Facultad de Ciencias

Grado en Matemáticas

Mathematical Modeling and Practical Application of Transformer Neural Networks

Author: Bruno Sánchez Gómez

Tutor: Jesús Martín Vaquero

Co-tutor: Roberto López González

2024

Facultad de Ciencias

Grado en Matemáticas

Mathematical Modeling and Practical Application of Transformer Neural Networks

Author: Bruno Sánchez Gómez

Tutor: Jesús Martín Vaquero

Co-tutor: Roberto López González

2024

Certificado de los tutores TFG Grado en Matemáticas

D. Jesús Martín Vaquero, profesor del Departamento de Matemática Aplicada de la Universidad de Salamanca, y D. Roberto López González, CEO de la empresa Artelnics,

HACEN CONSTAR:

Que el trabajo titulado "*Mathematical Modeling and Practical Application of Transformer Neural Networks*", que se presenta, ha sido realizado por D. Bruno Sánchez Gómez, con DNI ****3902A y constituye la memoria del trabajo realizado para la superación de la asignatura Trabajo de Fin de Grado en Matemáticas en esta Universidad.

Salamanca, a fecha de firma electrónica.

MARTIN VAQUERO JESUS - 70877112J

Firmado digitalmente por MARTIN VAQUERO JESUS - 70877112J
Fecha: 2024.06.27 09:59:47 +02'00'

Firmado por LOPEZ GONZALEZ ROBERTO - ***7144** el día 27/06/2024 con un certificado emitido por AC FNMT Usuarios

Fdo.: Jesús Martín Vaquero

Fdo.: Roberto López González

Contents

1 | Introduction

1.1 Neural Networks and Transformers

Neural networks have revolutionized various fields by providing advanced solutions to problems that were previously thought to be solvable only by humans, like image classification, natural language processing, autonomous driving, and more. Their importance in the modern world and society cannot be overstated. However, despite their widespread application, there is a notable lack of a solid and unified mathematical framework that encompasses all types of neural networks. This thesis proposes a solution to this problem by constructing a framework based on function spaces to formally describe the concepts of neural networks and their training processes.

Aside from building a general framework for neural networks, we delve deep into a specific kind of neural network: the *Transformer*. Transformer models have gained significant importance in recent years since their proposal in *Attention is All You Need* [1] by Vaswani et al. (2017). The relevance of Transformers is due to their attention mechanisms, which allow them to process data in a highly parallelizable manner, unlike Recurrent Neural Networks (RNNs) or Long Short-Term Memory networks (LSTMs), which process data sequentially. This parallelization capability has made it feasible to train much larger models within reasonable timescales. As a consequence, it is now possible to train models that excel at handling complex tasks and that achieve results that were unattainable with sequentially processed models. Transformers especially stand out in language modelling tasks, such as translation and text generation. Variations of the Transformer architecture, such as the "Vision Transformer" [2], have also excelled at other tasks like image processing or time-series prediction.

Throughout this thesis, we discuss the general components of the neural network training process, which include the dataset, the neural network architecture, the loss index, and the optimization algorithm. After introducing each component, we delve into its specifics for Transformer models, exploring the intricacies of the training process of this kind of neural network. Finally, the thesis culminates with a practical application that demonstrates the effectiveness of the proposed framework and Transformer model in a real-world scenario.

1.2 Preliminaries

Before getting into the main theme of the thesis, let us start by introducing some key mathematical concepts and notations of importance that are used later.

Parameterized function spaces

Let us consider a function between two arbitrary spaces, $\mathcal{X}, \mathcal{Y}$, parameterized by $\theta \in \Theta$, with Θ a parameter space. We denote it as $f_\theta : \mathcal{X} \to \mathcal{Y}$.

Notation. The following notations are equivalent:

$$f_\theta(x) \equiv f(x;\theta) \quad .$$

Definition 1.1. The *function space generated by* f_θ , $\mathcal{F}$, consists of all functions that can be realized by adjusting different sets of values to the parameters $\theta \in \Theta$. That is,

$$\mathcal{F} := \{f_\theta \mid \theta \in \Theta\} \quad ,$$

where each f_θ represents the specific function with a particular configuration of parameters θ. We say $\mathcal{F}$ is a *parameterized function space.*

Let us now consider another function $g : \mathcal{Y} \to \mathbb{R}$. Then, for any $x \in \mathcal{X}$, we can define a functional $G_x : \mathcal{F} \to \mathbb{R}$ such that:

$$G_x[f_\theta] := g(f_\theta(x)) \quad .$$

We say that g is the function associated with G_x, and it does not depend on the specific choice of $x \in \mathcal{X}$, which only determines the point in which g is evaluated for each $f_\theta \in \mathcal{F}$.

If we now fix a value of $x \in \mathcal{X}$, the expression $g(f(x;\theta))$ only depends on the specific choice of parameters $\theta \in \Theta$. Therefore, when fixing an $x \in \mathcal{X}$ the functional $G_x : \mathcal{F} \to \mathbb{R}$ is equivalent to a function $h : \Theta \to \mathbb{R}$, through the following expression:

$$G_x[f_\theta] = g(f(x;\theta)) =: h(\theta) \quad .$$

This sort of equivalence will be used to reduce the domain of a problem from a parameterized function space (like $\mathcal{F}$) to a more simple parameter space (like Θ).

Euclidean spaces and simplices

We will now introduce some notations and assumptions that we make about Euclidean spaces, and define a subset that is very relevant for Transformers: the simplex.

First of all, every time we consider a Euclidean space we assume that it is of the type $\mathbb{R}^d$ for some dimension $d \in \mathbb{N}$, and that it is equipped with the usual dot product:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{k=1}^{d} a_i \cdot b_i \quad ,$$

for any $\mathbf{a} = (a_1, \ldots, a_d)$, $\mathbf{b} = (b_1, \ldots, b_d) \in \mathbb{R}^d$. Note that, with this dot product, $\mathbb{R}^d$ is also provided with the norm $\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$.

Notation. A notation that we will often encounter is $\mathbb{R}^{n \times m \times d}$ (or similar), with $n, m, d \in \mathbb{N}$. It is interpreted as the space of $n \times m$ matrices of real-valued d-dimensional vectors. Given an element $\mathbf{A} \in \mathbb{R}^{n \times m \times d}$, its components are denoted:

- $\mathbf{A}_{i,j}$, referring to the vector at the position i, j ,
- $\mathbf{A}_{i,j,k}$, referring to the k-th component of the vector at the position i, j ,

with $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m\}$, $k \in \{1, \ldots, d\}$.

Observation 1.2. Most operations are applied upon the vectors, treating them as elements of the Euclidean space $\mathbb{R}^d$. The only exception is the addition operation, which is simply an element-wise addition:

$$(\mathbf{A} + \mathbf{B})_{i,j,k} = \mathbf{A}_{i,j,k} + \mathbf{B}_{i,j,k} \quad .$$

Now, let us define the simplex, which is a subset of the Euclidean space $\mathbb{R}^d$ that plays a critical role later on.

Definition 1.3. We define the $(d-1)$-*simplex* as the subset of the Euclidean space $\mathbb{R}^d$ of the vectors whose components are all non-negative and add up to exactly 1. Formally, that is:

$$\mathbb{S}_{(d-1)} = \left\{ \mathbf{p} = (p_1, \ldots, p_d) \in \mathbb{R}^d \mid \ p_k \geq 0 \ \forall k \ , \ \sum_{k=1}^{d} p_k = 1 \right\} \quad .$$

We refer to the elements of $\mathbb{S}_{(d-1)}$ as *probability vectors*. They represent the discrete probability distributions over an arbitrary set of d events, denoted by X_d, whose events are labeled with the integer numbers from 1 to d. That is, $X_d := \{1, \ldots, d\} \subset \mathbb{Z}$.

Properties. The $(d-1)$-simplex, $\mathbb{S}_{(d-1)}$, has the following properties:

1. **Convexity.** Given $\mathbf{p}, \overline{\mathbf{p}} \in \mathbb{S}_{(d-1)}$,

$$\lambda \mathbf{p} + (1 - \lambda)\overline{\mathbf{p}} \in \mathbb{S}_{(d-1)} \quad , \quad \forall \lambda \in [0, 1] \quad .$$

2. **Compactness.** $\mathbb{S}_{(d-1)}$ is closed and bounded. Therefore, by the Heine-Borel theorem, it is compact.

Remark. The vectors of the standard base of $\mathbb{R}^d$, $\{e_1, \ldots, e_d\}$, are in $\mathbb{S}_{(d-1)}$. They receive the name of *vertices* of the $(d-1)$-simplex and we consider them ordered like so:

$$\begin{aligned}
e_1 &= (1, 0, \dots, 0) \text{ is the 1st vertex.} \\
e_2 &= (0, 1, \dots, 0) \quad \text{is the 2nd vertex.} \\
&\vdots \\
e_d &= (0, \dots, 0, 1) \quad \text{is the } d\text{-th vertex.}
\end{aligned}$$

The vertices represent the degenerate probability distributions on each of the d elements of X_d (i.e. e_k is the degenerate probability distribution on the event corresponding to the label $k \in X_d$).

Let us now define a function that takes values on the $(d-1)$-simplex and that will be important.

Definition 1.4. We define the *softmax* function as the function that converts a real valued vector into a probability vector, through an exponential correspondence. That is, $\mathcal{S} : \mathbb{R}^d \to \mathbb{S}_{(d-1)} \subset \mathbb{R}^d$, such that:

$$\mathcal{S}\Big(x_1, \dots, x_d\Big) := \frac{1}{\sum_{i=1}^d e^{x_i}} \cdot \Big(e^{x_1}, \dots, e^{x_d}\Big) \quad .$$

Remark. It is common to shift the input vector by subtracting its maximum before computing the softmax function. That is,

$$\boxed{\mathcal{S}\Big(x_1, \dots, x_d\Big) := \frac{1}{\sum_{i=1}^d e^{\hat{x}_i}} \cdot \Big(e^{\hat{x}_1}, \dots, e^{\hat{x}_d}\Big)} \quad , \tag{1.2.1}$$

where $\hat{x}_i = x_i - \max_{j=1}^{d}(x_j)$. The use of $\hat{x}$ instead of x improves numerical stability when computing large values of x, and the result of the softmax operation does not change whether we apply this shifting or not.

Observation 1.5. The softmax function (1.2.1) is implemented in C++ code in Apendix B.4.

Jacobians

Eventually, we will want to calculate the Jacobians of multivariable functions between spaces like $\mathbb{R}^{n \times m \times d}$. The notation of these types of derivatives can easily get complicated, so let us now introduce how we denote them. Then, we calculate the Jacobian matrix of the softmax function (1.2.1) as an example since it is needed later.

Notation. Let us consider a multivariable function $f : \mathbb{R}^{n \times m \times d} \to \mathcal{Y}$, with $n, m, d \in \mathbb{N}$. We use 2 similar (though not identical) notations depending on the output space $\mathcal{Y}$.

- When $\mathcal{Y}$ is of the type $\mathcal{Y} = \mathbb{R}^{n\times m\times d}$, we say f is a *multi-valued function*, and the notation employed for its Jacobian is "component-wise":

$$\frac{\partial f_{i,j,k}}{\partial x_{\bar{i},\bar{j},\bar{k}}}(x) = \frac{\partial f(x)_{i,j,k}}{\partial x_{\bar{i},\bar{j},\bar{k}}} \quad ,$$

  with $i, \bar{i} = 1, \dots, n$, $j, \bar{j} = 1, \dots, m$, $k, \bar{k} = 1, \dots, d$, where $f(x)_{i,j,k}$ is the (i, j, k) component of the function f. Note that for auxiliary indices, we utilize the "overline" notation.

- When $\mathcal{Y} = \mathbb{R}$, f is not a multi-valued function, and the notation employed for its Jacobian is "block-wise":

$$\frac{\partial f}{\partial x}(x) = \left(\frac{\partial f(x)}{\partial x_{i,j,k}}\right)_{\substack{i=1,\dots,n\\ j=1,\dots,m\\ k=1,\dots,d}} \quad .$$

Let us now calculate the Jacobian of the softmax function (1.2.1), which is a multivariable and multi-valued function.

Proposition 1.6. *Given the softmax function, $\mathcal{S} : \mathbb{R}^d \to \mathbb{S}_{(d-1)} \subset \mathbb{R}^d$, its Jacobian is:*

$$\frac{\partial \mathcal{S}_k}{\partial x_{\bar{k}}}\Big(x_1, \dots, x_d\Big) = S_{\bar{k}} \cdot (\mathbf{1}\{\bar{k} = k\} - S_k) \quad , \qquad (1.2.2)$$

where $\mathbf{1}\{a = b\} = \begin{cases} 1, & a = b \\ 0, & a \neq b \end{cases}$ *is the indicator function (also known as Kronecker delta), and $S \equiv \mathcal{S}(x_1, \dots, x_d)$.*

Proof. Let us calculate $\frac{\partial \mathcal{S}_k}{\partial x_{\bar{k}}}$:

$$\begin{aligned}\frac{\partial \mathcal{S}_k}{\partial x_{\bar{k}}}\Big(x_1, \dots, x_d\Big) &= \frac{\partial}{\partial x_{\bar{k}}}\left(\frac{e^{x_k}}{\sum\limits_{i=1}^{d} e^{x_i}}\right) \\ &= \frac{e^{x_{\bar{k}}} \cdot \mathbf{1}\{\bar{k} = k\}}{\sum\limits_{i=1}^{d} e^{x_i}} - \frac{e^{x_k} \cdot e^{x_{\bar{k}}}}{\left(\sum\limits_{i=1}^{d} e^{x_i}\right)^2} \\ &= S_{\bar{k}} \cdot \mathbf{1}\{\bar{k} = k\} - S_k \cdot S_{\bar{k}} \quad ,\end{aligned}$$

which simplifies to the stated expression. □

1.3 Neural Network Training

Neural networks have been proved to be extremely proficient at performing complex tasks that only humans were thought to be capable of, like image classification, text generation or sentiment analysis. However, neural networks are not competent just by nature, they first have to learn how to perform any given task before they can get any good results. This is achieved through a training process, that is defined by the following 4 main components:

1. **Dataset.** It holds the information upon which the neural network will be trained, hence defining the task to be performed.
2. **Neural Network.** It establishes how the task will be performed, based on its layers and architecture.
3. **Loss Index.** It measures the neural network's performance at its task, and gives information on how to improve said performance.
4. **Optimization Algorithm.** It defines how the information provided by the Loss Index will be used in order to succeed in the training process.

In each of the Chapters 2-5, we give a formal definition of one of these components and detail its specific configuration for Transformer neural network training. Afterwards, in Chapter 6 we discuss the practical application of the defined concepts through an experiment where we train multiple Transformer models. Finally, in Chapter 7 we summarize our conclusions.

2 Dataset

The dataset is a fundamental part of the training process of a neural network. It establishes a collection of inputs, and the target values that we expect of the neural network for each of them; thus defining the task that said neural network will be trained on.

In this chapter we formally define the concept of dataset in the context of neural networks, and then define the type of dataset a Transformer model is trained on.

2.1 Definition

Definition 2.1. A *dataset* (D) with n samples is defined as a finite set of pairs $(x_i, \hat{y}_i)$, $i = 1, 2, \ldots, n$, where:

- $x_i \in \mathcal{X}$ represents the inputs.
- $\hat{y}_i \in \mathcal{Y}$ represents the targets.

Here, $\mathcal{X}$ and $\mathcal{Y}$ denote the input space and output space, respectively. Both are finite-dimensional spaces.

Observation 2.2. The input and output spaces cannot be generally defined, since they are specific to each neural network and its task, but some common examples are:

- **Input Space ($\mathcal{X}$):** This is often a subset of $\mathbb{R}^d$, where d is the number of features or dimensions in the input data. Some advanced neural network models can have more than one type of input, in which case we denote $\mathcal{X} = \mathcal{X}_1 \times \ldots \times \mathcal{X}_N$, where each $\mathcal{X}_i$ is an input sub-space, and $N \in \mathbb{N}$ is the number of different types of inputs to the neural network.
- **Output Space ($\mathcal{Y}$):** This space heavily depends on the task type. For regression tasks, $\mathcal{Y}$ might also be a subset of $\mathbb{R}$, or $\mathbb{R}^{\tilde{d}}$ for multiple outputs. For classification tasks, $\mathcal{Y}$ often consists of discrete labels, which can be expressed as a subset of $\mathbb{Z}$, or $\mathbb{Z}^{\tilde{d}}$ for multiple classification.

***Example* 2.3.** Datasets are often represented in tables like Table 2.1, which shows a dataset with n samples corresponding to a regression task. In said example the input space would be $\mathcal{X} = \mathbb{R}^d$, and the output space $\mathcal{Y} = \mathbb{R}$.

	Feature 1	**Feature 2**	...	**Feature d**	**Output**
Sample 1	3.4	5.6	...	9.0	10.1
Sample 2	4.5	6.7	...	10.1	11.2
			⋮		
Sample n	14.5	16.7	...	20.1	21.2

Table 2.1: Example of a dataset table.

2.2 Sequence Transduction Dataset

Transformers are designed as sequence transduction models. In the context of machine learning, sequence transduction refers to the process of transforming one sequence into another of a different kind.

Examples. Some common examples of sequence transduction are:

- Translation: Transforms a sentence from one language to another.
- Dictation: Transforms spoken language into written text.
- Summarization: Transforms a long piece of text into a shorter one.

These are the types of tasks Transformer models are typically used for and where they have shown good performance. Therefore, sequence transduction datasets are used for their training.

A sequence transduction dataset consists of pairs of sequences of labels. We call *source sequence* to the first of each pair and *reference sequence* to the other. During training, the model learns how to transform the source sequences into their corresponding reference sequences.

Once the model is trained, a new source sequence and an empty reference sequence would be provided. The model then would generate a new sequence recurrently: one label at a time, adding each new label to the reference sequence for the next step.

Input space

Each label in a sequence can represent a word, a word-piece, a DNA base, or any other discrete unit, depending on the specific type of data. Regardless of what they represent, we always assume there is a finite amount of possible labels, $v \in \mathbb{N}$, which we call *vocabulary size*. Then, the set of labels is formally defined as:

$$X_v := \{1, \dots, v\} \subset \mathbb{Z} \quad . \tag{2.2.1}$$

Initially, not all sequences necessarily have the same number of labels. To standardize them, we fix a length $m \in \mathbb{N}$ and follow these steps:

1. Prepend each sequence with a start label and append it with an end label.

2.1. If a sequence exceeds length m, trim it to m labels.

2.2. If a sequence is shorter than m, add padding labels until it reaches length m.

Then, a sequence is defined as an element of X_v^m. For simplicity, start, end, and padding labels are included in X_v.

The sets of possible labels are different for source and reference sequences, so we denote the source and reference vocabularies by X_{v_S} and X_{v_R}, respectively (where $v_S \in \mathbb{N}$ is the source vocabulary size and $v_R \in \mathbb{N}$ is the reference vocabulary size).

The lengths we use for standardization are also different. Let $m_S, m_R \in \mathbb{N}$; we use length m_S for source sequences and $m_R + 1$ for reference sequences. Therefore, a source sequence is an element of $X_{v_S}^{m_S}$ and a reference sequence is an element of $X_{v_R}^{m_R+1}$.

We can finally define the input space of a Transformer as

$$\mathcal{X} := X_{v_S}^{m_S} \times X_{v_R}^{m_R} \quad , \tag{2.2.2}$$

where an input sample is a pair (x_S, x_R), with x_S a source sequence and x_R a reference sequence without the last label.

Output space

Throughout the neural network, the source and reference sequences are compared and transformed. Ultimately, the output is a new sequence of length m_R. Instead of labels, each position of the output sequence contains a probability vector of dimension v_R that represents the likelihood of each element of X_{v_R} to occupy that position.

Therefore, the output space is defined as:

$$\mathcal{Y} := \mathbb{S}_{(v_R-1)}^{m_R} \quad , \tag{2.2.3}$$

where a target sample is a sequence $(\hat{y}_1, \dots, \hat{y}_{m_R})$, with each $\hat{y}_j$ a vertex of the $(v_R - 1)$-simplex, $\mathbb{S}_{(v_R-1)}$.

Note that said vertices are the unit vectors of the standard base of $\mathbb{R}^{v_R}$. This means that the vertex e_s , $s = 1, \dots, v_R$, represents the degenerate probability distribution on the s-th element of the vocabulary X_{v_R}.

Remark. We consider the vertices to be ordered according to their corresponding labels of the vocabulary (i.e. e_s is the s-th vertex of $\mathbb{S}_{(v_R-1)}$).

For a given sample, the reference sequence, x_R, and the target sequence, $\hat{y}$, are related as follows:

$$\text{For each } j = 1, \ldots, m_R\ , \quad \hat{y}_j \text{ is the } x_{Rj+1}\text{-th vertex of } \mathbb{S}_{(v_R-1)}.$$

Here, x_{Rm_R+1} represents the last label of the full reference sequence x_R before the last label is dropped. Note that this is the reason why we use length $m_R + 1$ for reference sequences instead of m_R.

By establishing this framework, the model learns to predict at each position the next label of the reference sequence by using the information in the source sequence. Figure 2.1 shows an example of how a sequence transduction sample is structured.

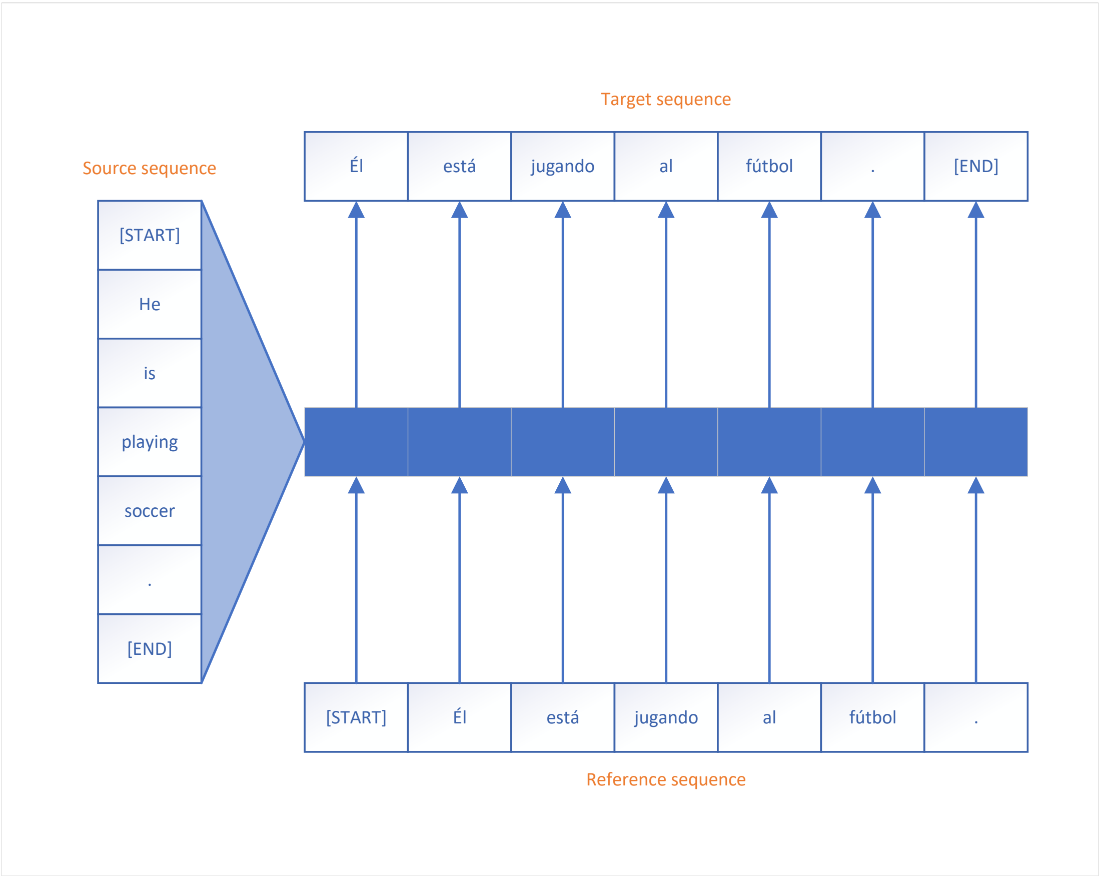


Figure 2.1: Diagram of a sequence transduction sample. The row in the centre represents the neural network. Each position of the target sequence holds the degenerate probability distribution on the label in the next position of the reference sequence.

