1}^m \boldsymbol{b}^{\mathrm{L}^2,\top} \\ \boldsymbol{X}^{\text{FFN}} \in \mathbb{R}^{m \times d_{\text{model}}} \\ \boldsymbol{W}^{\mathrm{L}^1} \in \mathbb{R}^{d_{\text{model}} \times 4 \cdot d_{\text{model}}}, \boldsymbol{b}^{\mathrm{L}^1} \in \mathbb{R}^{4 \cdot d_{\text{model}}} \\ \boldsymbol{W}^{\mathrm{L}^2} \in \mathbb{R}^{4 \cdot d_{\text{model}} \times d_{\text{model}}}, \boldsymbol{b}^{\mathrm{L}^2} \in \mathbb{R}^{d_{\text{model}}} \end{gathered} \tag{3.17}$$

The corresponding derivatives for L^2 are

$$\frac{\partial \boldsymbol{X}^{\text{FFN}}_{i,j}}{\partial \boldsymbol{X}^{\text{act}}_{p,q}} = \delta_{ip}\boldsymbol{W}^{\mathrm{L}^2}_{q,j}$$

for $i, p = 1, \ldots, m$ and $j = 1, \ldots, d_{\text{model}}$ and $q = 1, \ldots, 4 \cdot d_{\text{model}}$,

$$\frac{\partial \boldsymbol{X}^{\text{FFN}}_{i,j}}{\partial \boldsymbol{W}^{\mathrm{L}^2}_{p,q}} = \delta_{jq}\boldsymbol{X}^{\text{act}}_{i,p}$$

for $i = 1, \ldots, m$ and $j, q = 1, \ldots, d_{\text{model}}$ and $p = 1, \ldots, 4 \cdot d_{\text{model}}$, and

$$\frac{\partial \boldsymbol{X}^{\text{FFN}}_{i,j}}{\partial \boldsymbol{b}^{\mathrm{L}^2}_{p}} = \delta_{jp}$$

for $i = 1, \ldots, m$ and $j, p = 1, \ldots, d_{\text{model}}$.

The activation function derivatives are

$$\frac{\partial \boldsymbol{X}^{\text{act}}_{i,j}}{\partial \boldsymbol{X}^{\mathrm{L}^1}_{p,q}} = \delta_{ip}\delta_{jq}\frac{\partial}{\partial \boldsymbol{X}^{\mathrm{L}^1}_{p,q}} f_{\text{act}}\left(\boldsymbol{X}^{\mathrm{L}^1}_{i,j}\right)$$

for $i, p = 1, \ldots, m$ and $j, q = 1, \ldots, 4 \cdot d_{\text{model}}$.

Lastly, the corresponding derivatives for L^1 are

$$\frac{\partial \boldsymbol{X}^{\mathrm{L}^1}_{i,j}}{\partial \boldsymbol{X}^{\mathrm{LN}}_{p,q}} = \delta_{ip} \boldsymbol{W}^{\mathrm{L}^1}_{q,j}$$

for $i, p = 1, \ldots, m$ and $j = 1, \ldots, 4 \cdot d_{\text{model}}$ and $q = 1, \ldots, d_{\text{model}}$,

$$\frac{\partial \boldsymbol{X}^{\mathrm{L}^1}_{i,j}}{\partial \boldsymbol{W}^{\mathrm{L}^1}_{p,q}} = \delta_{jq} \boldsymbol{X}^{\mathrm{LN}}_{i,p}$$

for $i = 1, \ldots, m$ and $j, q = 1, \ldots, 4 \cdot d_{\text{model}}$ and $p = 1, \ldots, d_{\text{model}}$, and

$$\frac{\partial \boldsymbol{X}^{\mathrm{L}^1}_{i,j}}{\partial \boldsymbol{b}^{\mathrm{L}^1}_{p}} = \delta_{jp}$$

for $i = 1, \ldots, m$ and $j, p = 1, \ldots, 4 \cdot d_{\text{model}}$.

3.6 Classifier

After tokens have aggregated contextual information in Attention blocks and were subsequently processed in feed-forward blocks, the probabilities for the next-token prediction can be computed. As described in Eq. (3.18), a final Layer Normalization is performed, followed by a linear transformation to obtain the so-called **model logits** $\boldsymbol{X}^{\mathrm{L}}$. Applying the Softmax function yields probabilities $\hat{\boldsymbol{Y}}_{i,j}$ which represent the probability of **the j-th token of the vocabulary $\mathbb{V}$ following the i-th input token** .

For the linear transformation, the embedding table $\boldsymbol{W}^{\mathrm{TE}}$ is re-used, as Press et al. [26] showed that this *"weight tying"* reduces the model size without harming its performance.

$$\begin{aligned} \boldsymbol{X}^{\mathrm{L}} &= \mathrm{Layernorm}\left(\boldsymbol{X}^{\mathrm{FFN}}\right) \boldsymbol{W}^{\mathrm{TE},\top} \\ \hat{\boldsymbol{Y}} &= \mathrm{Softmax}\left(\boldsymbol{X}^{\mathrm{L}}\right) \\ \hat{\boldsymbol{Y}} \in [0,1]^{m\times n}, \boldsymbol{W}^{\mathrm{TE}} \in \mathbb{R}^{n\times d_{\text{model}}}, &\sum_{j=1}^{n} \hat{\boldsymbol{Y}}_{i,j} = 1 \end{aligned} \tag{3.18}$$

Computing the Layernorm and Softmax derivatives is analogous to Secs. 3.2 and 3.4, respectively. Using $\boldsymbol{X}^{\mathrm{LN}} = \mathrm{Layernorm}\left(\boldsymbol{X}^{\mathrm{FFN}}\right)$, the remaining derivatives are

$$\frac{\partial \boldsymbol{X}^{\mathrm{L}}_{i,j}}{\partial \boldsymbol{X}^{\mathrm{LN}}_{p,q}} = \delta_{ip} \boldsymbol{W}^{\mathrm{TE}}_{j,q}$$

for $i, p = 1, \ldots, m$ and $j = 1, \ldots, n$ and $q = 1, \ldots, d_{\text{model}}$, and

$$\frac{\partial \boldsymbol{X}^{\mathrm{L}}_{i,j}}{\partial \boldsymbol{W}^{\mathrm{TE}}_{p,q}} = \delta_{jp} \boldsymbol{X}^{\mathrm{LN}}_{i,q}$$

for $i = 1, \ldots, m$ and $j, p = 1, \ldots, n$ and $q = 1, \ldots, d_{\text{model}}$.