3 | Neural Network

Once we have a dataset, we know what task is to be performed. The next step is to build the neural network itself. Nowadays there a multitude of types neural networks, each specializing in a different kind of task. Hence, the choice of which neural network to build is crucial for its future performance in the given task.

Furthermore, the neural network is the only component that transcends the training process. The others are fundamental only for training, but the neural network persists afterwards as the tool employed to solve the task.

In this chapter, we define neural networks and delve deep into the Transformer architecture and each of its components. We assume to have a dataset D with $n \in \mathbb{N}$ samples. Note that, by having a dataset, the input space $\mathcal{X}$ and output space $\mathcal{Y}$ are defined. In addition, for the Transformer neural network, we will assume that D is specifically a sequence transduction dataset.

3.1 Definition

Definition 3.1. A *neural network* is a function $y : \mathcal{X}^n \to \mathcal{Y}^n$ parameterized by $\theta \in \Theta$, where Θ is a parameter space. Each choice of parameters $\theta \in \Theta$ specifies a particular function. This can be written as:

$$y_\theta : \mathcal{X}^n \to \mathcal{Y}^n \quad ,$$

where θ indicates the dependence on parameters.

Notation. The following notations are equivalent:

$$y_\theta(x) \equiv y(x;\theta) \quad .$$

Observation 3.2. Parameter space Θ is usually a subset of $\mathbb{R}^p$, where $p \in \mathbb{N}$ is the number of parameters in the neural network.

The specific definition of the function y_θ and the number of parameters p depends entirely on the architecture of the neural network. A neural network can be made up of a variety of layers, where each one performs a different set of operations and is defined by a different number of parameters.

Let $y_{\theta^{(1)}}, \dots, y_{\theta^{(l)}}$ be the functions of the layers that constitute the neural network, y_θ, where $l \in \mathbb{N}$ is the number of layers. Each layer, $y_{\theta^{(q)}}$, is parameterized by $\theta^{(q)}_{u_q}$, $u_q = 1, \dots, p_q$, where $p_q < p$ is the number of parameters of said layer. It is important to note that the parameters of the neural network consist of the parameters from all of its layers.:

$$\theta = \left(\theta^{(1)}_1, \dots, \theta^{(1)}_{p_1}, \dots, \theta^{(l)}_1, \dots, \theta^{(l)}_{p_l}\right) \quad . \tag{3.1.1}$$

A way of representing the function y_θ when the neural network is simply a concatenation of layers is:

$$y_\theta(x) = y_{\theta^{(l)}} \circ y_{\theta^{(l-1)}} \circ \cdots \circ y_{\theta^{(1)}}(x) \quad .$$

However, this notation is not valid for more complex neural network architectures, where layers intertwine (one layer's input might include the outputs of multiple previous layers, and its output might serve as input for multiple posterior layers). A way of denoting this is:

$$y(x;\theta) = y(\ \dots\ ,\ y(\ \dots\ ;\theta^{(q)})\ ,\ \dots\ ;\theta^{(l)}) \quad ,$$

where each $y_{\theta^{(q)}}$, $q = 2, \dots, l$, might take as input the output of previous layers $y_{\theta^{(\overline{q})}}$, $\overline{q} < q$, and/or inputs from the input space (except for the first layer that can only take inputs from the input space).

Due to this notation's complexity and lack of specificity, it is more common to use a diagram to represent the neural network's architecture. Figure 1 on Appendix A.1 shows the diagrams of a simple and a more complex neural network architectures.

We now expand the concept of neural network from a function to a function space, utilizing the property of being a parameterized function (following the idea introduced in Section 1.2).

Definition 3.3. The *function space generated by the neural network* y_θ, $\mathcal{V}$, consists of all functions that can be realized by adjusting different sets of values to the neural network's parameters. That is:

$$\mathcal{V} := \{y_\theta \mid \theta \in \Theta\} \quad ,$$

where each y_θ represents the function computed by the neural network with a particular configuration of parameters θ.

Remark. The complexity and capacity of $\mathcal{V}$ are influenced by the architecture of the neural network (e.g., number of layers, types of layers, their number of parameters). A more complex network can typically represent a wider variety of functions, thus increasing the "richness" of $\mathcal{V}$.

Furthermore, a significant theoretical aspect of neural networks, particularly relevant here, is the Universal Approximation Theorem [3]. It states that a sufficiently large and properly configured neural network can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to any desired degree of accuracy (given certain conditions). This implies that $\mathcal{V}$ for such networks is extremely broad, capable of approximating virtually any function within the space of continuous functions, assuming the network's architecture allows for it.

3.2 Transformer Architecture

A Transformer is a particular type of neural network that has a specific architecture, which was designed to have a good performance in sequence transduction tasks. In this section, we will conduct an in-depth study of the Transformer architecture, by explaining the reason and defining the expression of each of its constituent parts:

I. Embedding layer.

II. Multi-head attention layer.

III. Addition layer.

IV. Normalization layer.

V. Perceptron layer.

VI. Softmax normalization.

A Transformer takes a collection of n samples from the input space of a sequence transduction dataset, $\mathcal{X} = X_{v_S}^{m_S} \times X_{v_R}^{m_R}$ (2.2.2). Each sample is a pair of sequences of labels: the source sequence and the reference sequence.

Initially, the source sequence and the reference sequence are processed separately, each by one embedding layer. Then, the embedded source and reference sequences serve each as the input to a different layer compound. The source sequence is the input to the *encoder*, and the reference sequence to the *decoder*.

Inside both the encoder and the decoder there are self-attention mechanisms, where contextual information is obtained about each of the sequences (how the labels in a sequence relate to each other). Afterwards, the sequences are compared in the decoder's cross-attention mechanism, where the relationship between them is extracted (how each label of one sequence relates to the labels of the other).

Eventually, the decoder outputs a new sequence. This sequence is projected, and then normalized by the softmax function. The result is the output sequence of length m_R holding probability vectors of dimension v_R, which is an element of the output space $\mathbb{S}_{(v_R-1)}^{m_R}$.

Observation 3.4. The Figures on Appendix A.2 provide detailed illustrations of the architecture of a Transformer neural network and its individual layers.

It is worth mentioning that Figure 5 only shows one encoder and decoder compounds, but a Transformer model can have multiple of them stacked on top of one another. In that case, the input of any subsequent encoder would be the output of the previous one (the same happens with decoders), and the last encoder output would be used in all of the decoders' cross attention mechanisms. The number of compounds of each type is denoted by $\tau \in \mathbb{N}$, and it usually is the same for both encoders and decoders.

Observation 3.5. All the $\boxed{\text{boxed}}$ expressions below in this Section (within the layers of the Transformer) are implemented in C++ code in Apendix B.5.

I. Embedding layer

The embedding layer maps a finite set of labels onto a Euclidean space. The result is a point cloud in said Euclidean space where each point represents one of the labels. The distance of the Euclidean space is used to measure the similarity between labels, thus granting to an apparently arbitrary set of labels a relative sense of meaning. The layer also incorporates a positional encoding that helps retain the order of the sequence.

Figure 2 on Appendix A.1 illustrates an example of an embedding point cloud for a set of word labels.

We know that the inputs to the embedding layer are sequences of labels. The amount of labels in the set is called *vocabulary size*, denoted by $v \in \mathbb{N}$. That being so, the labels are represented by the elements of the set $X_v \subset \mathbb{Z}$ (2.2.1). The sequences have $m \in \mathbb{N}$ labels, which is the *input length*.

There are $n \in \mathbb{N}$ samples of sequences. Therefore, the input space of the layer is $X_v^{n\times m}$.

The dimension of the Euclidean space where we are to embed the labels is called *depth*, and it is denoted by $d \in \mathbb{N}$. The output consists on n sequences of length m of vectors from said Euclidean space, therefore the output space of the layer is $\mathbb{R}^{n\times m\times d}$.

Layer architecture

The embedding layer is parameterized by:

- $W \in \mathbb{R}^{v\times d}$, the embedding weights.

We define the *embedding lookup* function, $\phi : X_v \to \mathbb{R}^d$, as the function that assigns each integer number $x \in X_v$ to the x-th row of the matrix W. Its expression is:

$$\boxed{\phi(x; W) := \Big(W_{x,1}, W_{x,2}, \dots, W_{x,d}\Big)} \quad .$$

Since our data has a sequential nature, we add to each sample a constant (not learnable) *positional encoding matrix*, $PE \in \mathbb{R}^{m\times d}$, that provides a unique encoding for each position. This technique injects information about the position of elements in a sequence, thus allowing the model to better understand sequence dynamics. When adding this factor, the embeddings are scaled by $\sqrt{d}$ to balance both components. The positional encoding used in this case is based on sines and cosines of different frequencies. For each position $j = 1, \dots, n$, the positional encoding matrix is defined by:

$$\boxed{\begin{cases} PE_{j,k} = \sin\Big(j/10000^{2k/d}\Big) & , \quad 1 \leq k < {}^{d}\!/\!{}_{2} \\ PE_{j,k} = \cos\Big(j/10000^{2\hat{k}/d}\Big) & , \quad {}^{d}\!/\!{}_{2} \leq k \leq d \end{cases}} \quad ,$$

where $\hat{k} = k - \frac{d}{2}$.
The reason behind the use of this specific type of positional encoding (according to [1]) is that it allows the model to easily perceive relative positions, since for any fixed offset k, the row PE_{i+k} can be represented as a linear function of PE_i.

Observation 3.6. Embedding layers in general not always include positional encoding, but it is a key factor in sequence transduction models.

Layer function

Definition 3.7. The *function of an embedding layer* with samples number n, input length m, depth d, and vocabulary size v, is

$$y : X_v^{n\times m} \to \mathbb{R}^{n\times m\times d} \quad ,$$

such that,

$$y(x;W) := \Big(\phi(x_{i,j};W)_k \cdot \sqrt{d} + PE_{j,k}\Big)_{\substack{i=1,\dots,n \\ j=1,\dots,m \\ k=1,\dots,d}} \quad .$$

In matrix notation:

$$\boxed{y(x;W) = \phi(x;W) \cdot \sqrt{d} + PE} \quad ,$$

knowing that ϕ is applied to each element of x and that PE is added to each sample.

II. Multi-Head Attention layer

The multi-head attention layer, as its name suggests, contains multiple attention heads (or attention mechanisms). These attention heads compare the vector representations of two sequences (like the ones obtained through the embedding layer, item I), via their dot product. This way, we use the distance between points of a Euclidean space to get a measure of how relevant the labels at each position of a sequence are to each of the labels in the other sequence.

The result of an attention mechanism is illustrated in Figure 3 on Appendix A.1.

Remark. The two sequences being compared in a multi-head attention layer can be:

- The same reference or source sequence. In this case, we say the layer is a self-attention layer.
- A source sequence and its corresponding reference sequence. In this case, we say the layer is a cross-attention layer.

Because of this, we give the sequences a different name within the scope of the multi-head attention layer: *input* sequence and *context* sequence. In the case of cross-attention, the input sequence corresponds to the reference sequence, and the context sequence to the source sequence.

In any case, the two are paired, meaning each input sample has a corresponding context sample. Consecuently, we have the same number of samples for both of them, $n \in \mathbb{N}$. However, input sequences do not necessarily have the same length as context sequences. Thus, we have input length $m_I \in \mathbb{N}$ and context length $m_C \in \mathbb{N}$. Each entry in both has the same depth $d \in \mathbb{N}$.

As a result, the input space of the layer is $X = X_I \times X_C$, where $X_I \subset \mathbb{R}^{n \times m_I \times d}$, $X_C \subset \mathbb{R}^{n \times m_C \times d}$.

The layer has $h \in \mathbb{N}$ *attention heads*. It is relevant to mention that h is chosen as a divisor of the depth d to keep a balance on dimensionality while processing the data through the attention heads.

The outputs are transformations of the input sequences based on their comparisons with the context sequences. The dimensions of the input sequences are not changed through this transformations. Therefore, the output space of the layer is $\mathbb{R}^{n \times m_I \times d}$.

Layer architecture

We know that h is a divisor of d, so we define the layer's hidden depth as $\tilde{d} := \frac{d}{h}$. Then, the layer is parameterized by:

- $W_Q \in \mathbb{R}^{d \times \tilde{d} \times h}$ and $b_Q \in \mathbb{R}^{\tilde{d} \times h}$, the query weights and biases, respectively.
- $W_K \in \mathbb{R}^{d \times \tilde{d} \times h}$ and $b_K \in \mathbb{R}^{\tilde{d} \times h}$, the key weights and biases, respectively.
- $W_V \in \mathbb{R}^{d \times \tilde{d} \times h}$ and $b_V \in \mathbb{R}^{\tilde{d} \times h}$, the value weights and biases, respectively.
- $W_P \in \mathbb{R}^{\tilde{d} \times d \times h}$ and $b_P \in \mathbb{R}^{d}$, the projection weights and biases, respectively.

Let us consider $X \subset \mathbb{R}^{n \times m \times d}$, that could be either X_I or X_C. Then, we define the *linear transformation* function, $\mathcal{T} : X \to \hat{X} \subset \mathbb{R}^{n \times m \times \tilde{d} \times h}$, such that,

$$\mathcal{T}(x; W, b) := \left(\sum_{k=1}^{d} x_{i,j,k} \cdot W_{k,\tilde{k},r} + b_{\tilde{k},r} \right)_{\substack{i=1,\ldots,n \\ j=1,\ldots,m \\ \tilde{k}=1,\ldots,\tilde{d} \\ r=1,\ldots,h}} ,$$

where $W \in \mathbb{R}^{d \times \tilde{d} \times h}$ and $b \in \mathbb{R}^{\tilde{d} \times h}$ could be the query, key or value weights and biases, respectively.

We can express $\mathcal{T}$ in matrix notation as:

$$\boxed{\mathcal{T}(x; W, b) = \left(x \cdot W^{(r)} + b^{(r)}\right)_{r=1,\dots,h}} \quad .$$

Note that $\mathcal{T}$ transforms $x \in X$ through h different linear transformations. Each of them is parameterized by the weights $W^{(r)} \in \mathbb{R}^{d \times \tilde{d}}$ and the biases $b^{(r)} \in \mathbb{R}^{\tilde{d}}$. The end result of the transformation is h elements $\hat{x}^{(r)} \in \mathbb{R}^{n \times m \times \tilde{d}}$, each of which we say is the input to an attention head.

If $x_I \in X_I$ and $x_C \in X_C$ are the input and context, respectively, let us call:

- **Query:** $Q \equiv \mathcal{T}(x_I; W_Q, b_Q)$,
- **Key:** $K \equiv \mathcal{T}(x_C; W_K, b_K)$,
- **Value:** $V \equiv \mathcal{T}(x_C; W_V, b_V)$.

Each attention head gets one query, key and value, which we respectively denote $Q^{(r)} \in \mathbb{R}^{n \times m_I \times \tilde{d}}$, $K^{(r)} \in \mathbb{R}^{n \times m_C \times \tilde{d}}$, and $V^{(r)} \in \mathbb{R}^{n \times m_C \times \tilde{d}}$, with $r = 1, \dots, h$.

We will explain the attention computation performed by one attention head, since the same process is conducted by all of them in parallel:

1. First, a scaled dot product is performed between each sequence of the key and the corresponding sequence of the query. The results are called *attention scores*, $A^{(r)} \in \mathbb{R}^{n \times m_C \times m_I}$.

$$A^{(r)} \equiv \left(\tilde{d}^{-1/2} \cdot \sum_{\tilde{k}=1}^{\tilde{d}} K^{(r)}_{i,j_C,\tilde{k}} \cdot Q^{(r)}_{i,j_I,\tilde{k}}\right)_{\substack{i=1,\dots,n \\ j_C=1,\dots,m_C \\ j_I=1,\dots,m_I}} \quad .$$

In matrix notation,

$$A^{(r)} \equiv \frac{K^{(r)} \cdot Q^{(r)\intercal}}{\tilde{d}^{1/2}} \quad .$$

1.5 Since our data has a sequential nature, in some cases, we want to "hide" part of the attention scores so that each input sequence position can only attend to the previous context positions and not the ones ahead. This is achieved by adding a *causal mask*, $M \in \bar{\mathbb{R}}^{m_C \times m_I}$, to each sample of the attention scores. The causal mask is given by:

$$\boxed{M_{j_C,j_I} := \begin{cases} 0, & j_C \leq j_I \\ -\infty, & j_C > j_I \end{cases} \quad , \quad j_C = 1, \dots, m_C \ , \quad j_I = 1, \dots, m_I} \quad ,$$

where the masked positions are those where $M_{j_C,j_I} = -\infty$. The mask is applied to the attention scores in the following way:

$$A^{(r)} \equiv \left(\tilde{d}^{-1/2} \cdot \sum_{\tilde{k}=1}^{\tilde{d}} K^{(r)}_{i,j_C,\tilde{k}} \cdot Q^{(r)}_{i,j_I,\tilde{k}} + M_{j_C,j_I}\right)_{\substack{i=1,\dots,n \\ j_C=1,\dots,m_C \\ j_I=1,\dots,m_I}} \quad .$$

In matrix notation,

$$A^{(r)} \equiv \frac{K^{(r)} \cdot Q^{(r)\intercal}}{\tilde{d}^{1/2}} + M \quad .$$

2. No matter whether we have added the causal mask or not, we apply the softmax function (1.2.1) to each input position of each sample in the attention scores. The results are called *attention weights*, $\hat{A}^{(r)} \in \mathbb{S}^{n\times m_I}_{(m_C-1)} \subset \mathbb{R}^{n\times m_C\times m_I}$.

$$\hat{A}^{(r)} \equiv \Big(\mathcal{S}(A_{i,1,j_I}, A_{i,2,j_I}, \ldots, A_{i,m_C,j_I})\Big)_{\substack{i=1,\ldots,n\\ j_I=1,\ldots,m_I}} \quad . \tag{3.2.1}$$

   We know that the result of the softmax function is a probability vector. In this case, those probabilities represent how much attention each input position pays to each context position.

   Observation 3.8. If the causal mask was added, the masked positions have an attention weight of 0. This shows that each input position attends only to the preceding context positions.

3. Finally, a dot product is performed between the attention weights of each sample and its corresponding part of the value. The results are called *attention outputs*, $O^{(r)} \in \mathbb{R}^{n\times m_I\times \tilde{d}}$,

$$O^{(r)} \equiv \left(\sum_{j_C=1}^{m_C} \hat{A}^{(r)}_{i,j_C,j_I} \cdot V^{(r)}_{i,j_C,\tilde{k}}\right)_{\substack{i=1,\ldots,n\\ j_I=1,\ldots,m_I\\ \tilde{k}=1,\ldots,\tilde{d}}} \quad .$$

   In matrix notation,

$$O^{(r)} \equiv \hat{A}^{(r)\intercal} \cdot V^{(r)} \quad .$$

To this whole process is called *multi-head attention computation* [1], and it is represented by the function $\mathcal{M} : \hat{X}_I \times \hat{X}_C \times \hat{X}_C \to \hat{X}_I$, with $\hat{X}_I \subset \mathbb{R}^{n\times m_I\times\tilde{d}\times h}$ and $\hat{X}_C \subset \mathbb{R}^{n\times m_C\times\tilde{d}\times h}$. The abbreviated expression of $\mathcal{M}$ is:

$$\boxed{\mathcal{M}(Q,K,V) := \left(\mathcal{S}\Big(\frac{K^{(r)} \cdot Q^{(r)\intercal}}{\tilde{d}^{1/2}}\Big)^{\intercal} \cdot V^{(r)}\right)_{r=1,\ldots,h}} \quad , \tag{3.2.2}$$

knowing that $\mathcal{S}$ is applied to each input position (3.2.1) and that in some cases the causal mask M might be added to the attention scores. Note that $\mathcal{M}(Q,K,V) \equiv \Big(O^{(r)}\Big)_{r=1,\ldots,h}$.

Finally, the sequences are projected back to the original dimensions of the input, through the *linear projection* $\mathcal{P} : \hat{X}_I \to X_I \subset \mathbb{R}^{n\times m_I\times d}$, defined by:

$$\mathcal{P}(x; W_P, b_P) := \left(\sum_{r=1}^{h}\sum_{\tilde{k}=1}^{\tilde{d}} O^{(r)}_{i,j_I,\tilde{k}} \cdot {W_P}^{(r)}_{\tilde{k},k} + b_{Pk}\right)_{\substack{i=1,\ldots,n\\ j_I=1,\ldots,m_I\\ k=1,\ldots,d}} \quad .$$

Which can be expressed in matrix notation like:

$$\boxed{\mathcal{P}(x; W_P, b_P) = \sum_{r=1}^{h} O^{(r)} \cdot W_P^{(r)} + b_P} \quad .$$

Layer function

Definition 3.9. The *function of a multi-head attention layer* with samples number n, input length m_I, context length m_C, depth d, and attention heads number h, is

$$y : X_I \times X_C \to \mathbb{R}^{n\times m_I \times d} \quad ,$$

such that,

$$\boxed{\begin{aligned} y(x_I, x_C; &W_Q, b_Q, W_K, b_K, W_V, b_V, W_P, b_P) := \\ &:= \mathcal{P}\Big(\mathcal{M}\Big(\mathcal{T}(x_I; W_Q, b_Q), \mathcal{T}(x_C; W_K, b_K), \mathcal{T}(x_C; W_V, b_V)\Big); W_P, b_P\Big) \end{aligned}}$$

III. Addition layer

The addition layer aims to avoid the vanishing gradient problem (for more detail, see [4]) by granting more relevance to layers further from the output layer. It is also sometimes known as “residual connection” and it greatly benefits the training process of deep neural networks (with many layers).

The addition layer has $n \in \mathbb{N}$ samples of input sequences of length $m \in \mathbb{N}$ and depth $d \in \mathbb{N}$. Thus, the input space is $X = X_1 \times X_2$, where $X_1, X_2 \subset \mathbb{R}^{n\times m\times d}$.

The 2 inputs are added, combining features from different parts of the neural network.

Their dimensions are not transformed in the process, so the output space is $\mathbb{R}^{n\times m\times d}$.

Layer architecture

This layer does not have parameters or intermediate functions.

Layer function

Definition 3.10. The *function of an addition layer* with samples number n, input length m, and depth d, is

$$y : X_1 \times X_2 \to \mathbb{R}^{n\times m\times d},$$

such that,

$$\boxed{y\left(x_1, x_2\right) := x_1 + x_2}$$

IV. Normalization layer

The normalization layer aims to keep the magnitude of the data within reasonable limits to reduce numerical noise.

The input to the normalization layer consists of $n \in \mathbb{N}$ samples of sequences of length $m \in \mathbb{N}$ and depth $d \in \mathbb{N}$. Thus, the input space is $X \subset \mathbb{R}^{n\times m\times d}$.

Each input position is first normalized based on its mean and standard deviation. Then, the normalized inputs are scaled and shifted by the layer's learned parameters. This allows the layer to adapt to specific patterns that could arise in the data.

This process does not alter the input's dimensions, therefore the output space is also $\mathbb{R}^{n\times m\times d}$.

Layer architecture

The layer is parameterized by:

- $\gamma \in \mathbb{R}^d$, which define the scale of the layer.
- $\beta \in \mathbb{R}^d$, which define the shift of the layer.

First, we normalize the input entries through the *normalization* function, $\mathcal{N} : \mathbb{R}^d \to \mathbb{R}^d$, defined by:

$$\boxed{\mathcal{N}(x) := \left(\frac{x_k - \mu(x)}{\sigma(x) + \epsilon}\right)_{k=1,\dots,d}} \quad ,$$

where:

- $\mu : \mathbb{R}^d \to \mathbb{R}$ is the mean function:

$$\mu(x) := \frac{1}{d}\sum_{k=1}^{d} x_k$$

- $\sigma : \mathbb{R}^d \to \mathbb{R}$ is the standard deviation function:

$$\sigma(x) := \sqrt{\frac{1}{d}\sum_{k=1}^{d}(x_k - \mu(x))^2}$$

- $\epsilon > 0$ is a constant added for numerical stability (to avoid division by 0).

We denote:

$$\mathcal{N}(x) = \frac{x-\mu}{\sigma+\epsilon} \equiv \hat{x} \quad .$$

After that, we perform an *affine transformation* $\mathbf{A} : \mathbb{R}^d \to \mathbb{R}^d$ such that:

$$\mathbf{A}(\hat{x}; \gamma, \beta) := \Big(\hat{x}_k \cdot \gamma_k + \beta_k\Big)_{k=1,\dots,d} \quad .$$

In matrix notation,

$$\boxed{\mathbf{A}(\hat{x}; \gamma, \beta) = \hat{x} \cdot \gamma + \beta} \quad .$$

Layer function

Definition 3.11. The *function of a normalization layer* with samples number n, input length m, and depth d, is

$$y : X \to \mathbb{R}^{n\times m\times d} \quad ,$$

such that

$$\boxed{y\,(x; \gamma, \beta) := \Big(\mathbf{A}\Big(\mathcal{N}(x_{i,j}\Big)\Big)_{\substack{i=1,\dots,n\\ j=1,\dots,m}}} \quad ,$$

where $x_{i,j} = \Big(x_{i,j,1}, \dots, x_{i,j,d}\Big)$.

V. Perceptron layer

The perceptron layer (often called a "dense" layer) represents the concept of a layer of artificial neurons. Each artificial neuron is a simple representation of a biological one: it receives input signals, assigns a certain importance to each of them and depending on the magnitude of the combination of signals, the neuron is activated to a greater or lesser degree.

The input to the perceptron layer consists of $n \in \mathbb{N}$ samples of sequences of length $m \in \mathbb{N}$ and depth $d \in \mathbb{N}$, so the input space is $X \subset \mathbb{R}^{n\times m\times d}$.

The neurons receive vectors of depth d. Each neuron assigns a different weight to each of the d input features and adds them. A bias value is also added to each of the combinations. Then, every neuron outputs a single activation value based on the combination. Gathering the activations from all the neurons, we get a new vector with a number of features equal to the number of neurons, denoted by $\tilde{d} \in \mathbb{N}$.

Therefore, the outputs are n samples of sequences of length m and depth $\tilde{d}$, and the output space is $\mathbb{R}^{n\times m\times \tilde{d}}$.

Layer architecture

The layer is parameterized by:

- $W \in \mathbb{R}^{d \times \tilde{d}}$, the synaptic weights, where each column represents a neuron.
- $b \in \mathbb{R}^{\tilde{d}}$, the biases of the neurons.

We define the *combination* function $\mathcal{C} : X \to \mathbb{R}^{n \times m \times \tilde{d}}$ such that,

$$\mathcal{C}(x; W, b) := \Big(\sum_{k=1}^{d} x_{i,j,k} \cdot W_{k,\tilde{k}} + b_{\tilde{k}} \Big)_{\substack{i=1,\dots,n \\ j=1,\dots,m \\ \tilde{k}=1,\dots,\tilde{d}}} \quad .$$

We denote $C \equiv \mathcal{C}(x; W, b)$. It can be expressed in matrix notation as:

$$\boxed{C = x \cdot W + b} \quad .$$

The *activations* function $\mathcal{A} : \mathbb{R}^{n \times m \times \tilde{d}} \to \mathbb{R}^{n \times m \times \tilde{d}}$ is a function of the type:

$$\boxed{\mathcal{A}(x) := \Big(a(C_{i,j,k}) \Big)_{\substack{i=1,\dots,n \\ j=1,\dots,m \\ \tilde{k}=1,\dots,\tilde{d}}}} \quad ,$$

where the activation $a : \mathbb{R} \to \mathbb{R}$ can take different expressions.
In our model, perceptron layers have one of these 2 activations:

- **Linear (Identity):**
$$\boxed{a(x) \equiv \mathrm{Id}(x) = x}$$
- **Rectified Linear Unit (ReLU):**
$$\boxed{a(x) \equiv \mathrm{ReLU}(x) = \max(0, x)}$$

Figure 4 on Appendix A.1 illustrates the graphs of these activation functions.

Layer function

Definition 3.12. The *function of a perceptron layer* with samples number n, input length m, depth d, and neurons number $\tilde{d}$, is

$$y : X \to \mathbb{R}^{n \times m \times \tilde{d}} \quad ,$$

such that,

$$\boxed{y\,(x; W, b) := \mathcal{A}\Big(\mathcal{C}(x; W, b)\Big)} \quad .$$

VI. Softmax normalization

The softmax normalization transforms real-valued vectors into probability distributions.

The input of the softmax normalization are $n \in \mathbb{N}$ samples of sequences of length $m_R \in \mathbb{N}$ where each position holds a vector of dimension equal to the reference vocabulary size v_R, so the input space is $X \subset \mathbb{R}^{n\times m_R\times v_R}$.

The softmax function (1.2.1) is applied to the vectors at each position of each sample sequence separately. This way, we obtain at each position of a sequence a probability distribution over the elements of the vocabulary, X_v. Said probability distribution represents the likelihood that each element of the vocabulary occupies the corresponding position.

The softmax function does not change the dimensions of the input. However, the vectors of length v_R are normalized to form probability distributions, therefore they will now be vectors of the $(v_R - 1)$-simplex. That means the output space is $\mathbb{S}^{n\times m_R}_{(v_R-1)} \subset \mathbb{R}^{n\times m_R\times v_R}$.

Function

Definition 3.13. The *softmax normalization* with samples number n, input length m_R, and vocabulary size v_R,

$$y : X \to \mathbb{S}^{n\times m_R}_{(v_R-1)} \quad ,$$

applies the softmax function (1.2.1) in the following way:

$$\boxed{y\,(x) := \Big(\mathcal{S}(x_{i,j_R,1}, \dots, x_{i,j_R,v_R})\Big)^{i=1,\dots,n}_{j_R=1,\dots,m_R}} \quad .$$

Remark. In practice, the softmax normalization is not explicitly computed. This is because there is a more computationally efficient way to calculate it called the *log-softmax*. We explain this in detail in Section 4.2 (specifically, the expression is (4.2.3)).

Observation 3.14. We use the softmax function since our model performs multiple classification. In other cases where classification is binary, it is common to normalize the outputs using the more simple *logistic* function,

$$f(z) = \frac{1}{1+e^{-z}} \quad ,$$

where $z \in \mathbb{R}$ would be the pre-logistic output of the neural network.

4 | Loss Index

The loss index is another main component of a neural network's training process. Its purpose is to quantify the discrepancy between the neural network's predicted values and the expected targets of a dataset. Through the loss index, we can deduce how the neural network's parameters can be modified to improve its proficiency at the task. The loss index can include a term that promotes the regularization of parameters to achieve smoother results.

Throughout this chapter, we define the concept of loss index, describe how it is used to improve neural networks, and elaborate deeply on its role in Transformers.

4.1 Definition

To define the loss index, we first need to define its two components: error and regularization.

Let D be a dataset with n samples $(x_i, \hat{y}_i)$, with input space $\mathcal{X}$ and output space $\mathcal{Y}$. We denote $x \in \mathcal{X}^n$, $\hat{y} \in \mathcal{Y}^n$ the collections of inputs and targets of the dataset, respectively. Let $y_\theta : \mathcal{X}^n \to \mathcal{Y}^n$ be a neural network and $\mathcal{V} = \{y_\theta \mid \theta \in \Theta\}$ its corresponding function space.

Both the error and regularization (therefore, the loss index) are defined as functionals that act on the function space $\mathcal{V}$ and assign a positive real number to each function.

We start by giving a definition of the error functional.

Definition 4.1. For any $x \in \mathcal{X}^n$, $\hat{y} \in \mathcal{Y}^n$, an *error functional* on $\mathcal{V}$, $E_{x,\hat{y}} : \mathcal{V} \to \mathbb{R}^+$, is a functional such that it can be expressed as:

$$E_{x,\hat{y}}[y_\theta] := e(y_\theta(x), \hat{y}) \quad ,$$

where $e : \mathcal{Y}^n \times \mathcal{Y}^n \to \mathbb{R}^+$ is an *error function*, verifying:

- $e(y, \hat{y}) \geq 0$ for all $y, \hat{y} \in \mathcal{Y}^n \quad ,$
- $e(y, \hat{y}) = 0$ if and only if $y = \hat{y} \quad .$

We say that e is the error function associated with $E_{x,\hat{y}}$, and it does not depend on the specific choice of x and $\hat{y}$; these only determine the point in which e is evaluated for each $y_\theta \in \mathcal{V}$.

Observation 4.2. The expression of the error function e may vary depending on the type of task of the neural network.

We will now give a definition of the regularization functional.

Definition 4.3. A *regularization functional* on $\mathcal{V}$, $\Omega : \mathcal{V} \to \mathbb{R}^+$, is a functional such that it can be expressed as:

$$\Omega[y_\theta] := \omega(\theta) \quad ,$$

where $\omega : \Theta \to \mathbb{R}^+$ is a *regularization function*, verifying:

- $\omega(\theta) \geq 0$ for all $\theta \in \Theta \quad ,$
- $\omega(\theta) = 0$ if and only if $\theta = \mathbf{0}$, with $\mathbf{0}$ the null element of Θ.

We say that ω is the error function associated with Ω.

Observation 4.4. The regularization function is usually a norm of the parameters. The type of norm to be utilized depends on the degree of smoothness desired for the neural network after training.

Examples. Some common error and regularization functions are:

1.1. **Mean squared error**: Used in regression tasks ($\mathcal{Y}^n = \mathbb{R}^n$),

$$e(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \quad ,$$

1.2. **Log error**: Used in binary classification tasks ($\mathcal{Y}^n = [0,1]^n$),

$$e(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n} -(\hat{y}_i \cdot \log(y_i) + (1 - \hat{y}_i) \cdot \log(1 - y_i)) \quad , \tag{4.1.1}$$

Let p be the number of parameters ($\Theta = \mathbb{R}^p$).

2.1. **L1 regularization:** Achieves rougher results:

$$\omega(\theta) = \|\theta\|_1 = \sum_{u=1}^{p} |\theta_u| \quad .$$

2.2. **L2 regularization:** Achieves smoother results:

$$\omega(\theta) = \|\theta\|_2 = \sqrt{\sum_{u=1}^{p} \theta_u^2} \quad .$$

Finally, let us define the loss index.

Definition 4.5. Given a neural network y_θ, with its corresponding function space $\mathcal{V}$, for any $x \in \mathcal{X}$, $\hat{y} \in \mathcal{Y}$ a *loss index* is a functional $\mathcal{L}_{x,\hat{y}} : \mathcal{V} \to \mathbb{R}^+$ that can be expressed as:

$$\mathcal{L}_{x,\hat{y}}[y_\theta] = E_{x,\hat{y}}[y_\theta] + \Omega[y_\theta] \quad ,$$

where:

- $E_{x,\hat{y}} : \mathcal{V} \to \mathbb{R}^+$ is an *error functional*,
- $\Omega : \mathcal{V} \to \mathbb{R}^+$ is a *regularization functional.*

We can now formulate the training process of a neural network as the following variational problem on the neural network's function space:

Problem 4.6. Let $x \in \mathcal{X}^n$ and $\hat{y}^n \in \mathcal{Y}$, and let $y_\theta : \mathcal{X}^n \to \mathcal{Y}^n$ be a neural network with $\mathcal{V} = \{y_\theta \mid \theta \in \Theta\}$ its corresponding function space. Given a loss index $\mathcal{L}_{x,\hat{y}} : \mathcal{V} \to \mathbb{R}^+$, find $y_{\theta^*} \in \mathcal{V}$ such that:

$$y_{\theta^*} = \arg\min_{y_\theta \in \mathcal{V}} \mathcal{L}_{x,\hat{y}}[y_\theta] \quad .$$

Remark. The loss index $\mathcal{L}_{x,\hat{y}} = E_{x,\hat{y}} + \Omega$, can be expressed as:

$$\mathcal{L}_{x,\hat{y}}[y_\theta] = e(y_\theta(x), \hat{y}) + \omega(\theta) \quad ,$$

where e is the error function associated with $E_{x,\hat{y}}$ and ω is the regularization function associated with Ω. If we now fix the inputs $x \in \mathcal{X}^n$ and targets $\hat{y} \in \mathcal{Y}^n$, the previous expression only depends on the specific choice of parameters $\theta \in \Theta$. Hence, the following definition:

Definition 4.7. For any fixed inputs and targets, $x, \hat{y}$, the *loss function* associated with the loss index $\mathcal{L}_{x,\hat{y}} : \mathcal{V} \to \mathbb{R}^+$ is a function $L : \Theta \to \mathbb{R}^+$ such that:

$$L(\theta) := e(y(x;\theta), \hat{y}) + \omega(\theta) \quad , \tag{4.1.2}$$

where e is the error function associated with $E_{x,\hat{y}}$, the error functional of $\mathcal{L}_{x,\hat{y}}$, and ω is the regularization function associated with Ω, the regularization functional of $\mathcal{L}_{x,\hat{y}}$. Since the inputs and targets are now fixed, there is no longer need to denote the dependence of L on x and $\hat{y}$.

Then, we can reduce the neural network training Problem 4.6 from a variational problem on the function space $\mathcal{V}$ to the following optimization problem on the parameter space Θ:

Problem 4.8. Let D be a dataset with n samples, with $x \in \mathcal{X}^n$, $\hat{y} \in \mathcal{Y}^n$ its collections of inputs and targets, respectively, and let $y_\theta : \mathcal{X}^n \to \mathcal{Y}^n$ be a neural network. Given a loss function $L : \Theta \to \mathbb{R}^+$, find $\theta^* \in \Theta$ such that:

$$\theta^* = \arg\min_{\theta \in \Theta} L(\theta) \quad .$$

Observation 4.9. In this case, the collection of inputs and targets from the dataset are the fixed x and $\hat{y}$ upon which the loss function L is defined.

Back-Propagation Algorithm

The back-propagation algorithm is a method to calculate the derivatives of the error function with respect to every parameter of a neural network. It is based on the principle that a neural network is a composition of layers, each parameterized by a collection of the total parameters of the neural network.

In order to solve Problem 4.8, we will apply an optimization algorithm. The concept of what an optimization algorithm is and how it is applied will be explained in detail in Chapter 5. However, it is relevant to mention now that a necessary component for said algorithm is the gradient of the loss function with respect to the parameters,

$$\nabla_\theta L = \left(\frac{\partial L}{\partial \theta_1}, \dots, \frac{\partial L}{\partial \theta_p} \right) \quad .$$

For every parameter θ_u , $u = 1, \dots, p$, we have:

$$\frac{\partial L}{\partial \theta_u} = \frac{\partial e}{\partial \theta_u} + \frac{\partial \omega}{\partial \theta_u} \quad .$$

Then,

$$\nabla_\theta L = \nabla_\theta e + \nabla_\theta \omega \quad .$$

The gradient of the regularization term, $\nabla_\theta \omega = \left(\frac{\partial \omega}{\partial \theta_1}, \dots, \frac{\partial \omega}{\partial \theta_p} \right)$, is easily calculated based on the expression of the chosen regularization function.

As for the gradient of the error term, $\nabla_\theta e = \left(\frac{\partial e}{\partial \theta_1}, \dots, \frac{\partial e}{\partial \theta_p} \right)$, we employ the back-propagation algorithm:

Algorithm 4.10 (Back-Propagation Algorithm)**.** The goal of the back-propagation algorithm is to calculate the gradient of the error function, e, with respect to the parameters, θ, of a neural network y_θ.

Let $y_{\theta^{(1)}}, \dots, y_{\theta^{(l)}}$ be the functions of the layers that that make up the architecture of y_θ. Each layer $y_{\theta^{(q)}}$, $q = 1, \dots, l$ is parameterized by $\theta_1^{(q)}, \dots, \theta_{p_q}^{(q)}$, where $p_q < p$ is the number of parameters of said layer. We denote $\Delta^{(q)} \equiv \frac{\partial e}{\partial y_{\theta^{(q)}}}$.

First, we calculate the *output deltas*, denoted by Δ^O,

$$\Delta^O = \Delta^{(l)} = \frac{\partial e}{\partial y_{\theta^{(l)}}} = \frac{\partial e}{\partial y_\theta} \quad ,$$

which only depend on the specific expression of the chosen error function e.

Then, for each $q = l, l-1, \dots, 1$, the steps of the back-propagation algorithm are as follows:

1. If $q = l$, skip to Step 2. If not, then the output of the layer $y_{\theta^{(q)}}$ is the input to a subsequent layer $y_{\theta^{(\overline{q})}}$, $\overline{q} > q$. That is:

$$y_{\theta^{(q)}}(x^{(q)}) = x^{(\overline{q})} \quad ,$$

where $x^{(q)}$ denotes the input of the q-th layer. Therefore, the output derivatives of the q-th layer are also the input derivatives of the $\overline{q}$-th layer, which has already been calculated in a previous iteration:

$$\Delta^{(q)} = \frac{\partial e}{\partial y_{\theta^{(q)}}} = \frac{\partial e}{\partial x^{(\overline{q})}} \quad ,$$

where $\frac{\partial e}{\partial x^{(\overline{q})}}$ is already known.

Note that the output of $y_{\theta^{(q)}}$ might be used as input by more than one subsequent layer, $y_{\theta^{(\overline{q})}}$, with different values of $\overline{q}$. In that case, $\Delta^{(q)}$ is the sum of $\frac{\partial e}{\partial x^{(\overline{q})}}$ for all values of $\overline{q}$.

2. We calculate the derivatives of the error with respect to the parameters of the q-th layer, $\theta^{(q)}_{u_q}$, $u_q = 1, \dots, p_q$:

$$\frac{\partial e}{\partial \theta^{(q)}_{u_q}} = \frac{\partial y_{\theta^{(q)}}}{\partial \theta^{(q)}_{u_q}} \frac{\partial e}{\partial y_{\theta^{(q)}}} = \frac{\partial y_{\theta^{(q)}}}{\partial \theta^{(q)}_{u_q}} \cdot \Delta^{(q)} \quad ,$$

where we can calculate $\frac{\partial y_{\theta^{(q)}}}{\partial \theta^{(q)}_{u_q}}$ based on the specific expression of the function $y_{\theta^{(q)}}$.

3. If $y_{\theta^{(q)}}$ is an input layer (its inputs come only from the dataset), continue to the next iteration. If not, we calculate the derivatives of the error with respect to the inputs of the q-th layer:

$$\frac{\partial e}{\partial x^{(q)}} = \frac{\partial y_{\theta^{(q)}}}{\partial x^{(q)}} \cdot \Delta^{(q)} \quad .$$

where we can also calculate $\frac{\partial y_{\theta^{(q)}}}{\partial x^{(q)}}$ based on the specific expression of the function $y_{\theta^{(q)}}$.

When the algorithm is finished, we will have calculated $\frac{\partial e}{\partial \theta^{(q)}_{u_q}}$, $u_q = 1, \dots, p_q$, for every layer $q = 1, \dots, l$. Therefore, since the parameters of the neural network consist of the parameters from all of its layers (3.1.1), we have the gradient of the error with respect to the neural network's parameters:

$$\nabla_\theta e = \left(\frac{\partial e}{\partial \theta_1}, \dots, \frac{\partial e}{\partial \theta_p} \right) \quad .$$

Hence, we have the full gradient of the loss function with respect to the parameters, $\nabla_\theta L$.

4.2 Cross-Entropy Error

We know that the expression of a loss index is determined by the choice of the corresponding error and regularization functions, which in turn depend on the task and desired smoothness of the neural network, respectively. For Transformers, smoothness is not a relevant factor, so we will not utilize any regularization method. Then, the loss index only has the error term.

Transformers are sequence transduction models, which is a generalization of multiple classification (it is multiple classification for each position of the sequence). Thus, the error function employed for its training is the *cross-entropy* error, which is a generalization of the log error (4.1.1). In this section, we will define the cross-entropy error, justify its use and calculate its output deltas.

We start by introducing the concept of cross-entropy:

Definition 4.11. Let p be a discrete probability distribution of a set of events $\mathbb{E}$. Given another discrete probability distribution q over the same set of events $\mathbb{E}$, the *cross-entropy* of the distribution q relative to the distribution p is defined as:

$$H(p,q) := -\sum_{e\in\mathbb{E}} p(e)\cdot\log(q(e)) \quad . \tag{4.2.1}$$

Here, p represents the "true" distribution of the events of $\mathbb{E}$ and q represents an estimated distribution of them. Formally, the cross entropy of q relative to p, $H(p,q)$, measures the average number of bits needed to identify an event drawn from the set $\mathbb{E}$ when the coding scheme used for the set is optimized for an estimated probability distribution q, rather than the true distribution p [5]. What this means is that $H(p,q)$ can be used to quantify the difference between the probability distributions p and q.

We can now define the cross-entropy error function.