This concludes the components of a decoder-Transformer model.

Chapter 4

Training of Transformer Models

The training of LLMs typically consists of two or three phases:

- **Pre-training**: In this first phase, an untrained Transformer model with randomly initialized parameters is trained to generate text via next-token-prediction. This stage of training is computationally expensive, as it involves a supervised learning approach and requires a large dataset. A sufficiently pre-trained decoder large language model can generate sensible text by autoregressively predicting the next token, thereby completing the token-sequence.
- **Supervised fine-tuning**: Once an LLM went through the pre-training phase, it can predict next-token-probabilities following a given input text. However, for it to be used to generate coherent responses in a conversational setting (e.g. as an assistant or a chatbot), the generated text needs to adhere to a specific format (involving answers to input questions). For this reason, this second step involves fine-tuning the model to follow to a given format (again using a supervised learning approach).
- **(optional) Reinforcement learning**: In this last phase, a fine-tuned model may be trained further using reinforcement learning. In this approach, the model aims to maximize a reward-value, without being explicitly given the desired output (in contrast to earlier phases). Recently, Shao et. al [27] showed how this procedure allows models to develop reasoning capabilities required to solve challenging math problems.

Pre-training and supervised fine-tuning both involve training the model using supervised learning. This approach is therefore described in the following sections.

4.1 Training Data

In supervised learning, a machine learning model is given pairs of data $P = (\boldsymbol{x}^i, \boldsymbol{y}^i)$ with $\boldsymbol{x}^i, \boldsymbol{y}^i \in \mathbb{N}^m$, called **training patterns**. Each pattern contains an input token ID vector $\boldsymbol{x}^i$ and a corresponding target token ID vector $\boldsymbol{y}^i$. To predict the j-th token of the output vector, the Transformer model considers all input tokens $1, \ldots, j$, as described in Sec. 3.4.

During training, pairs are sampled from a **dataset** $\mathbb{D} = \{(\boldsymbol{x}^i, \boldsymbol{y}^i)\}, i = 1, \ldots, N,$. This dataset is obtained by tokenizing a training text $s_k \in \mathbb{S}$ and then forming vectors of consecutive tokens, where $\boldsymbol{t} = f_{\text{token}}(s_k)$ is the tokenized text.

$$\boldsymbol{x}^i = \begin{bmatrix} \boldsymbol{t}_j \\ \boldsymbol{t}_{j+1} \\ \vdots \\ \boldsymbol{t}_{j+m-1} \end{bmatrix}, \boldsymbol{y}^i = \begin{bmatrix} \boldsymbol{t}_{j+1} \\ \boldsymbol{t}_{j+2} \\ \vdots \\ \boldsymbol{t}_{j+m} \end{bmatrix}$$

4.2 Loss Term

Using a training pattern, the Transformer model produces predictions for the next token. Generating text in this sense is an interative, multi-class classification task, as the model predicts one token from a vocabulary of n possibilities. This involves computing probabilities for all of them. A commonly used loss function for this type of problem is the **cross-entropy loss**. In general, for two discrete probability distributions $\boldsymbol{y} \in \{0, 1\}^n$ (target probabilities) and $\hat{\boldsymbol{y}} \in [0, 1]^n$ (predicted probabilities), the cross-entropy loss is defined by Eq. (4.1).

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{n} \boldsymbol{y}_i \log \hat{\boldsymbol{y}}_i \tag{4.1}$$

In Transformer models, the output consists of m predicted probability distributions; i.e., for each of the m input tokens, the model produces probabilities for the $(m+1)$-th token. Consequently, the total loss, as described in Eq. (4.2), is computed as the average cross-entropy loss across all tokens.

$$\mathcal{L}\left(\boldsymbol{Y}, \hat{\boldsymbol{Y}}\right) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} \boldsymbol{Y}_{i,j} \log \hat{\boldsymbol{Y}}_{i,j} \tag{4.2}$$

The objective is thus to minimize the cross-entropy loss of the network with respect to the network parameters θ.

$$\min_{\theta} \mathcal{L}\left(\boldsymbol{Y}, \hat{\boldsymbol{Y}}\right)$$

The corresponding derivatives with respect to the predicted probabilities are

$$\frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{Y}}_{i,j}} = -\frac{1}{m} \frac{\boldsymbol{Y}_{i,j}}{\hat{\boldsymbol{Y}}_{i,j}}$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, n$.

4.3 Updating Model Parameters

Because loss functions used in neural networks are non-convex (the global minimum is not known beforehand), **gradient descent** optimization is used. In this algorithm, the gradient of the loss function with respect to the model parameters (weights) is used to update them iteratively. The gradient is usually estimated using only a subset (a **batch**) of the dataset $\mathbb{D}$. This variant is referred to as **stochastic gradient descent**. Using a subset improves computational efficiency and generalization of the model. This approach also introduces a scaling factor η, the **learning rate**. The full update rule for model parameters $\boldsymbol{\theta}$ is shown in Eq. (4.3).

$$\boldsymbol{\theta}_i^{t+1} = \boldsymbol{\theta}_i^t - \eta \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_i^t} \tag{4.3}$$

Various extensions, such as the **Adam optimizer** [28], have been proposed to improve the training process by modifying this update rule.

4.4 Backpropagation of Error

As Eq. (4.3) shows, the training of neural networks requires computing the gradient of the loss function with respect to every model parameter. For this, the **backpropagation of error** algorithm is used, which dates back to P. Werbos' [29] PhD thesis in 1974.

A neural network can be thought of as a composition of many functions that transform data. To compute the derivatives of function compositions the chain rule is used.

$$y = f_n\left(f_{n-1}\left(\ldots f_2\left(f_1\left(x, \theta_1\right)\right)\right)\right) \rightarrow \frac{\partial y}{\partial \theta_1} = \frac{\partial y}{\partial f_n} \cdot \frac{\partial f_n}{\partial f_{n-1}} \cdots \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial \theta_1}$$

This approach is used in deep learning to backpropagate the error gradient through the network as a model can be represented by a composition of (usually many) functions. Even though Transformer models typically contain billions of parameters, they are also trained using this simple algorithm.

Chapter 5

Generating Text

Model inference involves generating text in an autoregressive manner. This is done by invoking the model in a loop, where the output of one iteration is appended to the input of the next, as depicted in Fig. 5.1. In the initial iteration, the model receives an input text (referred to as the **prompt**) and uses it to generate probabilities for the next token. This output probability distribution is then sampled. A common approach is **top-k-sampling**, where only the k highest token-probabilities are considered. Choosing $k = 1$ results in deterministic text generation. However, this setting also limits the "creativity" of the model, which might be undesirable for real applications. Once the next token is determined, it is appended to the model input and the next iteration starts, thereby continuously appending new tokens to the input.

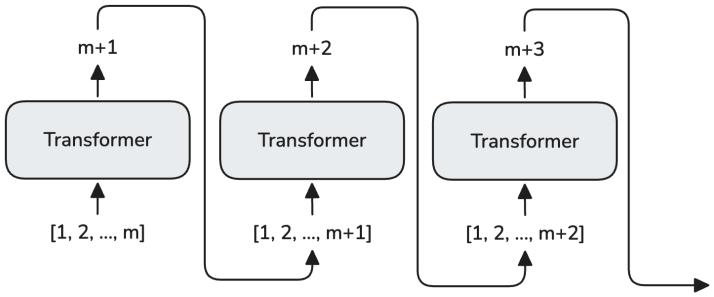

Figure 5.1: Autoregressive text generation.

While generating a probability distribution for all m input tokens is beneficial for the training process, it is not required for inference, since only the distribution of the next token is relevant. Furthermore, starting from the second iteration, only a single token in the input is new, allowing for the re-use of previously computed Attention scores for other tokens. Thereby, higher memory-usage is traded for improved speed.

Chapter 6

Implementation and Experiments

As part of this work, a Python deep-learning library was developed. It prioritizes a transparent and easy-to-read implementation, while offering the required performance to train models with millions of parameters. It is important to mention that outperforming an existing deep-learning framework was not the aim here. Currently, it is powerful enough to build and train Transformer models, while still having easy-to-understand code that is not hidden behind many layers of low-level C++ and CUDA code. Using it, a decoder-Transformer model was implemented and trained in two experiments:

- **Verification**: First, the same exact model architecture was implemented once using PyTorch and once using the developed library. Parameters were identically initialized, and random seeds set. The aim of this first experiment was to verify the correctness of the developed library and to compare both libraries regarding their performance.
- **Pre-Training**: Secondly, an LLM was built and trained using the developed library following a realistic pre-training process with tuned hyperparameters. The goal was to gain insights into the training process and generate text using a pre-trained Transformer model.

The subsequent sections detail both training experiments.

6.1 The Dataset Used

In both experiments, the Tiny Shakespeare dataset was used for simplicity. It is a comparatively small text with approximately 40,000 lines from Shakespeare's plays which fit in a 1 MB text file. An excerpt from the dataset is shown in Tab. 6.1. The file is provided by Andrej Karpathy[1] and is available on his GitHub page[2]. While this dataset is not useful for training a model to generate meaningful text, it is useful to experiment with pre-training a model due to its manageable size.

Table 6.1: The first few lines of the Tiny Shakespeare dataset.

First Citizen: Before we proceed any further, hear me speak.
All: Speak, speak.
First Citizen: You are all resolved rather to die than to famish?
All: Resolved. resolved.
First Citizen: First, you know Caius Marcius is chief enemy to the people.
...

A character-level tokenizer was used, again, for simplicity, with its vocabulary obtained from the unique characters in the dataset. This yielded a vocabulary of $n = 65$ characters, including all lowercase and uppercase letters, numbers, and some punctuation. For training, the dataset was then tokenized (converted into a vector of token IDs) and subsequently split into 90-10 training-validation datasets, resulting in 1,003,855 tokens for training and 111,539 tokens for validation. The vectors of token IDs were then used to generate training patterns, as described in Sec. 4.1.

[1] https://karpathy.ai/
[2] https://github.com/karpathy/char-rnn

6.2 Library Implementation

A lightweight Python library for automatic differentiation was developed for this project. The library relies purely on NumPy [30] (CPU) and CuPy [31] (GPU) for computation. It enables the training of deep learning models with minimal external dependencies while leveraging GPU acceleration.

At its core, the library features a `Tensor` object for storing data and gradients, and `Op` objects for defining differentiable operations. The `Tensor` object is the fundamental data structure. It holds numerical data as a NumPy array and keeps a reference to the `Op` that created it. The `Op` object represents a differentiable operation applied to tensors. Each operation implements methods for both a forward and backward pass.