Definition 4.12. Let $\mathcal{Y}^n = \mathbb{S}^{n\times m_r}_{(v_R-1)}$ be the output space of a sequence transduction neural network, where n is the samples number, v_R is the reference vocabulary size, m_R is the reference sequence length, and $\mathbb{S}_{(v_R-1)}$ is the (v_r-1)-simplex. Then, *cross-entropy* error function, $e:\mathcal{Y}^n\times\mathcal{Y}^n\to\mathbb{R}^+$, is given by the expression:

$$e(y,\hat{y}) := \frac{1}{n\cdot m_R}\cdot\sum_{i=1}^{n}\sum_{j_R=1}^{m_R} H(\hat{y}_{i,j_R}, y_{i,j_R}) \quad ,$$

where $\hat{y}_{i,j_R}, y_{i,j_R}\in\mathbb{S}_{(v_R-1)}$ are discrete probability distributions over the set X_{v_R}

and H is the cross-entropy. Then, substituting the expression (4.2.1),

$$e(y,\hat{y}) = \frac{1}{n\cdot m_R}\cdot\sum_{i=1}^{n}\sum_{j_R=1}^{m_R}\left(\sum_{s_R=1}^{v_R} -\hat{y}_{i,j_R,s_R}\cdot\log(y_{i,j_R,s_R})\right) \quad . \tag{4.2.2}$$

Proposition 4.13 (Log-Softmax)**.** *Let D be a sequence transduction dataset with n samples $(x_i,\hat{y}_i)$, and y_θ a Transformer neural network (whose outputs are normalized through the softmax function). Then, denoting by z the pre-softmax outputs of $y_\theta(x)$, the cross-entropy error function can be simplified to:*

$$\boxed{e(y_\theta(x),\hat{y}) = \frac{1}{n\cdot m_R}\cdot\sum_{i=1}^{n}\sum_{j_R=1}^{m_R}\log\left(\sum_{s_R=1}^{v_R} e^{\hat{z}_{i,j_R,s_R}}\right) - \hat{z}_{i,j_R,t_{i,j_R}}} \quad , \tag{4.2.3}$$

where $t_{i,j_R}\in X_{v_R}$ is the index of the vertex $\hat{y}_{i,j_R}\in\mathbb{S}_{(v_R-1)}$ (i.e. $\hat{y}_{i,j_R}$ is the t_{i,j_R}-th vertex of the (v_R-1)-simplex) and $\hat{z}_{i,j_R,s_R} = z_{i,j_R,s_R} - \max_{\bar{s}_R=1}^{v_R}(z_{i,j_R,\bar{s}_R})$.

This is known as the "log-softmax" expression.

Proof. We know that the targets of a sequence transduction dataset, $\hat{y}_i$, are sequences of m_R vertices of the (v_R-1)-simplex, $\mathbb{S}_{(v_R-1)}$. These vertices are vectors with all of their components equal to 0 except for one, which is equal to 1. Then, when applying the cross-entropy error function (4.2.2) onto said targets, $\hat{y}$, its expression can be simplified as:

$$e(y,\hat{y}) = \frac{1}{n\cdot m_R}\cdot\sum_{i=1}^{n}\sum_{j_R=1}^{m_R} -\log\Big(y_{i,j_R,t_{i,j_R}}\Big) \quad ,$$

where $t_{i,j_R}\in X_{v_R}$ is the index at which the vector $\hat{y}_{i,j_R}$ has a 1 (i.e. $\hat{y}_{i,j_R}$ is the t_{i,j_R}-th vertex of the (v_R-1)-simplex).

We also know that a softmax normalization has been applied to the neural network outputs. Then, denoting the inputs from the dataset by x and the pre-softmax outputs by z, the previous expression can be simplified further by substituting the softmax expression (1.2.1) of the outputs:

$$\begin{aligned} e(y_\theta(x),\hat{y}) &= \frac{1}{n\cdot m_R}\cdot\sum_{i=1}^{n}\sum_{j_R=1}^{m_R} -\log\left(\frac{e^{\hat{z}_{i,j_R,t_{i,j_R}}}}{\sum_{s_R=1}^{v_R} e^{\hat{z}_{i,j_R,s_R}}}\right) \\ &= \frac{1}{n\cdot m_R}\cdot\sum_{i=1}^{n}\sum_{j_R=1}^{m_R}\log\left(\sum_{s_R=1}^{v_R} e^{\hat{z}_{i,j_R,s_R}}\right) - \hat{z}_{i,j_R,t_{i,j_R}} \quad , \end{aligned}$$

where $\hat{z}_{i,j_R,s_R} = z_{i,j_R,s_R} - \max_{\bar{s}_R=1}^{v_R}(z_{i,j_R,\bar{s}_R})$, which is the log-softmax expression. □

The log-softmax is the expression that is actually computed at the end of the neural network during training, instead of first computing the softmax normalization and then the cross-entropy error, since it is more computationally efficient.

Observation 4.14. We know that when inputs and targets are fixed, we can consider the error as a function of the parameters (4.1.2). In that case, minimizing the cross-entropy (as a function of the parameters) is equivalent to maximizing the likelihood estimation of the targets with respect to the parameters [6].

Output deltas

We will now calculate the output deltas of the cross-entropy error. We assume the neural network to be a Transformer. Since the deltas are calculated only during training (where we necessarily have a dataset), we also assume to have a sequence transduction dataset.

Proposition 4.15. *Given the conditions and notation of Proposition 4.13, the output deltas of the log-softmax expression of the cross-entropy error function (4.2.3) are:*

$$\boxed{\Delta^O = \left(\frac{1}{n \cdot m_R} \cdot \left(y_{i,j_R,s_R} - \mathbf{1}\{s_R = t_{i,j_R}\}\right)\right)_{\substack{i=1,\dots,n \\ j_R=1,\dots,m_R \\ s_R=1,\dots,v_R}}} \quad . \tag{4.2.4}$$

Proof. The output deltas of the log-softmax expression are $\Delta^O = \frac{\partial e}{\partial z}$. Let $y \equiv y_\theta(x)$ be the output of the neural network (post-softmax). Then,

$$\begin{aligned}
\frac{\partial e}{\partial z_{\bar{i},\overline{j_R},\overline{s_R}}} &= \frac{1}{n \cdot m_R} \cdot \left(\frac{e^{\hat{z}_{\bar{i},\overline{j_R},\overline{s_R}}}}{\sum\limits_{s_R=1}^{v_R} e^{\hat{z}_{\bar{i},\overline{j_R},s_R}}} - \mathbf{1}\{\overline{s_R} = t_{i,j_R}\} \right) \\
&= \frac{1}{n \cdot m_R} \cdot \left(y_{\bar{i},\overline{j_R},\overline{s_R}} - \mathbf{1}\{\overline{s_R} = t_{\bar{i},\overline{j_R}}\} \right) \quad .
\end{aligned}$$

Note that we have simplified:

$$\sum_{\bar{i}=1}^{n} \sum_{\overline{j_R}=1}^{m_R} \mathbf{1}\{\bar{i} = i\} \cdot \mathbf{1}\{\overline{j_R} = j_R\} \cdot \frac{e^{\hat{z}_{i,j_R,\overline{s_R}}}}{\sum\limits_{s_R=1}^{v_R} e^{\hat{z}_{i,j_R,s_R}}} \quad = \quad \frac{e^{\hat{z}_{\bar{i},\overline{j_R},\overline{s_R}}}}{\sum\limits_{s_R=1}^{v_R} e^{\hat{z}_{\bar{i},\overline{j_R},s_R}}} \quad .$$

Then, the output deltas are given by (4.2.4). □

Observation 4.16. The log-softmax (4.2.3) and its output deltas (4.2.4) are implemented in C++ code in Apendix B.6.

4.3 Transformer Back-Propagation

We have already seen the Transformer architecture and all of its components, as well as the error function that we use for its training. In order to continue with the training process, we must apply the back-propagation Algorithm 4.10 to the neural network.

In this section, we will calculate the derivatives of the error with respect to the parameters of a Transformer neural network by applying the back-propagation algorithm. For this, we will calculate the necessary derivatives of each layer, starting from the end perceptron layer and ending with the starting embedding layers.

For each layer, we will assume to have its deltas, denoted simply by Δ, and we will first calculate its parameters derivatives, followed by its inputs derivatives. The two notable exceptions are: the end perceptron layer, in which the deltas correspond to the output deltas of the cross-entropy error (4.2.4), and the embedding layers, in which we will not calculate the input derivatives because they are not necessary to continue the algorithm.

Observation 4.17. All the $\boxed{\text{boxed}}$ expressions below in this Section (within the back-propagation of the Transformer) are implemented in C++ code in Apendix B.7.

V. Perceptron layer

Proposition 4.18 (Perceptron Parameters Derivatives)**.** *Given the perceptron layer function $y(x; W, b)$ (3.12), its parameters derivatives are:*

(1) Synaptic weights:

$$\boxed{\frac{\partial e}{\partial W}\Big(y(x;W,b)\Big) = \left(\sum_{i=1}^{n}\sum_{j=1}^{m} x_{i,j,k} \cdot \frac{\partial a}{\partial C_{i,j,\tilde{k}}} \cdot \Delta_{i,j,\tilde{k}}\right)_{\substack{k=1,\dots,d\\ \tilde{k}=1,\dots,\tilde{d}}}} \quad , \qquad (4.3.1)$$

(2) Biases:

$$\boxed{\frac{\partial e}{\partial b}\Big(y(x;W,b)\Big) = \left(\sum_{i=1}^{n}\sum_{j=1}^{m} \frac{\partial a}{\partial C_{i,j,\tilde{k}}} \cdot \Delta_{i,j,\tilde{k}}\right)_{\tilde{k}=1,\dots,\tilde{d}}} \quad , \qquad (4.3.2)$$

where a is the chosen activation function and C is the combination function.

Proof. We must calculate the derivatives of the error with respect to:

(1) Synaptic weights:

$$\frac{\partial e}{\partial W} = \frac{\partial y}{\partial W}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{A}}{\partial W} \cdot \Delta = \frac{\partial \mathcal{C}}{\partial W}\frac{\partial \mathcal{A}}{\partial C} \cdot \Delta \quad ,$$

(2) Biases:

$$\frac{\partial e}{\partial b} = \frac{\partial y}{\partial b}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{A}}{\partial b} \cdot \Delta = \frac{\partial \mathcal{C}}{\partial b}\frac{\partial \mathcal{A}}{\partial C} \cdot \Delta \quad .$$

First, let us calculate the activation derivatives $\frac{\partial \mathcal{A}}{\partial C}$. Since a is applied independently to each $C_{i,j,k}$,

$$\frac{\partial \mathcal{A}_{i,j,\tilde{k}}}{\partial C_{\bar{i},\bar{j},\bar{\tilde{k}}}}(C) = \mathbf{1}\{\bar{i} = i\} \cdot \mathbf{1}\{\bar{j} = j\} \cdot \mathbf{1}\{\bar{\tilde{k}} = \tilde{k}\} \cdot \frac{\partial a}{\partial C_{\bar{i},\bar{j},\bar{\tilde{k}}}}(C_{i,j,\tilde{k}}) \quad .$$

When multiplied with Δ, the previous expression simplifies to:

$$\frac{\partial \mathcal{A}}{\partial C} \cdot \Delta(C) = \frac{\partial e}{\partial C}(C) = \left(\frac{\partial a}{\partial C_{i,j,\tilde{k}}}(C_{i,j,\tilde{k}}) \cdot \Delta_{i,j,\tilde{k}} \right)_{\substack{i=1,\dots,n \\ j=1,\dots,m \\ \tilde{k}=1,\dots,\tilde{d}}} \quad . \tag{4.3.3}$$

Now, let us calculate $\frac{\partial \mathcal{C}}{\partial W}$ and $\frac{\partial \mathcal{C}}{\partial b}$.

$$\frac{\partial \mathcal{C}_{i,j,\tilde{k}}}{\partial W_{k,\bar{\tilde{k}}}}(x; W, b) = x_{i,j,k} \cdot \mathbf{1}\{\bar{\tilde{k}} = \tilde{k}\} \quad ,$$

$$\frac{\partial \mathcal{C}_{i,j,\tilde{k}}}{\partial b_{\bar{\tilde{k}}}}(x; W, b) = \mathbf{1}\{\bar{\tilde{k}} = \tilde{k}\} \quad .$$

Multiplying $\frac{\partial \mathcal{C}}{\partial W}$ and $\frac{\partial \mathcal{C}}{\partial b}$ with $\frac{\partial e}{\partial C}$, we conclude that the final expressions are given by (4.3.1) and (4.3.2). □

Proposition 4.19 (Perceptron Inputs Derivatives). *Given the perceptron layer function $y(x; W, b)$ (3.12), its inputs derivatives are:*

$$\boxed{\frac{\partial e}{\partial x}\Big(y(x; W, b)\Big) = \left(\sum_{\tilde{k}=1}^{\tilde{d}} \frac{\partial a}{\partial C_{i,j,\tilde{k}}} \cdot \Delta_{i,j,\tilde{k}} \cdot W_{k,\tilde{k}} \right)_{\substack{i=1,\dots,n \\ j=1,\dots,m \\ k=1,\dots,d}}} \quad , \tag{4.3.4}$$

where a is the chosen activation function and $\mathcal{C}$ is the combination function.

Proof. We must calculate the derivatives of the error with respect to the input, x:

$$\frac{\partial e}{\partial x} = \frac{\partial y}{\partial x}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{A}}{\partial x} \cdot \Delta = \frac{\partial \mathcal{C}}{\partial x}\frac{\partial \mathcal{A}}{\partial C} \cdot \Delta = \frac{\partial \mathcal{C}}{\partial x}\frac{\partial e}{\partial C} \quad .$$

We have already calculated $\frac{\partial e}{\partial C}$ (4.3.3), so only $\frac{\partial \mathcal{C}}{\partial x}$ remains:

$$\frac{\partial \mathcal{C}_{i,j,\tilde{k}}}{\partial x_{\bar{i},\bar{j},k}}(x; W, b) = \mathbf{1}\{\bar{i} = i\} \cdot \mathbf{1}\{\bar{j} = j\} \cdot W_{k,\tilde{k}} \quad .$$

Multiplying $\frac{\partial \mathcal{C}}{\partial x}$ with $\frac{\partial e}{\partial x}$, we get the expression in the statement (4.3.4). □

For the case of the Transformer model, the activation function can be either the linear activation or the ReLU activation. Then, we must calculate $\frac{\partial a}{\partial x}$ for both cases. Let $c \in \mathbb{R}$:

$$\boxed{\frac{d\mathrm{Id}}{dc}(c) = 1} \quad ,$$

$$\boxed{\frac{d\mathrm{ReLU}}{dc}(c) = \begin{cases} 0, & c < 0 \\ 1, & c > 0 \end{cases} = \mathbf{1}\{c > 0\}} \quad .$$

IV. Normalization layer

Proposition 4.20 (Normalization Parameters Derivatives)**.** *Given the normalization layer function $y(x; \gamma, \beta)$ (3.11), its parameters derivatives are:*

(1) Gammas:

$$\boxed{\frac{\partial e}{\partial \gamma}\big(y(x; \gamma, \beta)\big) = \left(\sum_{i=1}^{n}\sum_{j=1}^{m} \hat{x}_{i,j,k} \cdot \Delta_{i,j,k}\right)_{k=1,\dots,d}} \quad , \qquad (4.3.5)$$

(2) Betas:

$$\boxed{\frac{\partial e}{\partial \beta}\big(y(x; \gamma, \beta)\big) = \left(\sum_{i=1}^{n}\sum_{j=1}^{m} \Delta_{i,j,k}\right)_{k=1,\dots,d}} \quad , \qquad (4.3.6)$$

where $\hat{x}$ are the normalized inputs.

Proof. We must calculate the derivatives of the error with respect to:

(1) Gammas:

$$\frac{\partial e}{\partial \gamma} = \frac{\partial y}{\partial \gamma}\frac{\partial e}{\partial y} = \frac{\partial \mathbf{A}}{\partial \gamma} \cdot \Delta \quad ,$$

(2) Betas:

$$\frac{\partial e}{\partial \beta} = \frac{\partial y}{\partial \beta}\frac{\partial e}{\partial y} = \frac{\partial \mathbf{A}}{\partial \beta} \cdot \Delta \quad .$$

Since $\mathbf{A}$ is applied to each entry separately, let us calculate its derivatives with respect to a $\hat{x} \in \mathbb{R}^d$:

$$\frac{\partial \mathbf{A}_k}{\partial \gamma_{\overline{k}}}\Big(\hat{x};\gamma,\beta\Big) = \mathbf{1}\{\overline{k}=k\}\cdot x_{\overline{k}} \quad ,$$
$$\frac{\partial \mathbf{A_k}}{\partial \beta_{\overline{k}}}\Big(\hat{x};\gamma,\beta\Big) = \mathbf{1}\{\overline{k}=k\} \quad .$$

Multiplying $\frac{\partial \mathbf{A}}{\partial \gamma}$ and $\frac{\partial \mathbf{A}}{\partial \beta}$ with the deltas Δ , we conclude that the final expressions are given by (4.3.5) and (4.3.6). □

Proposition 4.21 (Normalization Inputs Derivatives)**.** *Given the normalization layer function $y(x;\gamma,\beta)$ (3.11), its inputs derivatives are:*

$$\boxed{\begin{aligned}&\frac{\partial e}{\partial x}\Big(y(x;\gamma,\beta)\Big) = \\ &= \left(\frac{\gamma_k\cdot\Delta_{i,j,k}}{\sigma_{i,j}+\epsilon} - \sum_{\overline{k}=1}^{d}\frac{\gamma_{\overline{k}}\cdot\Delta_{i,j,\overline{k}}}{d\cdot(\sigma_{i,j}+\epsilon)} - \sum_{\overline{k}=1}^{d}\frac{\hat{x}_{i,j,k}\cdot\hat{x}_{i,j,\overline{k}}\cdot\gamma_{\overline{k}}\cdot\Delta_{i,j,\overline{k}}}{d\cdot\sigma_{i,j}}\right)_{\substack{i=1,\dots,n\\ j=1,\dots,m\\ k=1,\dots,d}} ,\end{aligned}} \tag{4.3.7}$$

where $\hat{x}$ are the normalized inputs and $\sigma_{i,j} \equiv \sigma(x_{i,j})$ is the standard deviation of each input vector $x_{i,j}$.

Proof. We must calculate the derivatives of the error with respect to the input, x:

$$\frac{\partial e}{\partial x} = \frac{\partial y}{\partial x}\frac{\partial e}{\partial y} = \frac{\partial \mathbf{A}}{\partial x}\cdot\Delta = \frac{\partial \mathcal{N}}{\partial x}\frac{\partial \mathbf{A}}{\partial \hat{x}}\cdot\Delta \quad . \tag{4.3.8}$$

We easily see that:

$$\frac{\partial \mathbf{A}_k}{\partial \hat{x}_{\overline{k}}}\Big(\hat{x};\gamma,\beta\Big) = \mathbf{1}\{\overline{k}=k\}\cdot\gamma_{\overline{k}} \quad .$$

After that, let us calculate $\frac{\partial \mathcal{N}}{\partial x}$.
Since $\mathcal{N}$ is applied to each entry, let us consider $x \in \mathbb{R}^d$:

$$\frac{\partial \mathcal{N}_k}{\partial x_{\overline{k}}}(x) = \frac{\frac{\partial (x-\mu)_k}{\partial x_{\overline{k}}}(x)\cdot(\sigma+\epsilon) - (x_k-\mu)\cdot\frac{\partial(\sigma+\epsilon)}{\partial x_{\overline{k}}}(x)}{(\sigma+\epsilon)^2} \quad . \tag{4.3.9}$$

Now we calculate $\frac{\partial (x-\mu)_k}{\partial x_{\overline{k}}}$ and $\frac{\partial(\sigma+\epsilon)}{\partial x_{\overline{k}}}$ (calculations are provided with more detail in [7]):

$$\frac{\partial (x-\mu)_k}{\partial x_{\overline{k}}} = \frac{\partial x_k}{\partial x_{\overline{k}}} - \frac{\partial \mu}{\partial x_{\overline{k}}} = \mathbf{1}\{\overline{k}=k\} - \frac{1}{d} \quad , \tag{4.3.10}$$

$$\frac{\partial (\sigma+\epsilon)}{\partial x_{\overline{k}}} = \frac{\partial \sigma}{\partial x_{\overline{k}}} + 0 = \frac{x_{\overline{k}}-\mu}{d \cdot \sigma} \quad . \tag{4.3.11}$$

Substituting (4.3.10) and (4.3.11) in (4.3.9) and expanding, we get:

$$\begin{aligned}\frac{\partial \mathcal{N}_k}{\partial x_{\overline{k}}}(x) &= \frac{\mathbf{1}\{\overline{k}=k\}}{\sigma+\epsilon} - \frac{1}{d\cdot(\sigma+\epsilon)} - \frac{(x_k-\mu)\cdot(x_{\overline{k}}-\mu)}{d\cdot\sigma\cdot(\sigma+\epsilon)^2} \\ &= \frac{\mathbf{1}\{\overline{k}=k\}}{\sigma+\epsilon} - \frac{1}{d\cdot(\sigma+\epsilon)} - \frac{\hat{x}_k\cdot\hat{x}_{\overline{k}}}{d\cdot\sigma} \quad .\end{aligned}$$

Then, substituting the expression of each part in (4.3.8),

$$\begin{aligned}&\frac{\partial e}{\partial x}\Big(y(x;\gamma,\beta)\Big) = \\ &= \frac{\partial \mathcal{N}}{\partial x} \cdot \left(\sum_{\overline{k}=1}^{d} \mathbf{1}\{\overline{k}=k\}\cdot\gamma_{\overline{k}}\cdot\Delta_{i,j,\overline{k}}\right)_{\substack{i=1,\dots,n\\ j=1,\dots,m\\ k=1,\dots,d}} \\ &= \frac{\partial \mathcal{N}}{\partial x} \cdot \Big(\gamma_k\cdot\Delta_{i,j,k}\Big)_{\substack{i=1,\dots,n\\ j=1,\dots,m\\ k=1,\dots,d}} \\ &= \left(\sum_{\overline{k}=1}^{d}\left(\frac{\mathbf{1}\{\overline{k}=k\}}{\sigma_{i,j}+\epsilon} - \frac{1}{d\cdot(\sigma_{i,j}+\epsilon)} - \frac{\hat{x}_{i,j,k}\cdot\hat{x}_{i,j,\overline{k}}}{d\cdot\sigma_{i,j}}\right)\cdot\gamma_{\overline{k}}\cdot\Delta_{i,j,\overline{k}}\right)_{\substack{i=1,\dots,n\\ j=1,\dots,m\\ k=1,\dots,d}} \\ &= \left(\frac{\gamma_k\cdot\Delta_{i,j,k}}{\sigma_{i,j}+\epsilon} - \sum_{\overline{k}=1}^{d}\frac{\gamma_{\overline{k}}\cdot\Delta_{i,j,\overline{k}}}{d\cdot(\sigma_{i,j}+\epsilon)} - \sum_{\overline{k}=1}^{d}\frac{\hat{x}_{i,j,k}\cdot\hat{x}_{i,j,\overline{k}}\cdot\gamma_{\overline{k}}\cdot\Delta_{i,j,\overline{k}}}{d\cdot\sigma_{i,j}}\right)_{\substack{i=1,\dots,n\\ j=1,\dots,m\\ k=1,\dots,d}} \quad ,\end{aligned}$$

which is the expression in the statement. □

III. Addition layer

Observation 4.22. The addition layer does not have any parameters, so there are no parameters derivatives to calculate.