Typical Workflow:

1. **Computation Graph Construction**: When an operation (e.g., `Tensor.add`) is called, an `Op` instance is created. It performs the forward computation, while also caching intermediate values required for the backward pass. The resulting output tensor maintains references to the `Op` and parent tensors, forming a computational graph, as depicted in Fig. 6.1.

2. **Backpropagation**: Calling the `backward` method of the final tensor (e.g. a loss value) initiates gradient computation. The gradients propagate in reverse through the computational graph by calling `backward` on each `Op`, which distributes gradients to parent tensors.

3. **Gradient Storage**: As the gradients are propagated, they are stored in the `grad` attribute of each `Tensor`, enabling parameter updates for optimization.

The library is licensed under the MIT license and available as open-source software on GitHub [32].

```
        import auto_compyute as ac

        # create random data as 2x3 matrices
        x1 = ac.randn(2, 3, req_grad=True)
        x2 = ac.randn(2, 3, req_grad=True)
        x3 = ac.randn(2, 3, req_grad=True)

        # do computation
        y = x1 ** 2 + 4 * x2 + x3 + 10

        # draw graph
        ac.viz.draw_graph(y)

```

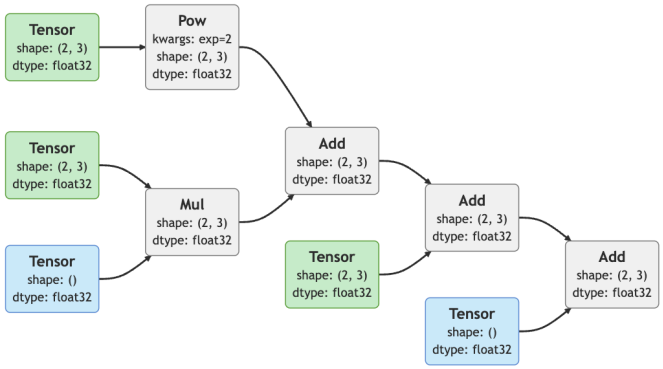

Figure 6.1: Example of computation graph construction from expressions. Green nodes indicate trainable parameters, while grey nodes represent operations.

6.3 Experiment Setup

All training experiments were performed on commodity hardware. Tabs. 6.2 to 6.4 show the used hardware, software versions, as well as other relevant details. Logging was done using the Tensorboard [3] and TensorboardX [4] Python packages.

Table 6.2: Hardware used for all training runs.

Component	**Specifications**
CPU	AMD Ryzen 7 5800X 8-Core Processor
RAM	64 GB DDR4 3200 MHz
GPU	NVIDIA GeForce RTX 4070 (12282 MiB VRAM)

[3] https://github.com/tensorflow/tensorboard
[4] http://github.com/lanpa/tensorboardX

Table 6.3: Software used in all training runs.

Software	**Version**
Operating System	Microsoft Windows 11 Home
Python	3.12
CUDA	12.3
NumPy	2.1.2
CuPy (CUDA 12x)	13.4.1
PyTorch (CUDA 12.4)	2.6.0

Table 6.4: Other details relevant to training.

Detail	**Value**
Parameter data type	32-bit floating point

6.3.1 Verification

This first experiment aimed at evaluating the developed library by comparing it to PyTorch. The model parameters were initialized beforehand and subsequently used in both models to achieve deterministic and consistent results. Furthermore, during training, fixed random seeds were used, dropout was deactivated, and training patterns were used in a fixed order to remove randomness. The hyperparameters used in this evaluation are listed in Tab. 6.5.

Table 6.5: Hyperparameters used in the verification experiment.

Hyperparameter	**Value**
Input sequence length m	256
Embedding dimensions d_{model}	384
Attention heads h	6
Transformer blocks	6
Number of parameters	10,788,864
Dropout probability	0
Optimizer	AdamW [33]
Objective function $\mathcal{L}$	Cross Entropy Loss
Learning rate η	3×10^{-4}
Batch size	64
Training steps	1000

6.3.2 Pre-Training

Once the correctness of the developed library was confirmed, in a second experiment, a more extensive model pre-training was conducted. The hyperparameters used are listed in Tab. 6.6. Furthermore, to observe the performance improvement of the model during training, text samples were generated every 500 steps, where sampling was done using $k = 1$ (token with highest probability). Model parameters were initialized following GPT-2 [19].

Table 6.6: Hyperparameters used for pre-training.

Hyperparameter	**Value**
Input sequence length m	256
Embedding dimensions d_{model}	384
Attention heads h	6
Transformer blocks	6
Number of parameters	10,788,864
Dropout probability	0.3
Optimizer	AdamW [33]
Objective function $\mathcal{L}$	Cross Entropy Loss
Learning rate η	3×10^{-4}
Batch size	64
Training steps	2500

The code used for training is available on GitHub[5].

[5] https://github.com/dakofler/master_thesis_impl

Chapter 7

Results

7.1 Verification

The training details are summarized in Tab. 7.1.

- **Loss**: Fig. 7.1 shows the loss curves, which seem to overlap perfectly, confirming a correct implementation. Fig. 7.2 shows the deviation between the two curves. A positive trend can be seen, as the deviation keeps increasing with the number of training steps. The values themselves, however, are very low in comparison to the actual loss values and can therefore be neglected.
- **VRAM Usage**: Fig. 7.3 shows the graphics memory usage over time. PyTorch uses a maximum of 3246 MiB while the implemented framework reaches a top usage of 5272 MiB, which is roughly 62% higher.
- **Processing speed**: Fig. 7.4 depicts the token processing speed of the libraries where PyTorch's performance is up to 30 times higher.