Proposition 4.23 (Addition Inputs Derivatives). *Given the addition layer function $y(x_1, x_2)$ (3.10), its inputs derivatives are:*

(1) Input 1:

$$\boxed{\frac{\partial e}{\partial x_1}\Big(y(x_1,x_2)\Big) = \Delta} \quad , \tag{4.3.12}$$

(2) Input 2:

$$\boxed{\frac{\partial e}{\partial x_2}\Big(y(x_1,x_2)\Big)=\Delta} \quad . \tag{4.3.13}$$

Proof. We must calculate the derivatives of the error with respect to:

(1) Input x_1:

$$\frac{\partial e}{\partial x_1}=\frac{\partial y}{\partial x_1}\frac{\partial e}{\partial y}=\frac{\partial y}{\partial x_1}\cdot\Delta \quad ,$$

(2) Input x_2:

$$\frac{\partial e}{\partial x_2}=\frac{\partial y}{\partial x_2}\frac{\partial e}{\partial y}=\frac{\partial y}{\partial x_2}\cdot\Delta \quad .$$

It is easy to see that for both x_1 and x_2, $\frac{\partial y}{\partial x}$ is:

$$\frac{\partial y}{\partial x_{\bar{i},\bar{j},\bar{k}}}=\mathbf{1}\{\bar{i}=i\}\cdot\mathbf{1}\{\bar{j}=j\}\cdot\mathbf{1}\{\bar{k}=k\} \quad ,$$

which, when multiplied with Δ results in (4.3.12) and (4.3.13). □

II. Multi-head attention layer

Proposition 4.24 (Multi-Head Attention Parameters Derivatives)**.** *Given the multi-head attention layer function $y(x_I,x_C;W_Q,b_Q,W_K,b_K,W_V,b_V,W_P,b_P)$ (3.9), its parameters derivatives are:*

(1) Query weights:

$$\boxed{\frac{\partial e}{\partial W_Q}\Big(x_I;W_Q,b_Q\Big)=\left(\sum_{i=1}^{n}\sum_{j_I=1}^{m_I}x_{I\,i,j_I,k}\cdot\frac{\partial e}{\partial Q}^{(r)}_{i,j_I,\tilde{k}}\right)_{\substack{k=1,\dots,d\\ \tilde{k}=1,\dots,\tilde{d}\\ r=1,\dots,h}}} \quad , \tag{4.3.14}$$

(2) Query biases:

$$\boxed{\frac{\partial e}{\partial b_Q}\Big(x_I;W_Q,b_Q\Big)=\left(\sum_{i=1}^{n}\sum_{j_I=1}^{m_I}\frac{\partial e}{\partial Q}^{(r)}_{i,j_I,\tilde{k}}\right)_{\substack{\tilde{k}=1,\dots,\tilde{d}\\ r=1,\dots,h}}} \quad , \tag{4.3.15}$$

(3) Key weights:

$$\boxed{\frac{\partial e}{\partial W_K}\Big(x_C;W_K,b_K\Big)=\left(\sum_{i=1}^{n}\sum_{j_C=1}^{m_C}x_{C\,i,j_C,k}\cdot\frac{\partial e}{\partial K}^{(r)}_{i,j_C,\tilde{k}}\right)_{\substack{k=1,\dots,d\\ \tilde{k}=1,\dots,\tilde{d}\\ r=1,\dots,h}}} \quad , \tag{4.3.16}$$

(4) Key biases:

$$\boxed{\frac{\partial e}{\partial b_K}\Big(x_C; W_K, b_K\Big) = \left(\sum_{i=1}^{n}\sum_{j_C=1}^{m_C} \frac{\partial e}{\partial K}^{(r)}_{i,j_C,\tilde{k}}\right)_{\substack{\tilde{k}=1,\dots,\tilde{d}\\ r=1,\dots,h}}} \quad , \tag{4.3.17}$$

(5) Value weights:

$$\boxed{\frac{\partial e}{\partial W_V}(x_C; W_V, b_V) = \left(\sum_{i=1}^{n}\sum_{j_C=1}^{m_C} x_{C\,i,j_C,k} \cdot \frac{\partial e}{\partial V}^{(r)}_{i,j_C,\tilde{k}}\right)_{\substack{k=1,\dots,d\\ \tilde{k}=1,\dots,\tilde{k}\\ r=1,\dots,h}}} \quad , \tag{4.3.18}$$

(6) Value biases:

$$\boxed{\frac{\partial e}{\partial b_V}(x_C; W_V, b_V) = \left(\sum_{i=1}^{n}\sum_{j_C=1}^{m_C} \frac{\partial e}{\partial V}^{(r)}_{i,j_C,\tilde{k}}\right)_{\substack{\tilde{k}=1,\dots,\tilde{k}\\ r=1,\dots,h}}} \quad , \tag{4.3.19}$$

(7) Projection weights:

$$\boxed{\frac{\partial e}{\partial W_P}(O; W_P, b_P) = \left(\sum_{i=1}^{n}\sum_{j_I=1}^{m_I} O^{(r)}_{i,j_I,\tilde{k}} \cdot \Delta_{i,j_I,k}\right)_{\substack{\tilde{k}=1,\dots,\tilde{d}\\ k=1,\dots,k\\ r=1,\dots,h}}} \quad , \tag{4.3.20}$$

(8) Projection biases:

$$\boxed{\frac{\partial e}{\partial b_P}(O; W_P, b_P) = \left(\sum_{i=1}^{n}\sum_{j_I=1}^{m_I} \Delta_{i,j_I,k}\right)_{k=1,\dots,k}} \quad , \tag{4.3.21}$$

where Q, K, V, O are the query, key, value and attention outputs, respectively.

Proof. We must calculate the derivatives of the error with respect to:

(1) Query weights:

$$\frac{\partial e}{\partial W_Q} = \frac{\partial y}{\partial W_Q}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{P}}{\partial W_Q}\cdot\Delta = \frac{\partial \mathcal{M}}{\partial W_Q}\frac{\partial \mathcal{P}}{\partial O}\cdot\Delta = \frac{\partial \mathcal{T}}{\partial W_Q}\frac{\partial \mathcal{M}}{\partial Q}\frac{\partial \mathcal{P}}{\partial O}\cdot\Delta \quad ,$$

(2) Query biases:

$$\frac{\partial e}{\partial b_Q} = \frac{\partial y}{\partial b_Q}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{P}}{\partial b_Q}\cdot\Delta = \frac{\partial \mathcal{M}}{\partial b_Q}\frac{\partial \mathcal{P}}{\partial O}\cdot\Delta = \frac{\partial \mathcal{T}}{\partial b_Q}\frac{\partial \mathcal{M}}{\partial Q}\frac{\partial \mathcal{P}}{\partial O}\cdot\Delta \quad ,$$

(3) Key weights:

$$\frac{\partial e}{\partial W_K} = \frac{\partial y}{\partial W_K}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{P}}{\partial W_K} \cdot \Delta = \frac{\partial \mathcal{M}}{\partial W_K}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta = \frac{\partial \mathcal{T}}{\partial W_K}\frac{\partial \mathcal{M}}{\partial K}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta \quad ,$$

(4) Key biases:

$$\frac{\partial e}{\partial b_K} = \frac{\partial y}{\partial b_K}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{P}}{\partial b_K} \cdot \Delta = \frac{\partial \mathcal{M}}{\partial b_K}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta = \frac{\partial \mathcal{T}}{\partial b_K}\frac{\partial \mathcal{M}}{\partial K}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta \quad ,$$

(5) Value weights:

$$\frac{\partial e}{\partial W_V} = \frac{\partial y}{\partial W_V}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{P}}{\partial W_V} \cdot \Delta = \frac{\partial \mathcal{M}}{\partial W_V}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta = \frac{\partial \mathcal{T}}{\partial W_V}\frac{\partial \mathcal{M}}{\partial V}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta \quad ,$$

(6) Value biases:

$$\frac{\partial e}{\partial b_V} = \frac{\partial y}{\partial b_V}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{P}}{\partial b_V} \cdot \Delta = \frac{\partial \mathcal{M}}{\partial b_V}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta = \frac{\partial \mathcal{T}}{\partial b_V}\frac{\partial \mathcal{M}}{\partial V}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta \quad ,$$

(7) Projection weights:

$$\frac{\partial e}{\partial W_P} = \frac{\partial y}{\partial W_P}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{P}}{\partial W_P} \cdot \Delta \quad ,$$

(8) Projection biases:

$$\frac{\partial e}{\partial b_P} = \frac{\partial y}{\partial b_P}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{P}}{\partial b_P} \cdot \Delta \quad .$$

We will start with the outermost parameters of the layer function, the projection weights W_P and biases b_P:

$$\frac{\partial \mathcal{P}_{i,j_I,k}}{\partial {W_P}^{(\overline{r})}_{\bar{\overline{k}},\overline{k}}} = O^{(\overline{r})}_{i,j_I,\bar{\overline{k}}} \cdot \mathbf{1}\{\overline{k} = k\} \quad ,$$

$$\frac{\partial \mathcal{P}_{i,j_I,k}}{\partial {b_P}_{\overline{k}}} = \mathbf{1}\{\overline{k} = k\} \quad .$$

Multiplying with Δ, we get expressions (4.3.20) and (4.3.21) from the statement.

Next, we calculate $\frac{\partial e}{\partial W_V}$ and $\frac{\partial e}{\partial b_V}$. We must start with $\frac{\partial \mathcal{P}}{\partial O}$:

$$\frac{\partial \mathcal{P}_{i,j_I,k}}{\partial O^{(\overline{r})}_{\overline{i},\overline{j_I},\bar{\overline{k}}}} = \mathbf{1}\{\overline{i} = i\} \cdot \mathbf{1}\{\overline{j_I} = j_I\} \cdot {W_P}^{(\overline{r})}_{\bar{\overline{k}},k} \quad .$$

Then, the attention outputs derivatives are:

$$\frac{\partial e}{\partial O} = \left(\sum_{k=1}^{d} \Delta_{i,j_I,k} \cdot W_{P\,\tilde{k},k}^{(r)} \right)_{\substack{i=1,\dots,n \\ j_I=1,\dots,m_I \\ \tilde{k}=1,\dots,\tilde{k} \\ r=1,\dots,h}} \quad .$$

Now let us continue with $\frac{\partial \mathcal{M}}{\partial V}$:

$$\frac{\partial \mathcal{M}^{(r)}_{i,j_I,\tilde{k}}}{\partial V^{(\overline{r})}_{\overline{i},\overline{j_C},\overline{\tilde{k}}}} = \hat{A}^{(\overline{r})}_{\overline{i},\overline{j_C},j_I} \cdot \mathbf{1}\{\overline{i} = i\} \cdot \mathbf{1}\{\overline{\tilde{k}} = \tilde{k}\} \cdot \mathbf{1}\{\overline{r} = r\} \quad ,$$

where $\hat{A}$ are the attention weights. Then, the value derivatives are:

$$\frac{\partial e}{\partial V} = \left(\sum_{j_I=1}^{m_I} \hat{A}^{(r)}_{i,j_C,\tilde{k}} \cdot \frac{\partial e}{\partial O}^{(r)}_{i,j_I,\tilde{k}} \right)_{\substack{i=1,\dots,n \\ j_C=1,\dots,m_C \\ \tilde{k}=1,\dots,\tilde{d} \\ r=1,\dots,h}} \quad . \tag{4.3.22}$$

Now let us calculate $\frac{\partial \mathcal{T}}{\partial W}$ and $\frac{\partial \mathcal{T}}{\partial b}$, where W and b can be the query, key or value weights and biases, respectively. Let x be either x_I or x_C:

$$\frac{\partial \mathcal{T}^{(r)}_{i,j,\tilde{k}}}{\partial W^{(\overline{r})}_{\overline{k},\overline{\tilde{k}}}} = x_{i,j,\overline{k}} \cdot \mathbf{1}\{\overline{\tilde{k}} = \tilde{k}\} \cdot \mathbf{1}\{\overline{r} = r\} \quad , \tag{4.3.23}$$

$$\frac{\partial \mathcal{T}^{(r)}_{i,j,\tilde{k}}}{\partial b^{(\overline{r})}_{\overline{\tilde{k}}}} = \mathbf{1}\{\overline{\tilde{k}} = \tilde{k}\} \cdot \mathbf{1}\{\overline{r} = r\} \quad . \tag{4.3.24}$$

Multiplying with the value derivatives, we get expressions (4.3.18) and (4.3.19) from the statement.
To calculate the remaining derivatives, we first must find $\frac{\partial e}{\partial Q}$ and $\frac{\partial e}{\partial K}$. We will do it by following the steps 1-3 of $\mathcal{M}$ as described in Section II:

$$\frac{\partial e}{\partial Q} = \frac{\partial \mathcal{M}}{\partial Q}\frac{\partial e}{\partial O} = \frac{\partial A}{\partial Q}\frac{\partial \hat{A}}{\partial A}\frac{\partial O}{\partial \hat{A}}\frac{\partial e}{\partial O} \quad ,$$

$$\frac{\partial e}{\partial K} = \frac{\partial \mathcal{M}}{\partial K}\frac{\partial e}{\partial K} = \frac{\partial A}{\partial K}\frac{\partial \hat{A}}{\partial A}\frac{\partial O}{\partial \hat{A}}\frac{\partial e}{\partial O} \quad ,$$

where A are the attention scores.
Let us follow the previously mentioned steps:

3. Attention weights derivatives:

$$\frac{\partial O^{(r)}_{i,j_I,\tilde{k}}}{\partial \hat{A}^{(\overline{r})}_{\overline{i},\overline{j_C},\overline{j_I}}} = V^{(\overline{r})}_{\overline{i},\overline{j_C},\tilde{k}} \cdot \mathbf{1}\{\overline{i}=i\}\mathbf{1}\{\overline{j_I}=j_I\}\mathbf{1}\{\overline{r}=r\}$$

$$\Rightarrow \frac{\partial e}{\partial \hat{A}} = \frac{\partial O}{\partial \hat{A}}\frac{\partial e}{\partial O} = \left(\sum_{\tilde{k}=1}^{\tilde{d}} V^{(r)}_{i,j_C,\tilde{k}} \cdot \frac{\partial e}{\partial O}^{(r)}_{i,j_I,\tilde{k}}\right)_{\substack{i=1,\dots,n\\ j_C=1,\dots,m_C\\ j_I=1,\dots,m_I\\ r=1,\dots,h}} .$$

2. Attention scores derivatives:
We already know (1.2.2) that:

$$\frac{\partial \mathcal{S}_{j_C}}{\partial x_{\overline{j_C}}}\Big(x_1,\dots,x_{m_C}\Big) = S_{\overline{j_C}} \cdot \left(\mathbf{1}\{\overline{j_C}=j_C\} - S_{j_C}\right) ,$$

where $S \equiv \mathcal{S}(x_1,\dots,x_{m_C})$. Then, we deduce that:

$$\begin{aligned}\frac{\partial \hat{A}^{(r)}_{i,j_C,j_I}}{\partial A^{(\overline{r})}_{\overline{i},\overline{j_C},\overline{j_I}}} &= \frac{\partial \mathcal{S}_{j_C}}{\partial A^{(\overline{r})}_{\overline{i},\overline{j_C},\overline{j_I}}}\Big(A^{(r)}_{i,1,j_I},\dots,A^{(r)}_{i,m_C,j_I}\Big)\\ &= \mathbf{1}\{\overline{i}=i\}\cdot\mathbf{1}\{\overline{j_I}=j_I\}\cdot\mathbf{1}\{\overline{r}=r\}\cdot \hat{A}^{(\overline{r})}_{\overline{i},\overline{j_C},\overline{j_I}} \cdot \left(\mathbf{1}\{\overline{j_C}=j_C\} - \hat{A}^{(\overline{r})}_{\overline{i},j_C,\overline{j_I}}\right)\end{aligned}$$

$$\Rightarrow \frac{\partial e}{\partial A} = \frac{\partial \hat{A}}{\partial A}\frac{\partial e}{\partial \hat{A}} = \left(\hat{A}^{(r)}_{i,j_C,j_I} \cdot \left(\frac{\partial e}{\partial \hat{A}}^{(r)}_{i,j_C,j_I} - \sum_{\overline{j_C}=1}^{m_C} \hat{A}^{(r)}_{i,\overline{j_C},j_I} \cdot \frac{\partial e}{\partial \hat{A}}^{(r)}_{i,\overline{j_C},j_I}\right)\right)_{\substack{i=1,\dots,n\\ j_C=1,\dots,m_C\\ j_I=1,\dots,m_I\\ r=1,\dots,h}} .$$

Note that it does not affect the gradient's expression whether we add the causal mask between steps 1 and 2. It has an implicit effect on the values of the attention weights $\hat{A}$, but no additional terms have to be added to the derivatives.

1. Query and key derivatives:

$$\frac{\partial A^{(r)}_{i,j_C,j_I}}{\partial Q^{(\overline{r})}_{\overline{i},\overline{j_I},\overline{\tilde{k}}}} = \tilde{d}^{-1/2} \cdot \mathbf{1}\{\overline{i}=i\}\cdot\mathbf{1}\{\overline{j_I}=j_I\}\cdot\mathbf{1}\{\overline{r}=r\}\cdot K^{(\overline{r})}_{\overline{i},j_C,\overline{\tilde{k}}} ,$$

$$\frac{\partial A^{(r)}_{i,j_C,j_I}}{\partial K^{(\overline{r})}_{\overline{i},\overline{j_C},\overline{\tilde{k}}}} = \tilde{d}^{-1/2} \cdot \mathbf{1}\{\overline{i}=i\}\cdot\mathbf{1}\{\overline{j_C}=j_C\}\cdot\mathbf{1}\{\overline{r}=r\}\cdot Q^{(\overline{r})}_{\overline{i},j_I,\overline{\tilde{k}}} .$$

Then, finally:

$$\frac{\partial e}{\partial Q} = \frac{\partial A}{\partial Q}\frac{\partial e}{\partial A} = \left(\tilde{d}^{-1/2} \cdot \sum_{j_C=1}^{m_C} \frac{\partial e}{\partial A}^{(r)}_{i,j_C,j_I} \cdot K^{(r)}_{i,j_C,\tilde{k}}\right)_{\substack{i=1,\dots,n\\ j_I=1,\dots,m_I\\ \tilde{k}=1,\dots,\tilde{d}\\ r=1,\dots,h}} , \tag{4.3.25}$$

$$\frac{\partial e}{\partial K} = \frac{\partial A}{\partial K}\frac{\partial e}{\partial A} = \left(\tilde{d}^{-1/2} \cdot \sum_{j_I=1}^{m_I} \frac{\partial e}{\partial A}^{(r)}_{i,j_C,j_I} \cdot Q^{(r)}_{i,j_I,\tilde{k}}\right)_{\substack{i=1,\dots,n\\ j_C=1,\dots,m_C\\ \tilde{k}=1,\dots,\tilde{d}\\ r=1,\dots,h}} . \tag{4.3.26}$$

Now we have all the necessary components to calculate the remaining derivatives. Multiplying each of the two previous expressions with (4.3.23) we get (4.3.14) and (4.3.16), respectively; and multiplying each with (4.3.24) we get (4.3.15) and (4.3.17). □

Proposition 4.25 (Multi-Head Attention Inputs Derivatives). *Given the multi-head attention layer function $y(x_I, x_C; W_Q, b_Q, W_K, b_K, W_V, b_V, W_P, b_P)$ (3.9), its inputs derivatives are:*

(1) Input:

$$\boxed{\frac{\partial e}{\partial x_I} = \left(\sum_{\tilde{k}=1}^{\tilde{d}} \sum_{r=1}^{h} \frac{\partial e}{\partial Q}^{(r)}_{i,j_I,\tilde{k}} \cdot W_Q{}^{(r)}_{\tilde{k},k}\right)_{\substack{i=1,\dots,n \\ j_I=1,\dots,m_I \\ k=1,\dots,d}}} \quad , \tag{4.3.27}$$

(2) Context:

$$\boxed{\frac{\partial e}{\partial x_C} = \left(\sum_{\tilde{k}=1}^{\tilde{d}} \sum_{r=1}^{h} \frac{\partial e}{\partial K}^{(r)}_{i,j_C,\tilde{k}} \cdot W_K{}^{(r)}_{\tilde{k},k} + \frac{\partial e}{\partial V}^{(r)}_{i,j_C,\tilde{k}} \cdot W_V{}^{(r)}_{\tilde{k},k}\right)_{\substack{i=1,\dots,n \\ j_C=1,\dots,m_C \\ k=1,\dots,d}}} \quad , \tag{4.3.28}$$

where Q, K, V are the query, key and value, respectively.

Proof. We have to calculate the derivatives of the error with respect to:

(1) Input x_I:

$$\frac{\partial e}{\partial x_I} = \frac{\partial y}{\partial x_I}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{M}}{\partial x_I}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta = \frac{\partial \mathcal{T}}{\partial x_I}\frac{\partial \mathcal{M}}{\partial Q}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta \quad ,$$

(2) Context x_C:

$$\frac{\partial e}{\partial x_C} = \frac{\partial y}{\partial x_C}\frac{\partial e}{\partial y} = \frac{\partial \mathcal{M}}{\partial x_C}\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta = \left(\frac{\partial \mathcal{T}}{\partial x_C}\frac{\partial \mathcal{M}}{\partial K} + \frac{\partial \mathcal{T}}{\partial x_C}\frac{\partial \mathcal{M}}{\partial V}\right)\frac{\partial \mathcal{P}}{\partial O} \cdot \Delta \quad .$$

All the necessary terms are already calculated except $\frac{\partial \mathcal{T}}{\partial x}$, where x can be x_I or x_C:

$$\frac{\partial \mathcal{T}^{(r)}_{i,j,\tilde{k}}}{\partial x_{\bar{i},\bar{j},\bar{k}}} = \mathbf{1}\{\bar{i} = i\} \cdot \mathbf{1}\{\bar{j} = j\} \cdot W^{(\bar{r})}_{\bar{k},\tilde{k}} \quad .$$

Multiplying the previous expression with (4.3.25) we get (4.3.27), and multiplying with (4.3.26) and (4.3.22) (and adding them) we get (4.3.28). □

I. Embedding layer

Proposition 4.26 (Embedding Parameters Derivatives)**.** *Given the embedding layer function $y(x;W)$ (3.7), its parameters derivatives are:*

$$\boxed{\frac{\partial e}{\partial W}\big(y(x;W)\big) = \left(\sqrt{d}\cdot\sum_{i=1}^{n}\sum_{j=1}^{m}\mathbf{1}\{s = x_{i,j}\}\cdot\Delta_{i,j,k}\right)_{\substack{s=1,\dots,v\\k=1,\dots,d}}} \quad , \tag{4.3.29}$$

where W are the embedding weights and d is the embedding depth.