Table 7.1: Summarized performance results.

Property	**PyTorch**	**Implementation**
Total training time	101 s	319 s
Max VRAM memory usage	3246 MiB	5272 MiB
Avg processing speed	1792819 Tokens/s	58645 Tokens/s

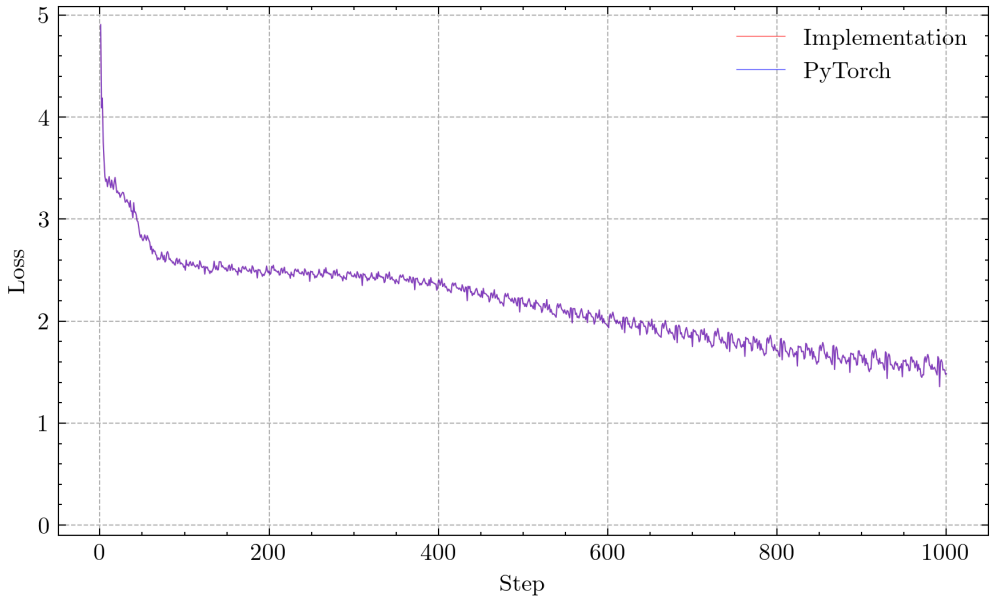

Figure 7.1: The loss curves of PyTorch and the implementated library match perfectly.

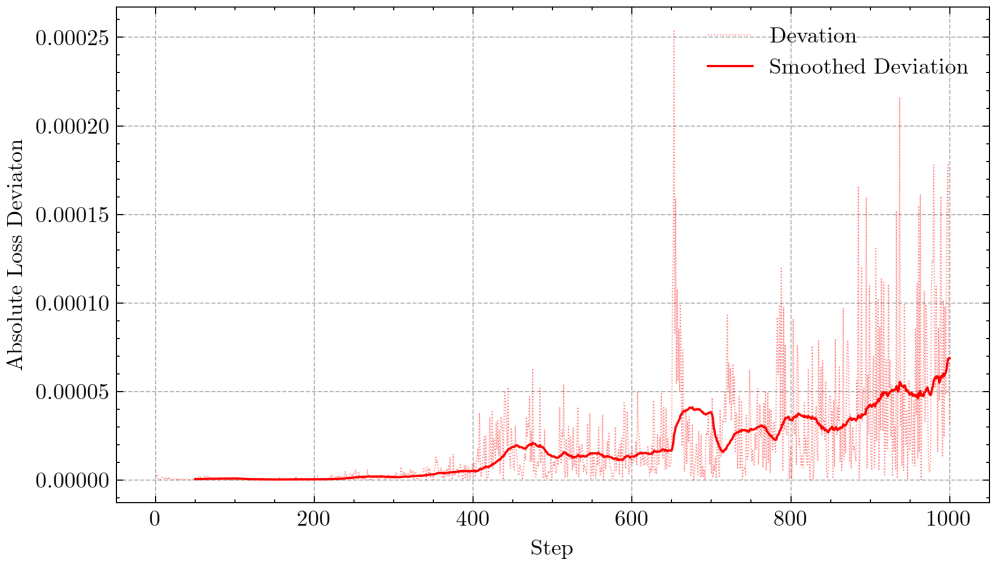

Figure 7.2: The absolute deviation of loss values between PyTorch and the implementatation is insignificantly low but shows a positive trend. For the smoothed line, the rolling mean over 50 steps was applied.

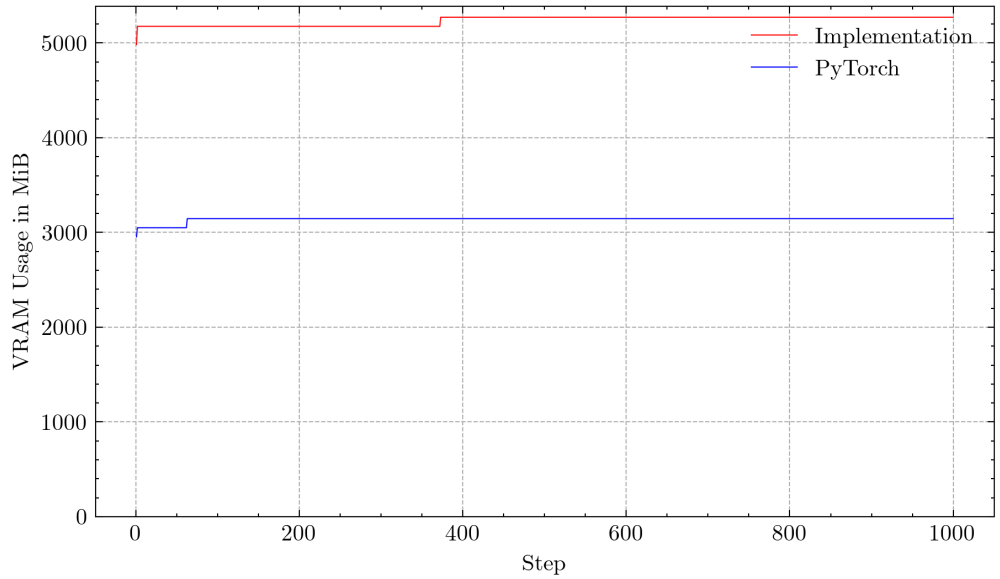

Figure 7.3: The developed library shows an approximately 62% higher VRAM usage throughout the training run.