Proof. We have to calculate the derivatives of the error with respect to the embedding weights, W:

$$\frac{\partial e}{\partial W} = \frac{\partial y}{\partial W}\frac{\partial e}{\partial y} = \sqrt{d}\cdot\frac{\partial l}{\partial W}\cdot\Delta \quad .$$

We only need to calculate $\frac{\partial l}{\partial W}$:

$$\frac{\partial l(x;W)_k}{\partial W_{s,\bar{k}}} = \frac{\partial W_{x,k}}{\partial W_{s,\bar{k}}} = \mathbf{1}\{s = x\}\cdot\mathbf{1}\{\bar{k} = k\} \quad ,$$

which when multiplied with Δ and scaled by $\sqrt{d}$ is the expression in the statement. □

Observation 4.27*.* Since embedding layers are always input layers of the neural network (i.e. they never take input from a previous layer), there is no need to calculate their input derivatives.

5 | Optimization Algorithm

We have seen 3 of the 4 main components of the training process of a neural network: dataset, neural network and loss index. The only component left is the optimization algorithm, which establishes how the information obtained from the loss index is employed to train the neural network on its task.

Throughout this chapter, we define the concept of an optimization algorithm, give different examples and explain what "batches" are in neural network training. Then, we specify which optimization algorithm is most commonly used for the training of Transformer models and detail some of its properties.

5.1 Definition

We have defined the training process of a neural network as the optimization Problem 4.8. The optimization algorithm is the method through which we solve said problem. Formally,

Definition 5.1. An *optimization algorithm* is an algorithm that iteratively updates the parameters of the neural network, θ, aiming to find the solution of the optimization Problem 4.8 (find the optimal parameters θ^* that minimize the loss function L). That is, it sets a rule:

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

where:

- t indexes the iteration in the optimization process.
- θ_t represents the parameters of the neural network at iteration t.
- $\Delta\theta_t$ is the update to the parameters at iteration t, determined by the specific optimization algorithm being used.

The optimization algorithm must have a stopping condition, which most commonly is a maximum number of iterations or a goal value of the loss function.

Observation 5.2. For most optimization algorithms, the update rule at iteration t, $\Delta\theta_t$, often involves the gradient of the loss function with respect to the parameters of said iteration, $\nabla_{\theta_t} L$, which we saw how to calculate in Section 4.1. The reason is because the opposite of said gradient points in the direction of steepest decrease of the loss function within the parameter space. By following that direction, we seek a local minimum of the function. Some optimization algorithms include mechanisms that try to avoid local minima, in search of the absolute minimum of the function.

Examples. Some examples of optimization algorithms commonly used are:

1. **Gradient Descent**. One of the simplest algorithms. It simply follows the steepest decrease direction:

$$\Delta\theta_t = -\eta \cdot \nabla_{\theta_t} L$$

   where $\eta \in \mathbb{R}^+$ is a learning rate coefficient.

2. **Momentum-Based Methods**. These methods incorporate a function of past updates, providing inertia to escape local minima and smooth out updates:

$$\Delta\theta_t = \beta \cdot \Delta\theta_{t-1} - \eta \cdot \nabla_{\theta_t} L$$

   where $\beta \in \mathbb{R}$ is the momentum coefficient.

Batch training

In practice, a dataset can have millions of samples, each with hundreds of input and output features, which makes its handling during the training process a considerable computational challenge. For this reason, it is a widely spread practice to split the dataset into smaller pieces and process each of them separately. This process is called *batch training*.

Let us start by defining what a batch is:

Definition 5.3. Given a dataset D with n samples, a *batch* is defined as a subset of D with a fixed number of samples, $n_B \leq n$, denoted as $B \subseteq D$.

Then, the batch training process is as follows:

1. The dataset is shuffled and split into batches of the same size, n_B.

2. An iteration of the optimization algorithm is performed over each batch: instead of calculating the gradients of the loss function over the entire dataset, the gradients are computed for each batch and used to update the model parameters. Each iteration over a batch is called a *step*.

3. This process repeats for all the batches in the dataset. Each iteration over the entire set of batches is called an *epoch*.

Note that the dataset is reshuffled (and batched) at the start of every epoch of the optimization algorithm. In the case that n_B is not a divisor of n, the remainder samples can be either batched into a smaller batch or simply discarded.

Observation 5.4. A particular case of batch training is when $n_B = n$, in which case there is only one batch, that is the entire dataset. That is what we had seen before this section.

Remark. When performing batch training, n_B is the samples number employed by the neural network and its layers, instead of n.

Batch training offers two key advantages:

- **Computational Efficiency.** Calculating gradients over smaller batches reduces memory usage and speeds up computations, making it feasible to train large models with limited computational resources.
- **Improved Convergence.** Using batches introduces stochasticity in the gradient estimates, which can help in escaping local minima and improving the convergence properties of the optimization algorithm, as it was demonstrated in [8].

In the following section we assume to be using batch training during the described optimization algorithm.

5.2 Adaptative Moment Estimation

Transformer neural networks often produce noisy or sparse gradients due to the nature of the sequence transduction task. The use of labels to represent discrete data often results in some labels being significantly more common than others. Thus, the more relevant labels show greater impact on the gradient, while the rest have barely any relevance. The consequence is poor learning for the parameters corresponding to the less frequent labels, which in the end gives rise to poor results where only the most common labels are properly placed.

To solve that issue, it is common to employ the Adaptive Moment Estimation [9] (ADAM) optimization algorithm for the training process. By computing an adaptive learning rate for each parameter, the ADAM algorithm proves extremely useful in this type of problems with noisy gradients.

The ADAM optimization algorithm combines the advantages of two other popular methods: AdaGrad [10] and RMSProp [11]. It leverages both the first moment (mean) and second moment (uncentered variance) of the gradient to calculate a learning rate for each parameter. The formal algorithm is described as follows:

Algorithm 5.5 (Adaptive Moment Estimation)**.** We denote the estimates of the first and second moment of the gradient at step t by $m_t, v_t \in \Theta$, respectively, which are defined by:

$$m_t := \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla_{\theta_t} L \quad ,$$

$$v_t := \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla_{\theta_t} L)^2 \quad ,$$

where $\beta_1, \beta_2 \in (0,1)$ are the exponential decay rates for the first and second moment estimates, respectively. The most common values are $\beta_1 = 0.9$, $\beta_2 = 0.999$.

The first and second moment estimates are initialized to $\mathbf{0} \in \Theta$:

$$m_0 = \mathbf{0} \qquad , \qquad v_0 = \mathbf{0} \quad .$$

The initialization at $\mathbf{0}$ causes the moment estimates to be biased towards it, especially in the early steps. To prevent a slow start of the algorithm and to accurately reflect the true moments of the gradient, we include a bias correction term to each of the estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad ,$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \quad .$$

Then, the update rule of the ADAM algorithm is:

$$\boxed{\Delta\theta_t = -\eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}} \quad , \qquad (5.2.1)$$

where $\eta \in \mathbb{R}^+$ is a learning rate coefficient and $\epsilon \in \mathbb{R}$ is a small constant to prevent division by zero.

Observation 5.6. The term $\frac{1}{\sqrt{\hat{v}_t}+\epsilon}$ acts as an effective learning rate for each parameter. This term adapts based on the magnitude of the gradient and its variance:

- Parameters with large gradients have their updates scaled down due to the larger second moment estimate $\hat{v}_t$.
- Parameters with small gradients have relatively larger updates since the second moment estimate $\hat{v}_t$ is smaller.

This is how the ADAM algorithm solves the noisy gradient problem.

Remark. When applying the ADAM algorithm specifically to Transformers, it is common (see [1]) to include a variable learning rate that updates at each step. That is, at step t, the learning rate is:

$$\eta_t = \eta_0 \cdot \min(t^{-0.5} \ , \ t \cdot t_w^{-1.5}) \quad ,$$

where $\eta_0 \in \mathbb{R}^+$ is the initial learning rate and $t_w \in \mathbb{N}$ is a number of "warm-up steps".

Observation 5.7. The ADAM update rule (5.2.1) is implemented in C++ code in Apendix B.8.

Convergence

The convergence of the ADAM algorithm is a problematic and nuanced topic. In 2018, Reddit et al. [12] pointed out a divergence issue with ADAM, and ever since, there have been multiple studies proposing different convergence conditions [13] or variants of the algorithm with more general convergence [10]. However, ADAM has been proved to achieve great results in practice, and it is still more popular than most of its variants.

We will consider the framework of convergence proposed by Bock and Weiß [14], which proves (under some conditions) local convergence for non-convex loss functions, L, and global convergence for the strictly convex case.

We know that during the training process (where a dataset is necessary), the loss function only depends on the choice of parameters, $L(\theta)$ (4.1.2). An assumption we have to make so we can guarantee local convergence is that the loss as a function of the parameters is a twice continuously differentiable function, $L \in \mathcal{C}^2(\Theta, \mathbb{R})$ with $\Theta \subset \mathbb{R}^p$, p the number of parameters. We then denote the Hessian on a minimum θ^* by $\nabla^2_\theta L(\theta^*)$, and its eigenvalues by λ_u , $u = 1, \dots, p$.

Now, let us state the convergence theorem proposed by Bock and Weiß and give an indication of how it is proved. The full proof can be found in [14].

Theorem 5.8 (Local Convergence of ADAM [14])**.** *Let $\theta^* \in \Theta \subset \mathbb{R}^p$ be a minimum of $L \in \mathcal{C}^2(\Theta, \mathbb{R})$, with definite positive Hessian $\nabla^2_\theta L(\theta^*)$, and $\eta, \epsilon, \beta_1, \beta_2$ defined as in Algorithm 5.5 verifying:*

$$\frac{\eta \cdot \max_{u=1}^{p}(\lambda_u)}{\sqrt{\epsilon}}(1-\beta_1) < 2\beta_1 + 2 \quad , \tag{5.2.2}$$

with λ_u , $u = 1, \dots, p$, the eigenvalues of $\nabla^2_\theta L(\theta^*)$.

Then, Algorithm 5.5 converges locally with exponential rate of convergence.

Proof. We consider the algorithm from the standpoint of dynamical systems. We define $x = (m, v, \theta)$ as the state variable of our dynamic system and consider the ADAM algorithm as the iteration of a time-variant dynamical system: $x_{t+1} = (m_{t+1}, v_{t+1}, \theta_{t+1}) = T(t, x_t) = T(t, (m_t, v_t, \theta_t))$. This system can be split into an autonomous and a non-autonomous part: $T(t, x) = \tilde{T}(x) + \Phi(t, x)$.

The point $x^* = (0, 0, \theta^*)$ is proved to be a fixed point for $\tilde{T}(x)$. Then, the stability of this equilibrium is studied with the goal of asymptotic stability for local minima θ^* .

For that, the jacobian of $\tilde{T}$ on said point is calculated, $J_{\tilde{T}}(0, 0, \theta^*)$, and it is proved that, under the inequality (5.2.2), its spectral radius verifies: $\rho(J_{\tilde{T}}(0, 0, \theta^*)) < 1$.

Since $L \in \mathcal{C}^2(\Theta, \mathbb{R})$, the gradient $\nabla_\theta L(\theta)$ is locally Lipschitz in some neighbourhood of θ^*. Using that and $\nabla_\theta L(\theta^*) = \mathbf{0}$, it is proved that:

$$\|\Phi(t,x)\| \leq C\beta^t \cdot \|x - x^*\| \quad ,$$

for some $C > 0$, where $\beta = \max\{\beta_1, \beta_2, \beta_1^2\}$.

This estimate is sufficient to prove the exponential stability of a fixed point of $\tilde{T}$. Thus, using $\rho(J_{\tilde{T}}(0,0,\theta^*)) < 1$, the fixed point x^* is locally exponentially stable, which gives local exponential convergence of the non-autonomous system $T(t,x)$, i.e. the ADAM algorithm. □

Remark. This result, however, also has some issues:

- While the assumption that $L \in \mathcal{C}^2(\Theta, \mathbb{R})$ is known to be reasonable, the inequality (5.2.2) needs the eigenvalues of $\nabla_\theta^2 L(\theta^*)$, and therefore it needs θ^*. Thus, it is an a posteriori estimation, and it is not of use when setting the values of β_1, β_2. In replacement, the following a priori estimations (proposed in [12] and [9], respectively) are used:

$$\begin{aligned} \beta_1 &< \sqrt{\beta_2} \quad , \\ \beta_1^2 &< \sqrt{\beta_2} \quad . \end{aligned} \tag{5.2.3}$$

- Theorem 5.8 works under the assumption of complete batch mode. That means it is assumed that there is only one batch with batch size equal to the amount of training data, which we know is less computationally efficient (5.1).

Despite these issues, it is an elegant result that gives useful insight into the convergence of the ADAM method, which has proven to perform excellently in practical scenarios regardless of its theoretical troubles.

6 | Practical Application

Now that we have the mathematical framework of the neural network training process and know how Transformer models fit into it, let us test its practical application in a real-world example.

In this chapter, we acquire a sequence transduction dataset, build a Transformer model, set the cross-entropy error and perform the ADAM optimization algorithm. All of this process is implemented in the programming language C++, by developing the necessary modules to the open-source library OpenNN. The most relevant code snippets were included throughout Appendix B.

Then, we compare the results of different configurations of Transformers and demonstrate their usability with an example of their deployment after training.

6.1 Translation Transformer Model

The example that we use for our testing is an English-to-Spanish translator. This is a sequence transduction task where the source sequences are sentences in English and the reference sequences are their translations to Spanish.

We will now detail each component of the setup employed for the translation Transformer model.

Dataset

The dataset used for the model training is the *English-Spanish Translation Dataset* by the Kaggle user *Loonie* (from https://www.kaggle.com/datasets/lonnieqin/english-spanish-translation-dataset/data). It is a sequence transduction dataset (see Section 2.2) with 118964 sentence-pairs of varying length. Each pair contains an English sentence and its translation to Spanish.

Not all 118964 samples are used to train the model. It is common practice to split samples between training (60%), selection (20%), and testing (20%). The selection samples are used in parallel with the training process to check if the model is learning correctly. The testing samples are reserved to test the model's performance once the training process is finished.

The dataset is in CSV (Comma-Separated Values) format. Before its usage, it was adapted to TSV (Tabular-Separated Values) format to circumvent any issues with sentences containing commas and make it easier to load with OpenNN.

Note that the dataset is made of text sequences and must be transformed to label sequences before the Transformer model can process it. The procedure through which text is turned into label-representations is called *tokenization*. The employed tokenization algorithm was the WordPiece Algorithm, introduced in [15]. This algorithm calculates an optimal vocabulary of word-pieces over the entire text corpus. Then, the text sentences are split into the corresponding word-pieces and, through a correspondence of word-pieces to labels, the text is finally transformed into label sequences that can be processed by a neural network.

While loading the dataset, start and end labels are added to each sequence. When necessary, padding labels are also added after the end labels to reach the appropriate sequence length.

Neural Network

As we mentioned earlier, the neural network built for this example is a Transformer model, see Section 3.2. However, the specific configuration of the neural network architecture is determined by a few factors. These are often called *hyper-parameters*. The hyper-parameters of the Transformer are:

1. **Source sequence length (m_S).** The maximum length of source sequences.
2. **Reference sequence length (m_R).** The maximum length of reference sequences.
3. **Source vocabulary size (v_S).** The number of labels in the source vocabulary, X_{v_S}.
4. **Reference vocabulary size (v_R).** The number of labels in the reference vocabulary, X_{v_R}.
5. **Embedding depth (d).** The dimension of the space where labels are embedded in the embedding layers.
6. **Hidden perceptron depth (d_p).** The neurons number of the first perceptron layers of each layer compound (encoder and decoder).
7. **Number of attention heads (h).** The number of attention heads in multi-head attention layers.
8. **Number of layer compounds (τ).** The number of encoders and decoders in the architecture.

Note that the first 4 hyper-parameters are determined by the dataset, and the other 4 are arbitrarily set by the user. Figure 5 shows more specifically how the last 4 hyper-parameters affect each layer of the architecture.

For our example, the values of the 4 hyper-parameters determined by the dataset are:

$$m_S = 56 \quad , \quad m_R = 53 \quad ,$$
$$v_S = 4562 \quad , \quad v_R = 6134 \quad .$$

As for the other 4 hyper-parameters, we will test different combinations of values and compare their performance in Section 6.2.

Loss Index

For the loss index we employed the cross-entropy error function (4.2.2) and no regularization term.

Like it was mentioned in item VI of Section 3.2 and Remark 4.13, the softmax normalization at the end of the neural network is not explicitly computed. Instead, we use the log-softmax expression (4.2.3) of the cross-entropy error function.

When calculating the error, the positions holding padding labels must be ignored, since predictions made at positions after an end label are irrelevant. Therefore, said positions are masked out (multiplied by 0) so that they do not add to the error. To account for this masking, we divide the error by the number of unmasked positions instead of the total number of positions.

Apart from the cross-entropy error, we also keep track of an accuracy metric. To calculate the accuracy, we take the most likely label of the vocabulary according to the neural network's output at each unmasked position and see if it matches the target label. Then, we add the number of successful matches and divide by the number of unmasked positions. The result is the *accuracy rate*, denoted by α.

The output deltas are calculated as detailed in Section 4.2, and the gradient of the loss function with respect to the parameters of the Transformer neural network is calculated through the back-propagation Algorithm 4.10. The back-propagation across the layers of the Transformer is performed using the calculations of Section 4.3.

Optimization Algorithm

We performed the ADAM optimization Algorithm 5.5 to train the neural network.

We applied batch training with a batch size of $n_B = 64$. The selection samples are also batched and, at the end of each epoch of the optimization algorithm, the neural network function is evaluated on the selection batches. This way, we can measure the error and accuracy of the model on samples that it has not seen during training. When the training error decreases while the selection error increases we say the model is suffering *overfitting*.

The exponential decay rates were set at their default value of $\beta_1 = 0.9$, $\beta_2 = 0.999$, which satisfy (5.2.3). We also used a variable learning rate, as described in Remark 5.2, with an initial value of $\eta_0 = 0.001$. As the stopping condition for the optimization algorithm we set an accuracy goal of $\alpha = 0.90$.

6.2 Results

In this section, we will discuss the results obtained in the practical application of the translator Transformer model and compare the training and performance of different configurations.

We trained 3 model configurations: “Large”, “Medium” and “Small”, with the following hyper-parameters and total number of parameters:

	$\mathbf{d}$	$\mathbf{d_p}$	$\mathbf{h}$	τ	**Number of parameters** (p)
Small	64	128	4	1	1,166,966
Medium	128	256	4	2	2,822,902
Large	256	512	8	2	6,950,390

The 3 models were trained on the same hardware. The following table summarizes the time (in seconds) and number of training epochs it took each model to achieve the goal of 0.90 accuracy:

	Epochs	Training time (s)	Avg. time/epoch
Small	43	12,043	280.1
Medium	16	22,360	1,397.5
Large	12	69,652	5,804.3

We observe that the smallest model takes significantly more epochs to converge than the 2 larger ones, which could be considered unusual. However, this behaviour likely reflects the limited generalization capabilities of the Small model. As we mentioned in Remark 3.1, a model with a higher number of parameters can represent a wider variety of functions. Since language translation is a complex task, Small is too limited to perform it proficiently, unlike Medium and Large, and therefore converges worse than them (the same thing happens between Medium and Large, though at a much lesser scale). The theoretical convergence issues of the ADAM method might also play a role in this unusual scenario, in which case it may be beneficial to set a different training goal.

Figures of Appendix A.3 show in-depth graphs of the 3 training processes, including the evolution of the error and accuracy throughout them. The 3 error graphs show overfitting (the selection error stops decreasing while the training error converges to 0), which is not particularly worrisome in language processing (due to its nuanced nature) unless the selection accuracy starts decreasing. We see that happening in the Small accuracy graph, highlighting its limited generalization capabilities. Meanwhile, the Medium and Large selection accuracies plateau but do not decrease, showing that their overfitting is not excessive.

After training, we performed a testing analysis on our models, utilizing the testing samples to examine their performance. This is achieved by calculating each model's error and accuracy on unseen samples. The following table shows the testing results of our models:

	Error	Accuracy
Small	3.14	0.55
Medium	2.18	0.60
Large	1.77	0.66

We observe that the accuracies are considerably lower than the 0.90 goal we set as the training goal. This is to be expected since the models have never seen the testing samples, and they may have "memorized" specific patterns of the training data that do not apply now. That being said, there is clearly a trend which indicates that Transformers benefit from larger architecture configurations.

Due to limited time and resources, our study could not test an even larger architecture that perhaps might have demonstrated better performance than Large. This could be a line of future work as a further study of the characteristics of Transformers.

As a final point, let us showcase a sample translation of the Large model after its training:

***Example* 6.1.** A translation from English to Spanish by the Large Transformer model:

- **Input sentence:** *The plant is green.*
- **Predicted sentence:** *La planta está verde.*
- **Target sentence:** *La planta es verde.*

This example demonstrates the model's understanding of language. Even though the prediction is not entirely correct, it clearly captures the key elements of the input sentence while still being a coherent sentence in Spanish that, in some contexts, could be a correct translation.

Ultimately, we consider this experiment satisfactory since we successfully trained a model that shows competent (though improvable) performance at an extremely complex task such as language translation. We believe the attained results validate the usefulness of our proposed mathematical model, and we encourage future studies to build upon it.

7 | Conclusion

We have introduced a mathematical framework for neural network modelling. Our study builds upon the concepts introduced in [1, 9, 16, 17] (among many others) by introducing a general foundation that spans any kind of neural network and its training process.

To showcase how each of the components of this framework may be utilized, we have focused specifically on Transformer models. These were chosen as the main object of study due to their relevance in the current state of the art in the field of neural networks, which they have gained thanks to their scalability and effectiveness in handling complex tasks.

We have also demonstrated the effectiveness of the proposed framework in a practical application by training a series of Transformer models and testing their performance on a real-world problem. The results obtained through the training processes of these neural networks are satisfactory and validate the mathematical model by delivering good performance in a complex task, such as language translation.

Although the experimental results provide valuable insight into Transformer behaviour, they also showcase some limitations of both the model and the study itself. The convergence of the ADAM method is a notoriously problematic subject that may have a detrimental effect on our experiment, and the resource limitation of our research disables us from further testing the capabilities of Transformers to their optimal performance.

In conclusion, our research has made meaningful contributions to the understanding and application of Transformers and neural networks in general. We have paved the way for further advancements in this promising area by providing a solid mathematical foundation and demonstrating its practical utility, as well as proposing areas that would benefit from additional examination.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[2] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[3] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[4] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[5] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 2nd edition, 1999.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, NY, USA, 2006.

[7] Lior Sinai. Backpropagation through a layer norm, May 2022.

[8] Galen Wilson and Tony Martinez. Theoretical analysis of batch and on-line training for gradient descent. *Artificial Intelligence*, 86(1):353–386, 2003.

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

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[11] Geoffrey Hinton. Neural networks for machine learning. Coursera Lecture 6e, 2012. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[12] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2018.

[13] Yushun Zhang, Congliang Chen, Naichen Shi, Ruoyu Sun, and Zhi-Quan Luo. Adam can converge without any modification on update rules, 2023.

[14] Sebastian Bock and Martin Weiß. *A Proof of Local Convergence for the ADAM Optimizer.* IEEE, 2019.

[15] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152. IEEE, 2012.

[16] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Inc., 1995.

[17] Artificial Intelligence Techniques S.L. Internal report on advanced neural networks, 2024.

[18] Joram Soch, Maja, Pietro Monticone, Thomas J. Faulkenberry, Alex Kipnis, Kenneth Petrykowski, Carsten Allefeld, Heiner Atze, Adam Knapp, and Ciarán D. McInerney. *The Book of Statistical Proofs.* Zenodo, January 2024.

[19] Israel M. Gelfand, Sergei V. Fomin, and Richard A. Silverman. *Calculus of Variations.* Dover Books on Mathematics. Dover Publications, 2000.

[20] Andrey N. Kolmogorov and Sergei V. Fomin. *Elements of the Theory of Functions and Functional Analysis.* Number v. 1 in Dover books on mathematics. Dover, 1999.

[21] Anqi Mao, Mehryar Mohri, and Yutao Zhong. Cross-entropy loss functions: Theoretical analysis and applications, 2023.

Glossary

Lowercase letters

a	Activation function.
b	Biases.
$\tilde{d}$	Neurons number / Hidden depth.
d	Depth.
h	Attention heads number.
i	Sample index.
j	Sequence position index.
$\tilde{k}$	Hidden depth/Neuron index.
k	Depth index.
l	Number of layers.
m	Input/Sequence length.
n	Samples number.
p	Parameters number.
q	Layer index.
r	Attention head index.
s	Vocabulary index.
u	Parameter index
v	Vocabulary size.
$\hat{x}$	Transformed/Normalized inputs.
x	Input.
$\hat{y}$	Target.
y	Layer function.

y_θ	Neural network function.

Uppercase letters

$\hat{A}$	Attention weights.
A	Attention scores.
C	Combinations.
D	Dataset.
K	Key.
M	Causal mask.
O	Attention outputs.
PE	Positional encoding matrix.
Q	Query.
V	Value.
W	Weights.
X	Layer input space.
X_v	Vocabulary of size v.

Calligraphic letters

$\mathcal{A}$	Activations function.
$\mathcal{C}$	Combination function.
$\mathcal{M}$	Multi-head attention computation function.
$\mathcal{N}$	Normalization function.
$\mathcal{P}$	Linear projection function.
$\mathcal{S}$	Softmax function.
$\mathcal{T}$	Linear transformation function.
$\mathcal{V}$	Neural network function space.
$\mathcal{X}$	Input space.
$\mathcal{Y}$	Output space.

Blackboard bold letters

$\mathbb{N}$	Natural numbers.
$\mathbb{R}$	Real numbers.
$\mathcal{S}_{(d-1)}$	$(d-1)$-simplex.

$\mathbb{Z}$	Integer numbers.

Boldface letters

$\mathbf{A}$	Affine transformation function.
$\mathbf{p}$	Probability vector.

Greek letters

β	Shift parameters.
ϵ	Small numerical constant.
γ	Scale parameters.
μ	Mean function.
ϕ	Embedding lookup function.
σ	Standard deviation function.
Θ	Parameter space.
θ	Neural network parameters.

Specific notations

$\bullet^{(r)}$	$\bullet$ of the r-th attention head (Section II).
$\bullet_C$	Context $\bullet$ (Section II).
$\bullet_I$	Input $\bullet$ (Section II).
$\bullet_R$	Reference $\bullet$ (2.2).
$\bullet_S$	Source $\bullet$ (2.2).
$\overline{\bullet}$	Auxiliary $\bullet$ index. Mostly used in sums and partial derivatives.

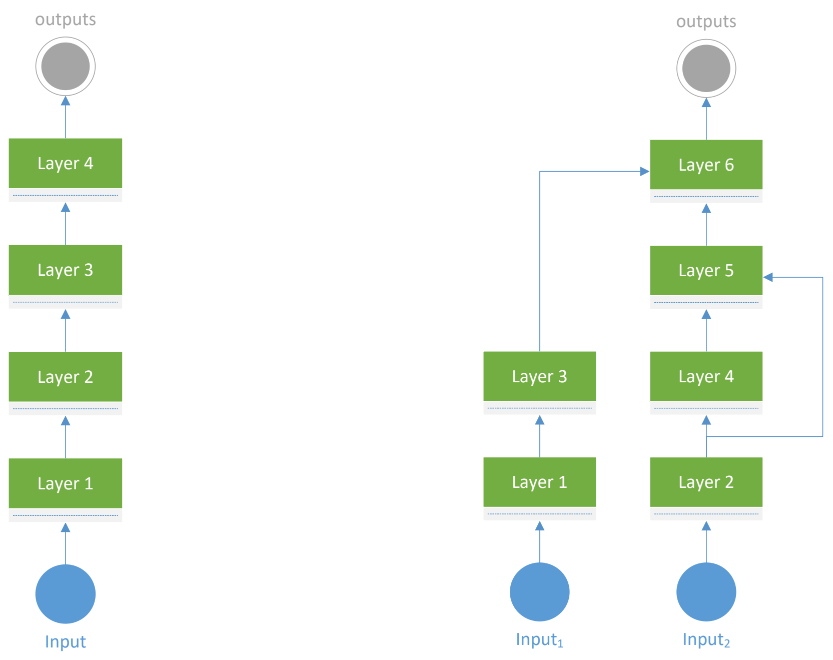
Apendix A: Figures

This appendix contains all of the figures mentioned previously.

A.1 Illustrative Examples

This section has figures that may help the reader understand better some key concepts relative to transformers, through illustrative examples.

Simple & complex neural network architectures



(a) A simple neural network architecture. It is only a concatenation of layers.

(b) A more complex neural network architecture. It has 2 inputs and its layers intertwine.

Figure 1: Two neural network architectures of varying complexities.

Embedding point cloud

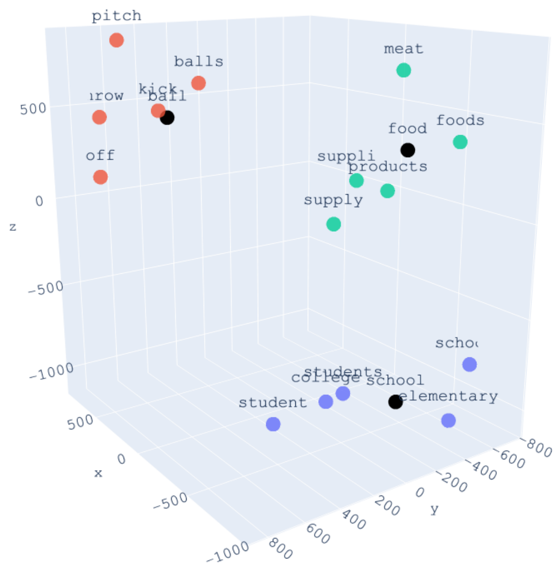


Figure 2: Embedding point cloud of a small set of words. There are clearly 3 clusters, corresponding to the 3 different topics in this reduced vocabulary (https://towardsdatascience.com/visualizing-word-embedding-with-pca-and-t-sne-961a692509f5).

Attention mechanism

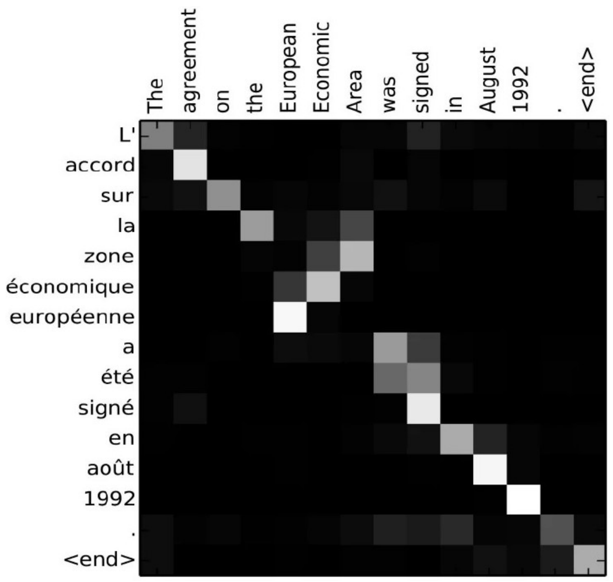
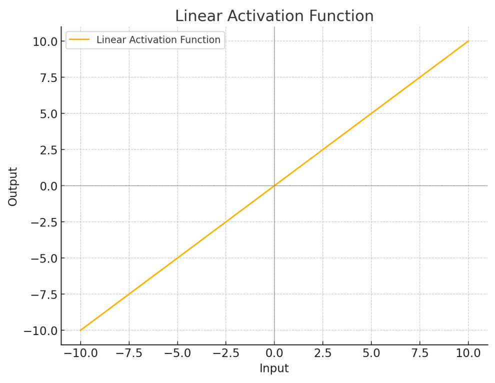


Figure 3: Attention mechanism where an English sentence is compared to its French translation. The relevance of each French word with respect to each English word is represented on a scale from black to white (https://www.scaler.com/topics/deep-learning/attention-mechanism-deep-learning/).

Perceptron activation functions



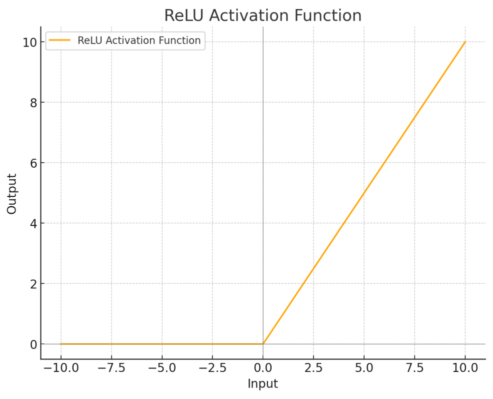


Figure 4: Linear and ReLU activation functions.

A.2 Transformer Architecture

This section has figures corresponding to the architecture of the transformer neural network and each of its layers.

Transformer neural network

(Next page)

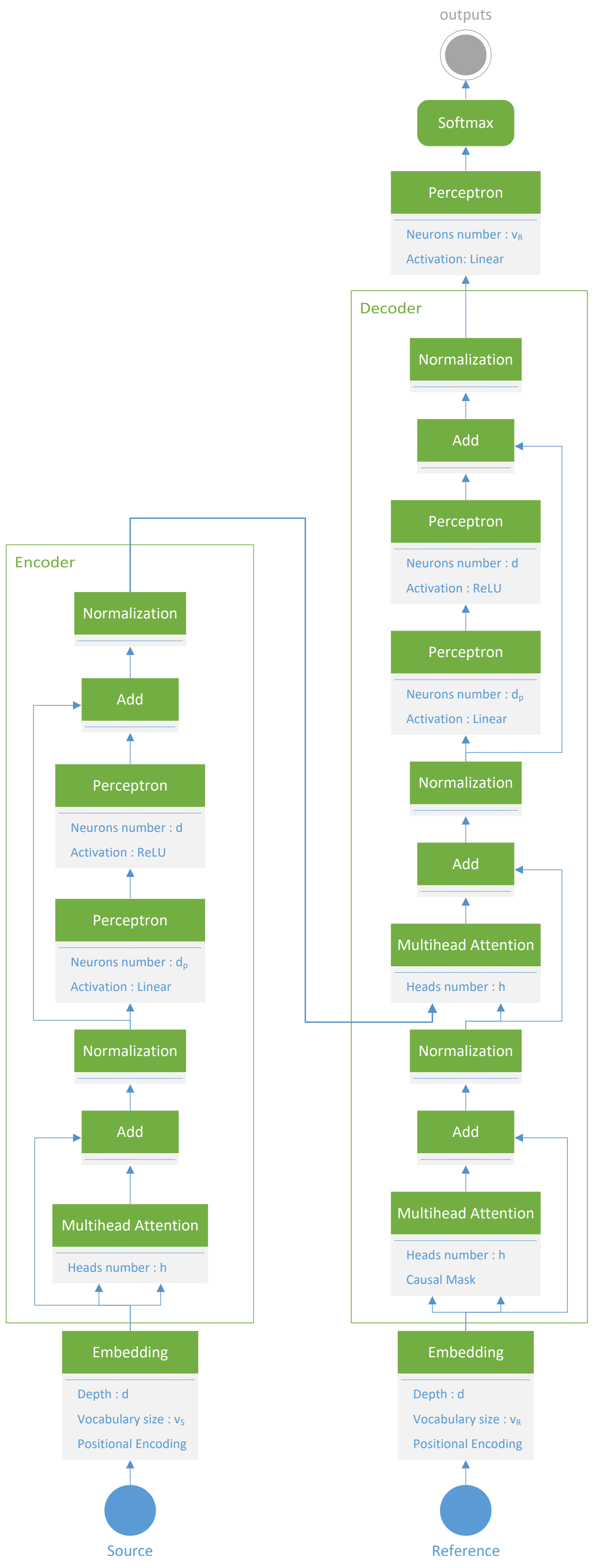


Figure 5: Transformer neural network architecture.

Embedding layer architecture

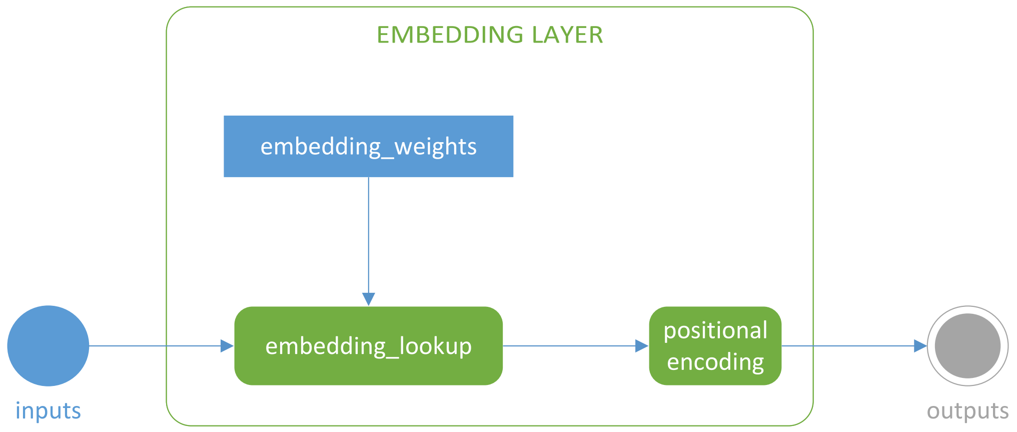


Figure 6: Embedding layer architecture.

Multi-head attention layer architecture.

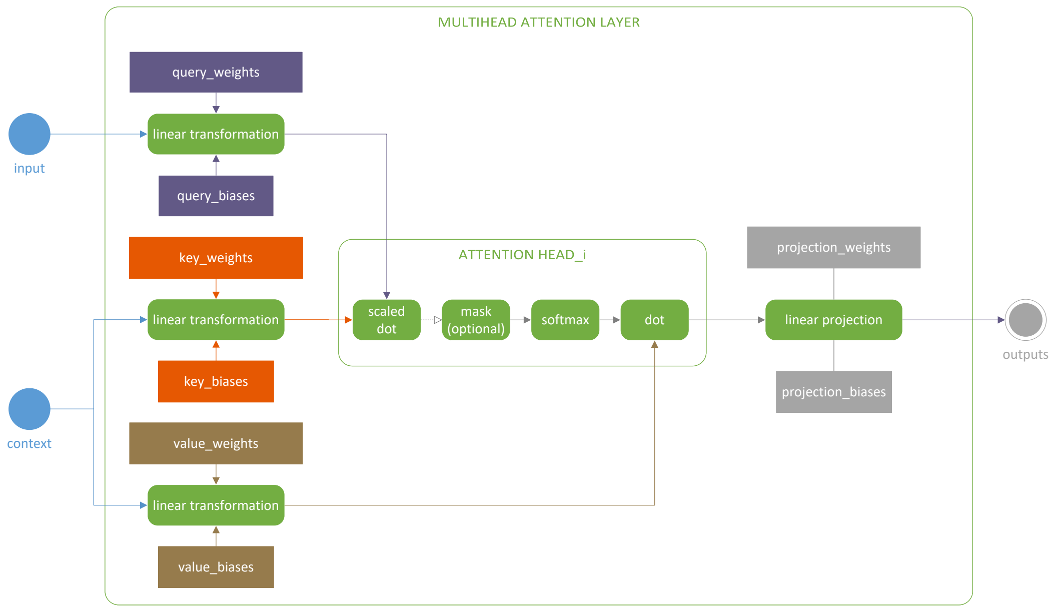


Figure 7: Multi-Head Attention layer architecture.

Addition layer architecture.

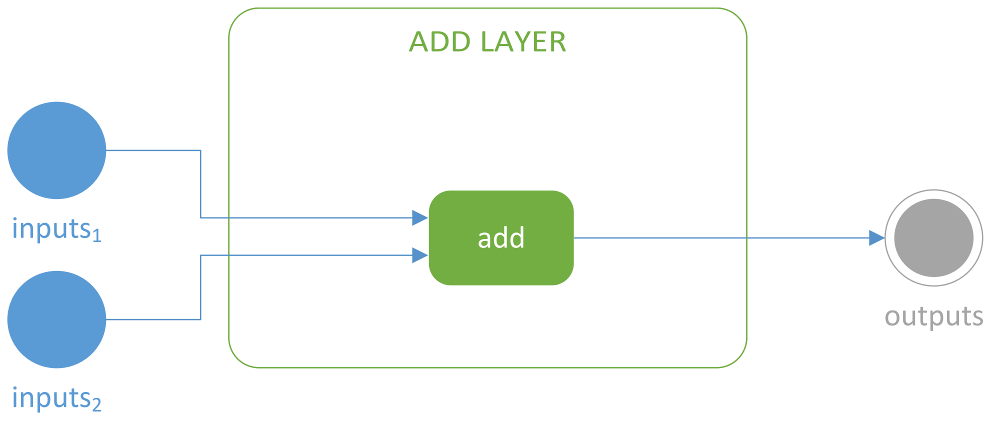


Figure 8: Addition layer

Normalization layer architecture.

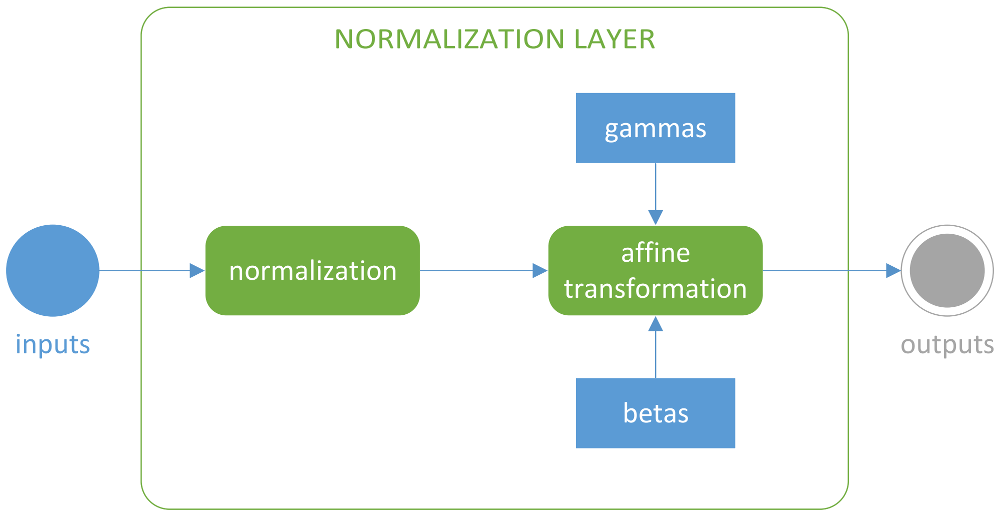


Figure 9: Normalization layer

Perceptron layer architecture.

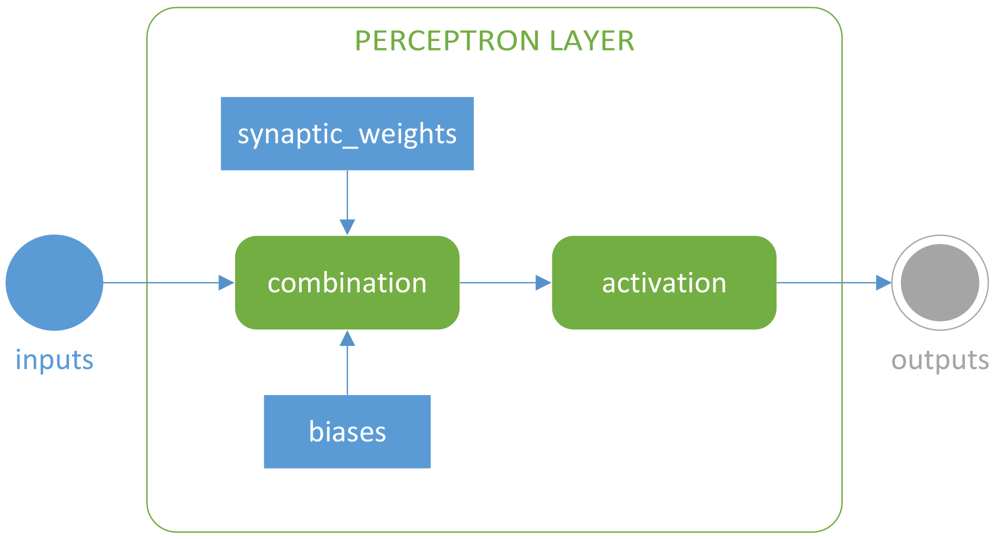


Figure 10: Perceptron layer

A.3 Practical Application

This section has figures corresponding to the training results of the practical application.

Small

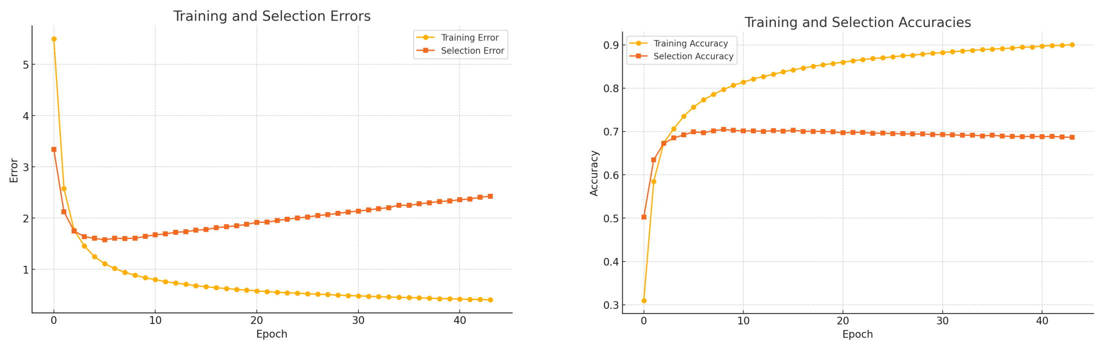


Figure 11: Small model training graphs.