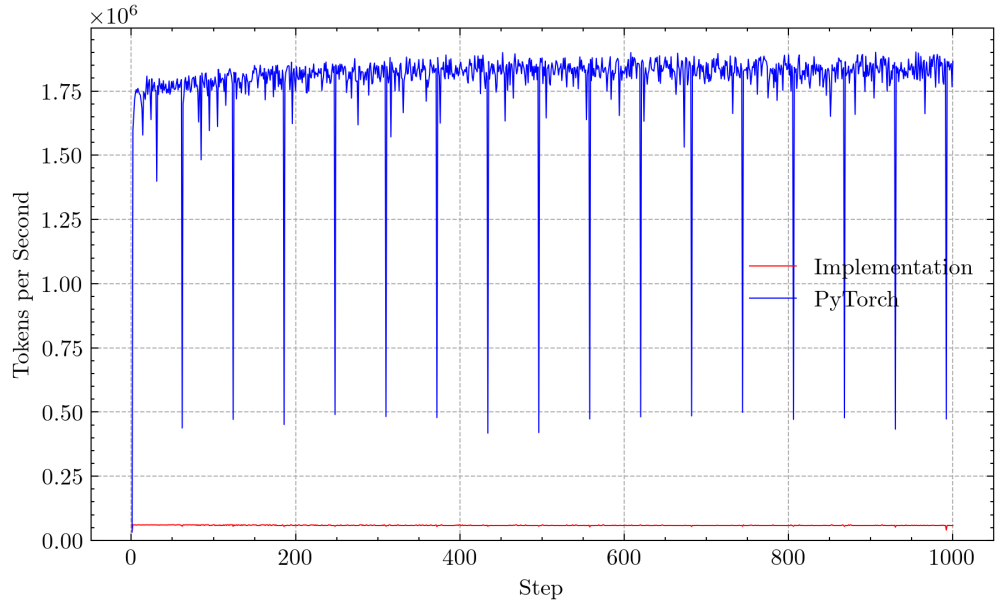

Figure 7.4: PyTorch massively outperforms the implementated library. It shows up to a 30-times higher number of tokens processed per second.

7.2 Pre-Training

The loss curves of the pre-training experiment are shown in Figs. 7.5 and 7.6. While the training loss kept decreasing, the validation loss seemed to plateau and start to slightly increase, indicating that the model started to overfit towards the end.

Appendix A shows all text samples generated during the experimental training. In early steps, the model seems to have adopted the overall text structure while still struggling with forming words and word ordering. By step 1500 the model learned the structure of sentences and used punctuation correctly. Furthermore, it seems to have learned to generate short, but sensible word sequences. By the end of the experiment, the model was able to form correct sentences, although their content was mostly nonsensical.

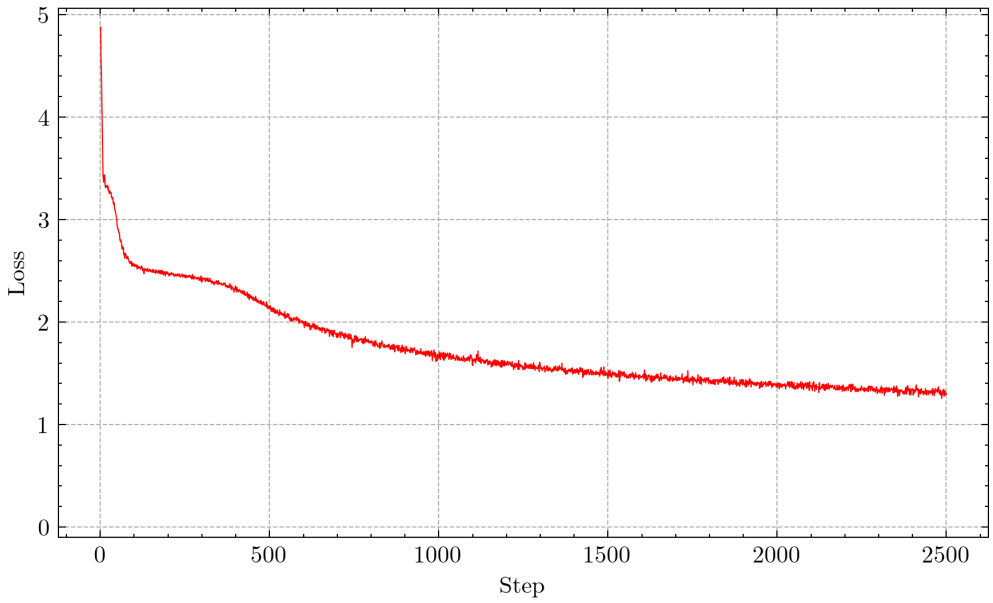

Figure 7.5: Training loss curve.

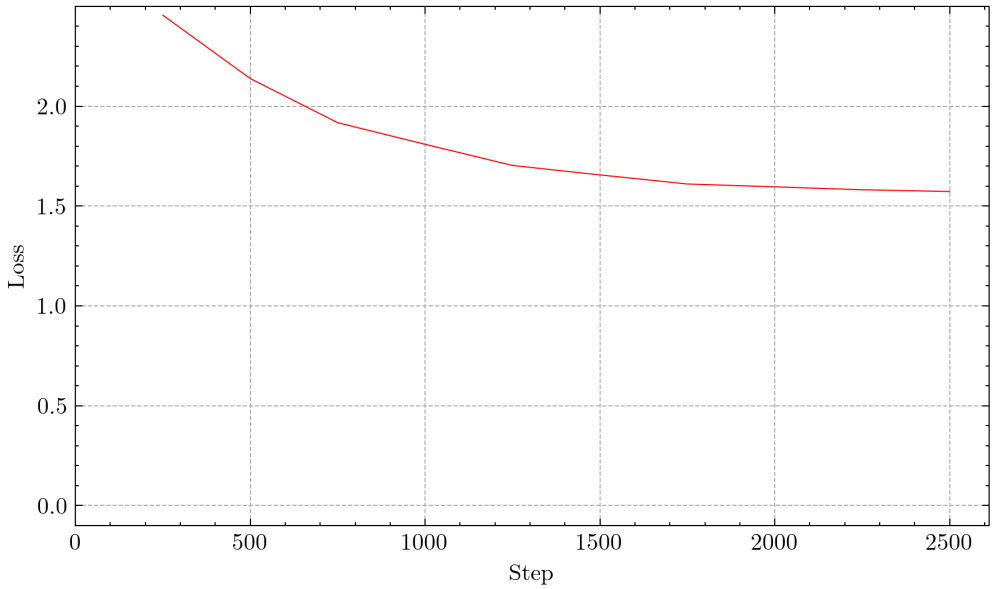

Figure 7.6: Validation loss curve.

Chapter 8

Discussion

8.1 Remarks on the Library Comparison

The results confirm the correct implementation; however, they also clearly indicate a much worse performance of the developed library compared to PyTorch. While the memory usage and total training time are not far behind, the measured processing speed highlights the performance gap. This could be explained by the following:

- **Code Optimization**: While the computations are correct, there is still optimization potential that might help to reduce memory usage, avoid unnecessary memory access and improve performance. For example, there might be redundant caching of matrices for gradient computation that could be avoided.
- **Kernel Fusion**: The library currently makes use of CuPy's operations, each of which read data from memory, perform computation and then save the result back to memory. There are countless situations where data is saved to memory only to be immediately read back to perform the next computation. PyTorch avoids this redundancy by using fused Kernels (multiple operations combined to avoid unnecessary memory reads and writes).

Furthermore, Fig. 7.2 showed slight deviations in the training loss. These might occur due to lower level differences of PyTorch and CuPy, such as floating-point precision and specific algorithmic choices.

8.2 Remarks on Pre-Training

During pre-training, the model clearly showed improvements in its capabilities to generate Shakespeare-like text. While the scale of the pre-training experiment (in terms of dataset used and model size) was sufficient to demonstrate the process, it is not useful for real applications. To train models that are useful in practice, much larger and more general datasets, such as HuggingFace's FineWeb [34] and an increase in the number of model parameters are required. Tab. 8.1 shows a comparison of datasets and model sizes used for current LLMs. Training these larger models furthermore requires significantly more hardware resources and engineering expertise, usually not available to academia.

Table 8.1: Comparison to some published training configurations.

Model	**Training Tokens**	**Number of parameters**
Implementated model	41 M	10.9 M
GPT-2 (smallest) [19]	8 B	117 M
GPT-2 (largest) [19]	8 B	1.542 B
GPT-3 (largest) [20]	400 B	175 B

8.3 Outlook

This work detailed how Transformers function and demonstrated their pre-training process. Since their introduction, interest and investment in generative AI have surged. Public attention has shifted toward AI, and industry adoption is accelerating. In recent years, LLMs have improved significantly, largely due to increased computational resources and new innovations. The race for more powerful models continues with no clear end in sight.

Chapter 9

Conclusions

This work formally describes the architecture of the Transformer neural network model introduced by Vaswani et al. [1], and the training process of large language models. Each component of a typical decoder-Transformer is explained, with mathematical derivations, including training-related gradients, provided. A custom library was developed to implement a Transformer model, supporting the formal descriptions. The implementation was validated against PyTorch, confirming its correctness despite significant performance differences. Finally, the model was pre-trained on the Tiny Shakespeare dataset to demonstrate the learning process. This work thus serves as an introduction to the subject for mathematicians interested in the theory behind LLMs and computer scientists seeking a practical implementation guide.

References

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

[2] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T. Liu, "On layer normalization in the transformer architecture," in *International conference on machine learning*. PMLR, 2020, pp. 10 524–10 533.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

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

[6] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. pmlr, 2015, pp. 448–456.

[7] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[8] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[9] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[10] D. Foster, *Generative deep learning*. " O'Reilly Media, Inc.", 2022.

[11] J. Alammar and M. Grootendorst, *Hands-on large language models: language understanding and generation*. " O'Reilly Media, Inc.", 2024.

[12] M. Phuong and M. Hutter, "Formal algorithms for transformers," *arXiv preprint arXiv:2207.09238*, 2022.

[13] M. Schuster and K. Nakajima, "Japanese and korean voice search," in *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2012, pp. 5149–5152.

[14] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[15] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[16] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," in *International conference on machine learning*. PMLR, 2023, pp. 28 492–28 518.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

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

[19] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[20] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. 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*, vol. 33, pp. 1877–1901, 2020.

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

[22] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," in *International conference on machine learning*. PMLR, 2017, pp. 1243–1252.

[23] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[24] L. A. Goodman, "On the exact variance of products," *Journal of the American statistical association*, vol. 55, no. 292, pp. 708–713, 1960.

[25] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

[26] O. Press and L. Wolf, "Using the output embedding to improve language models," *arXiv preprint arXiv:1608.05859*, 2016.

[27] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu *et al.*, "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *arXiv preprint arXiv:2402.03300*, 2024.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[29] P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," *PhD thesis, Committee on Applied Mathematics, Harvard University, Cambridge, MA*, 1974.

[30] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[31] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf

[32] D. Kofler, "Auto compyute," https://github.com/dakofler/auto_compyute, 2025, version 0.1.0.

[33] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[34] G. Penedo, H. Kydlíček, L. B. allal, A. Lozhkov, M. Mitchell, C. Raffel, L. V. Werra, and T. Wolf, "The fineweb datasets: Decanting the web for the finest text data at scale," in *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. [Online]. Available: https://openreview.net/forum?id=n6SCkn2QaG

Appendix A

Pre-Training Text Samples

Bold texts highlight the input prompt, followed by the generated text (100 new tokens).

Step 500

CLIFFORD:
Here burns my candle out; ay, here it dies,
Which, whiles it lasted, gave King Henry light.

KING RICHARD III:
What hat he mat the the he se theat the so the sto the the the so
The the shat th

BISHOP OF ELY:
Now in good time, here comes the duke himself.

GLOUCESTER:
The the the the the she son the the the so the so son the the spear
The the the the she shat the so

DUKE VINCENTIO:
There was a friar told me of this man.

KING RICHARD III:
What the the the the the she she st the the the so so
The the the sthe st the so

Boatswain:

When the sea is. Hence! What cares these roarers
for the name of king? To cabin: silence! trouble us not.

GONZALO:
The the the she the the the so so the seat the the the sto
The she sto so the sto the the so sthe t

First Senator:
There's no remedy;
Unless, by not so doing, our good city the the the the so the sour the she the son the
The the the she speat the the the the sthe so the t

Step 1000

CLIFFORD:
Here burns my candle out; ay, here it dies,
Which, whiles it lasted, gave King Henry light.

KING EDWARD IV:
Why, what the will the see the so so the so see the comes,
And the with the shall

BISHOP OF ELY:
Now in good time, here comes the duke himself.

GLOUCESTER:
What the will the will the so so the words the words the words,
And the will the words the words th

DUKE VINCENTIO:
There was a friar told me of this man.

KING EDWARD IV:
And the shall the will the come the world the words,
And the will the will the war

Boatswain:
When the sea is. Hence! What cares these roarers
for the name of king? To cabin: silence! trouble us not.

GONZALO:
I would the words the so the words the so the so so so so so so the see the come
To the words the s

First Senator:
There's no remedy;
Unless, by not so doing, our good city
The so see the so the so see the wards of the some
To the words the words of the world the so the

Step 1500

CLIFFORD:
Here burns my candle out; ay, here it dies,
Which, whiles it lasted, gave King Henry light.

KING RICHARD III:
The shall be so the shall be so the word.

KING RICHARD III:
The shall be shall

BISHOP OF ELY:
Now in good time, here comes the duke himself.

GLOUCESTER:
I will not thee the world of the world.

KING EDWARD IV:
Why, then the shall the shall be so more t

DUKE VINCENTIO:
There was a friar told me of this man.

LUCIO:
I will not the words of the world.

LUCIO:
I will not the word.

LUCIO:
I would not the wor

Boatswain:
When the sea is. Hence! What cares these roarers
for the name of king? To cabin: silence! trouble us not.

GONZALO:
I will not thee the world of the world of the state,
The shall be shall be be be be stand the body.

First Senator:
There's no remedy;
Unless, by not so doing, our good city
To the senate of the world of the world.

LEONTES:
I will be not the shall be so be the world.

LEO

Step 2000

CLIFFORD:
Here burns my candle out; ay, here it dies,
Which, whiles it lasted, gave King Henry light.

KING RICHARD II:
The sent of the king of the king,
And the sea of the season of the court.

KING R

BISHOP OF ELY:
Now in good time, here comes the duke himself.

GLOUCESTER:

The king of the country's son, the king's son,
And the season of the world of the world.

KING RICH

DUKE VINCENTIO:
There was a friar told me of this man.

DUKE VINCENTIO:
I would not be so much a word of the world.

DUKE VINCENTIO:
I will not be so thin

Boatswain:
When the sea is. Hence! What cares these roarers
for the name of king? To cabin: silence! trouble us not.

GONZALO:
I would be say you shall be so.

LEONTES:
I will be so the strike of the country.

LADY ANNE:
I wil

First Senator:
There's no remedy;
Unless, by not so doing, our good city,
That we shall be seen to the world.

Second Servant:
The senators of the country of the country.

Step 2500

CLIFFORD:
Here burns my candle out; ay, here it dies,

Which, whiles it lasted, gave King Henry light.

KING HENRY VI:
What says thou wilt thou hast not stand the state of the house?

KING HENRY VI:
Then

BISHOP OF ELY:
Now in good time, here comes the duke himself.

GLOUCESTER:
The gracious lord, and the gracious lord.

LADY ANNE:
I will not stay the gracious lord.

LADY ANNE

DUKE VINCENTIO:
There was a friar told me of this man.

LUCIO:
The gracious lady is not the seat of the world.

LUCIO:
The gods of the prince, and the sta

Boatswain:
When the sea is. Hence! What cares these roarers
for the name of king? To cabin: silence! trouble us not.

GONZALO:
What shall be so?

POLIXENES:
What is the straighter of the state of the state?

PAULINA:
The gods

First Senator:
There’s no remedy;
Unless, by not so doing, our good city.

Second Servant:
The gods of the prince of the state of the state,
And the state of the seat of th