Medium

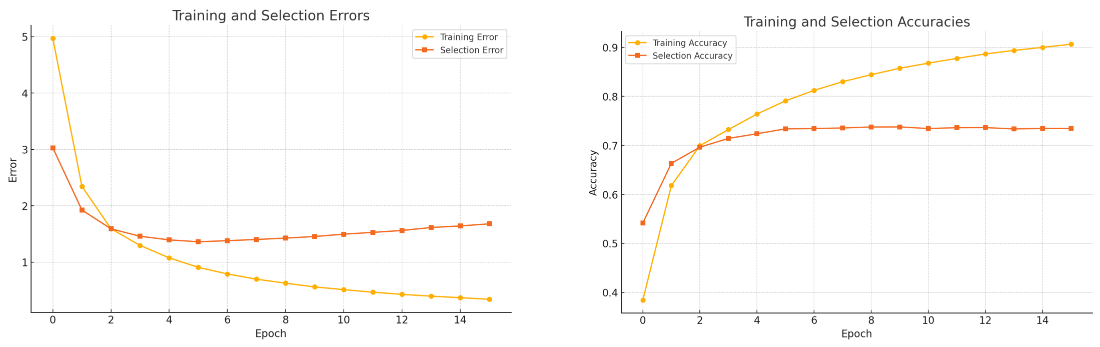


Figure 12: Medium model training graphs.

Large

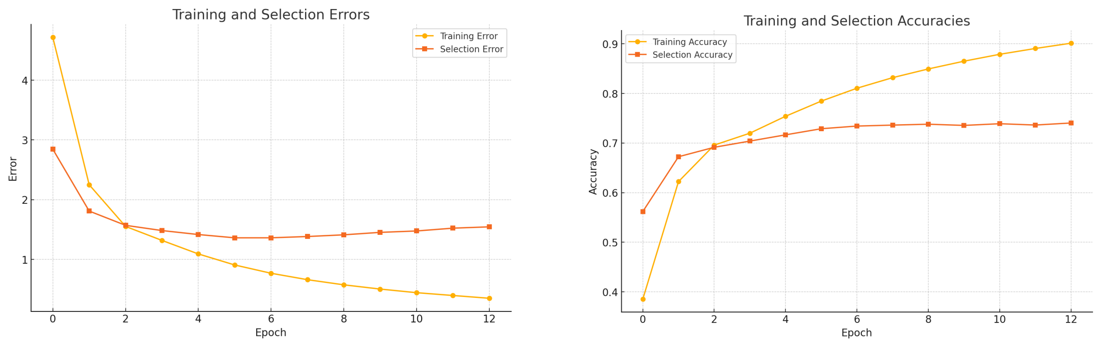


Figure 13: Large model training graphs.

Apendix B: Code

This appendix contains the code implementations of all the important formulas previously mentioned, organized by the sections in which they appear.

Only the code snippets corresponding to said formulas will be displayed. To find the entire project, visit the OpenNN GitHub repository.

Remark. Some code snippets might be slightly modified for simplicity.

B.4 Preliminaries

```cpp
void Layer::softmax(
    const Tensor<type, 3>& x,
    Tensor<type, 3>& y
    ) const
{
    const Eigen::array<Index, 1> softmax_dimension{ { 2 } };

    const Index rows_number = y.dimension(0);
    const Index columns_number = y.dimension(1);
    const Index channels_number = y.dimension(2);

    const Eigen::array<Index, 3> range_3{ { rows_number, columns_number, 1 } };
    const Eigen::array<Index, 3> expand_softmax_dim{ { 1, 1, channels_number } };

    y.device(*thread_pool_device) = x - x.maximum(softmax_dimension)
                                          .eval()
                                          .reshape(range_3)
                                          .broadcast(expand_softmax_dim);

    y.device(*thread_pool_device) = y.exp();

    y.device(*thread_pool_device) = y / y.sum(softmax_dimension)
                                          .eval()
                                          .reshape(range_3)
                                          .broadcast(expand_softmax_dim);

}
```

Listing 1: Softmax function.

B.5 Transformer Architecture

Embedding layer

```
void EmbeddingLayer::lookup_embedding(
    const Tensor<type, 2>& inputs,
    Tensor<type, 3>& outputs
    )
{
    const Index batch_size = inputs.dimension(0);

#pragma omp parallel for
    for(Index row = 0; row < batch_size; row++)
    {
        for(Index input_position = 0; input_position <
   inputs_number; input_position++)
        {
            outputs.chip(row, 0).chip(input_position, 0)
                = embedding_weights.chip(inputs(row,
   input_position), 0);
        }
    }
}
```

Listing 2: Embedding lookup function.

```
void EmbeddingLayerForwardPropagation::
   build_positional_encoding_matrix()
{
    const EmbeddingLayer* embedding_layer = static_cast<
   EmbeddingLayer*>(layer);

    const Index inputs_number = embedding_layer->
   get_inputs_number();
    const Index depth = embedding_layer->get_depth();

    positional_encoding.resize(inputs_number, depth);

    positional_encoding.setZero();

    const type half_depth = type(depth) / 2;

    #pragma omp parallel for
    for (Index i = 0; i < inputs_number; i++)
    {
        for (Index j = 0; j < Index(depth); j++)
        {
            if (j < Index(half_depth))
```

```
                positional_encoding(i, j) = sin((i) / pow(10000,
   (j) / half_depth));
            else
                positional_encoding(i, j) = cos((i) / pow(10000,
   (j - Index(half_depth)) / half_depth));
        }
    }

    built_positional_encoding_matrix = true;
}
```

Listing 3: Positional encoding matrix.

```
void EmbeddingLayer::forward_propagate(
    const Tensor<pair<type*, dimensions>, 1>& inputs_pair,
    LayerForwardPropagation* layer_forward_propagation,
    const bool& is_training
    )
{
    const TensorMap<Tensor<type, 2>> inputs(inputs_pair(0).first,
    inputs_pair(0).second[0], inputs_pair(0).second[1]);

    EmbeddingLayerForwardPropagation*
   embedding_layer_forward_propagation
        = static_cast<EmbeddingLayerForwardPropagation*>(
   layer_forward_propagation);

    Tensor<type, 3>& outputs =
   embedding_layer_forward_propagation->outputs;

    lookup_embedding(inputs, outputs);

    if(positional_encoding)
    {
        outputs.device(*thread_pool_device) = outputs * outputs.
   constant(sqrt(depth));

        const Tensor<type, 2>& positional_encoding =
   embedding_layer_forward_propagation->positional_encoding;

        for(Index batch_element = 0; batch_element < outputs.
   dimension(0); batch_element++)
        {
            outputs.chip(batch_element, 0).device(*
   thread_pool_device) += positional_encoding;
        }
    }
}
```

Listing 4: Embedding layer function.

Multi-head attention layer

```
void MultiheadAttentionLayer::calculate_linear_transformation(
    const Tensor<type, 3>& input,
    Tensor<type, 4>& transformed_input,
    const Tensor<type, 3>& weights,
    const Tensor<type, 2>& biases,
    Tensor<type, 2>& sample_matrix
    ) const
{
    const Index batch_size = input.dimension(0);

    type* weights_data = (type*)weights.data();
    type* biases_data = (type*)biases.data();
    type* transformed_input_data = transformed_input.data();

    for (Index head_index = 0; head_index < heads_number; head_index++)
    {
        type* head_weights_data = weights_data + head_index * depth * hidden_depth;
        type* head_biases_data = biases_data + head_index * hidden_depth;
        type* head_transformed_input_data = transformed_input_data + head_index * batch_size * sequence_size * hidden_depth;

        const TensorMap<Tensor<type, 2>> head_weights(head_weights_data, depth, hidden_depth);
        const TensorMap<Tensor<type, 1>> head_biases(head_biases_data, hidden_depth);

        for (Index sample_index = 0; sample_index < batch_size; sample_index++)
        {
            sample_matrix = input.chip(sample_index, 0);

            type* sample_transformed_input_data = head_transformed_input_data + sample_index * sequence_size * hidden_depth;

            TensorMap<Tensor<type, 2>> sample_transformed_input(sample_transformed_input_data, sequence_size, hidden_depth);

            sample_transformed_input.device(*thread_pool_device)
                = sample_matrix.contract(head_weights, A_B);

            sum_columns(thread_pool_device, head_biases, sample_transformed_input);
        }
    }
}
```

Listing 5: Linear transformation function.

```
void MultiheadAttentionLayer::build_causal_mask()
{
    constexpr type m_inf = -numeric_limits<type>::infinity();
```

```cpp

    causal_mask.resize(context_size, input_size);
    causal_mask.setZero();

    for (Index input_index = 0; input_index < input_size; input_index++)
    {
        for (Index context_index = input_index + 1; context_index < context_size; context_index++)
        {
            causal_mask(context_index, input_index) = m_inf;
        }
    }
}
```

Listing 6: Causal mask.

```cpp
void MultiheadAttentionLayer::compute_attention(
    const Tensor<type, 4>& query,
    const Tensor<type, 4>& key,
    const Tensor<type, 4>& value,
    Tensor<type, 4>& attention_scores,
    Tensor<type, 4>& attention_weights,
    Tensor<type, 4>& attention_outputs
    ) const
{
    batch_matrix_multiplication(thread_pool_device, key, query, attention_scores, A_BT);

    attention_scores.device(*thread_pool_device) = attention_scores * scaling_factor;

    if (use_causal_mask)
    {
        #pragma omp parallel for
        for (Index head_index = 0; head_index < heads_number; head_index++)
        {
            for (Index sample_index = 0; sample_index < batch_samples_number; sample_index++)
            {
                type* sample_attention_scores_data = attention_scores.data()
                    + sample_index * context_size * input_size
                    + head_index * context_size * input_size * batch_samples_number;

                TensorMap<Tensor<type, 2>> sample_attention_scores(sample_attention_scores_data, context_size, input_size);

                sample_attention_scores.device(*thread_pool_device) += causal_mask;
            }
        }
    }

    softmax(attention_scores, attention_weights);

    batch_matrix_multiplication(thread_pool_device,
   attention_weights, value, attention_outputs, AT_B);
}
```

Listing 7: Multi-head attention computation.

```
void MultiheadAttentionLayer::calculate_output_projection(
    const Tensor<type, 4>& attention_outputs,
    Tensor<type, 4>& projection_outputs,
    Tensor<type, 3>& outputs
    ) const
{
    const Index batch_size = outputs.dimension(0);

    type* attention_outputs_data = (type*)attention_outputs.data
   ();
    type* projection_outputs_data = projection_outputs.data();
    type* projection_weights_data = (type*)projection_weights.
   data();

    for(Index head_index = 0; head_index < heads_number;
   head_index++)
    {
        type* head_projection_output_data =
   projection_outputs_data + head_index * batch_size * input_size
    * depth;
        type* head_projection_weights_data =
   projection_weights_data + head_index * hidden_depth * depth;
        type* head_attention_output_data = attention_outputs_data
    + head_index * input_size * hidden_depth * batch_size;

        TensorMap<Tensor<type, 3>> head_projection_output(
   head_projection_output_data, batch_size, input_size, depth);

        const TensorMap<Tensor<type, 2>> head_projection_weights(
   head_projection_weights_data, hidden_depth, depth);

        for(Index sample_index = 0; sample_index < batch_size;
   sample_index++)
        {
            type* sample_attention_output_data =
   head_attention_output_data + sample_index * input_size *
   hidden_depth;

            const TensorMap<Tensor<type, 2>>
   sample_attention_output(sample_attention_output_data,
   input_size, hidden_depth);

            head_projection_output.chip(sample_index, 0).device(*
   thread_pool_device)
                = sample_attention_output.contract(
head_projection_weights, A_B);
        }
    }

    outputs.device(*thread_pool_device) = projection_outputs.sum(projection_sum_index);

    sum_matrices(thread_pool_device, projection_biases, outputs);
}
```

Listing 8: Linear projection function.

```
void MultiheadAttentionLayer::forward_propagate(
    const Tensor<pair<type*, dimensions>, 1>& inputs_pair,
    LayerForwardPropagation* layer_forward_propagation,
    const bool& is_training
    )
{
    MultiheadAttentionLayerForwardPropagation* multihead_attention_layer_forward_propagation
        = static_cast<MultiheadAttentionLayerForwardPropagation*>(layer_forward_propagation);

    const TensorMap<Tensor<type, 3>> input(inputs_pair(0).first,
                                           inputs_pair(0).second[0],
                                           inputs_pair(0).second[1],
                                           inputs_pair(0).second[2]);

    const TensorMap<Tensor<type, 3>> context(inputs_pair(1).first,
                                             inputs_pair(1).second[0],
                                             inputs_pair(1).second[1],
                                             inputs_pair(1).second[2]);

    Tensor<type, 4>& query = multihead_attention_layer_forward_propagation->query;
    Tensor<type, 4>& key = multihead_attention_layer_forward_propagation->key;
    Tensor<type, 4>& value = multihead_attention_layer_forward_propagation->value;

    Tensor<type, 2>& sample_matrix = multihead_attention_layer_forward_propagation->sample_matrix;

    Tensor<type, 4>& attention_scores = multihead_attention_layer_forward_propagation->attention_scores;
    Tensor<type, 4>& attention_weights = multihead_attention_layer_forward_propagation->
    attention_weights;

     Tensor<type, 4>& attention_outputs =
    multihead_attention_layer_forward_propagation->
    attention_outputs;

     Tensor<type, 4>& projection_outputs =
    multihead_attention_layer_forward_propagation->
    projection_outputs;
     Tensor<type, 3>& outputs =
    multihead_attention_layer_forward_propagation->outputs;

     calculate_transformation(input, query, query_weights,
    query_biases, sample_matrix);

     calculate_transformation(context, key, key_weights,
    key_biases, sample_matrix);

     calculate_transformation(context, value, value_weights,
    value_biases, sample_matrix);

     compute_attention_scores(query,
                              key,
                              attention_scores,
                              attention_weights);

     compute_attention_outputs(value,
                               attention_weights,
                               attention_outputs);

     calculate_output_projection(attention_outputs,
                                 projection_outputs,
                                 outputs);
}
```

Listing 9: Multi-head attention layer function.

Addition layer

```
void AdditionLayer3D::forward_propagate(
     const Tensor<pair<type*, dimensions>, 1>& inputs_pair,
     LayerForwardPropagation* layer_forward_propagation,
     const bool& is_training
     )
{
     const TensorMap<Tensor<type, 3>> input_1(inputs_pair(0).first
    , inputs_pair(0).second[0], inputs_pair(0).second[1],
    inputs_pair(0).second[2]);
     const TensorMap<Tensor<type, 3>> input_2(inputs_pair(1).first
    , inputs_pair(1).second[0], inputs_pair(1).second[1],
    inputs_pair(1).second[2]);

     AdditionLayer3DForwardPropagation*
    addition_layer_3d_forward_propagation =
```

```cpp
        static_cast<AdditionLayer3DForwardPropagation*>(layer_forward_propagation);

    Tensor<type, 3>& outputs = addition_layer_3d_forward_propagation->outputs;

    outputs.device(*thread_pool_device) = input_1 + input_2;

}
```

Listing 10: Addition layer function.

Normalization layer

```cpp
void NormalizationLayer3D::normalize(
    const Tensor<type, 3>& inputs,
    const type& epsilon,
    Tensor<type, 3>& means,
    Tensor<type, 3>& standard_deviations,
    Tensor<type, 3>& normalized_inputs
    )
{
    const Index samples_number = inputs.dimension(0);
    const Index inputs_number = inputs.dimension(1);
    const Index inputs_depth = inputs.dimension(2);

    const Eigen::array<Index, 1> normalization_axis{ { 2 } };
    const Eigen::array<Index, 3> range_3{ { samples_number, inputs_number, 1 } };
    const Eigen::array<Index, 3> expand_normalization_axis{ { 1, 1, inputs_depth } };

    means.device(*thread_pool_device) = inputs.mean(normalization_axis)
                                    .reshape(range_3).broadcast(expand_normalization_axis);

    standard_deviations.device(*thread_pool_device) = (inputs - means).pow(2).mean(normalization_axis).sqrt()
                                        .reshape(range_3).broadcast(expand_normalization_axis);

    normalized_inputs.device(*thread_pool_device) = (inputs - means) / (standard_deviations + epsilon);
}
```

Listing 11: Normalization function.

```cpp
void NormalizationLayer3D::compute_affine_transformation(
    const Tensor<type, 1>& gammas,
    const Tensor<type, 1>& betas,
    Tensor<type, 3>& outputs
    )
{
    multiply_matrices(thread_pool_device, outputs, gammas);

    sum_matrices(thread_pool_device, betas, outputs);
}
```

Listing 12: Affine transformation function.

```
void NormalizationLayer3D::forward_propagate(
    const Tensor<pair<type*, dimensions>, 1>& inputs_pair,
    LayerForwardPropagation* layer_forward_propagation,
    const bool& is_training
    )
{
    const Index samples_number = inputs_pair(0).second[0];
    const Index inputs_number = inputs_pair(0).second[1];
    const Index inputs_depth = inputs_pair(0).second[2];

    const TensorMap<Tensor<type, 3>> inputs(inputs_pair(0).first,
    samples_number, inputs_number, inputs_depth);

    NormalizationLayer3DForwardPropagation*
   normalization_layer_3d_forward_propagation =
        static_cast<NormalizationLayer3DForwardPropagation*>(
   layer_forward_propagation);

    Tensor<type, 3>& normalized_inputs =
   normalization_layer_3d_forward_propagation->normalized_inputs;
    Tensor<type, 3>& outputs =
   normalization_layer_3d_forward_propagation->outputs;

    Tensor<type, 3>& means =
   normalization_layer_3d_forward_propagation->means;
    Tensor<type, 3>& standard_deviations =
   normalization_layer_3d_forward_propagation->
   standard_deviations;
    const type& epsilon =
   normalization_layer_3d_forward_propagation->epsilon;

    normalize(inputs, epsilon, means, standard_deviations,
   normalized_inputs);

    outputs.device(*thread_pool_device) = normalized_inputs;

    compute_affine_transformation(gammas, betas, outputs);
}
```

Listing 13: Normalization layer function.

Perceptron layer

```
void PerceptronLayer3D::calculate_combinations(
    const Tensor<type, 3>& inputs,
    const Tensor<type, 1>& biases,
    const Tensor<type, 2>& synaptic_weights,
    Tensor<type, 3>& combinations
    ) const
{
    const Eigen::array<IndexPair<Index>, 1> contraction_indices =
    {IndexPair<Index>(2, 0)};

    combinations.device(*thread_pool_device) = inputs.contract(
   synaptic_weights, contraction_indices);

    sum_matrices(thread_pool_device, biases, combinations);
}
```

Listing 14: Combination function.

```
void PerceptronLayer3D::calculate_activations(
    const Tensor<type, 3>& combinations,
    Tensor<type, 3>& activations
    ) const
{
    switch(activation_function)
    {
    case ActivationFunction::Linear: linear(combinations,
   activations); return;

    case ActivationFunction::RectifiedLinear: rectified_linear(
   combinations, activations); return;

    default: return;
    }
}
```

Listing 15: Activations function.

```
template <int rank>
void linear(
    const Tensor<type, rank>& x,
    Tensor<type, rank>& y
    ) const
{
    y.device(*thread_pool_device) = x;
}
```

Listing 16: Linear activation function.

```
template <int rank>
void rectified_linear(
    const Tensor<type, rank>& x,
    Tensor<type, rank>& y
    ) const
{
    y.device(*thread_pool_device) = x.cwiseMax(type(0));
}
```

Listing 17: ReLU activation function.

```cpp
void PerceptronLayer3D::forward_propagate(
    const Tensor<pair<type*, dimensions>, 1>& inputs_pair,
    LayerForwardPropagation* layer_forward_propagation,
    const bool& is_training
    )
{
    const TensorMap<Tensor<type, 3>> inputs(inputs_pair(0).first,
        inputs_pair(0).second[0],
        inputs_pair(0).second[1],
        inputs_pair(0).second[2]);

    PerceptronLayer3DForwardPropagation*
   perceptron_layer_3d_forward_propagation =
        static_cast<PerceptronLayer3DForwardPropagation*>(
   layer_forward_propagation);

    Tensor<type, 3>& outputs =
   perceptron_layer_3d_forward_propagation->outputs;

    calculate_combinations(inputs,
                           biases,
                           synaptic_weights,
                           outputs);

    calculate_activations(outputs,
                          outputs);
}
```

Listing 18: Perceptron layer function.

B.6 Cross-Entropy Error

```cpp
void CrossEntropyError3D::calcualte_log_softmax_error(
    const Tensor<type, 3>& outputs,
    const Tensor<type, 2>& targets,
    Tensor<type, 3>& shifted_outputs,
    Tensor<type, 3>& exp_outputs,
    Tensor<type, 2>& sumexp_outputs,
    Tensor<type, 2>& errors,
    type& error
    ) const
{
    const Index batch_samples_number = outputs.dimension(0);
    const Index outputs_number = outputs.dimension(1);
    const Index outputs_depth = outputs.dimension(2);

    const Eigen::array<Index, 1> softmax_dimension{ { 2 } };
    const Eigen::array<Index, 3> range_3{ { batch_samples_number,
    outputs_number, 1 } };
    const Eigen::array<Index, 3> expand_softmax_dim{ { 1, 1,
   outputs_depth } };

    Tensor<bool, 2> mask;
    Tensor<type, 0> mask_sum;
    Tensor<type, 0> cross_entropy_error;

    mask.device(*thread_pool_device) = targets != targets.
   constant(0);

    mask_sum = mask.cast<type>().sum();

    shifted_outputs.device(*thread_pool_device) = outputs -
   outputs.maximum(softmax_dimension)

      .eval()

      .reshape(range_3)

      .broadcast(expand_softmax_dim);

    exp_outputs.device(*thread_pool_device) = shifted_outputs.exp
   ();

    sumexp_outputs.device(*thread_pool_device) = exp_outputs.sum(
   softmax_dimension);

#pragma omp parallel for

    for (Index i = 0; i < batch_samples_number; i++)
        for (Index j = 0; j < outputs_number; j++)
            errors(i, j) = log(sumexp_outputs(i, j)) -
   shifted_outputs(i, j, Index(targets(i, j)));

    errors.device(*thread_pool_device) = errors * mask.cast<type
   >();

    cross_entropy_error.device(*thread_pool_device) = errors.sum
   ();

    error = cross_entropy_error(0) / mask_sum(0);
}
```

Listing 19: Log-softmax cross-entropy error.

```
void CrossEntropyError3D::calculate_output_delta(
    const Batch& batch,
    ForwardPropagation& forward_propagation,
    BackPropagation& back_propagation
    ) const
{
    // Batch data

    const Index batch_samples_number = batch.
   get_batch_samples_number();

    const pair<type*, dimensions> targets_pair = batch.
   get_targets_pair();

```

```
    const Index outputs_number = targets_pair.second[1];

    const TensorMap<Tensor<type, 2>> targets(targets_pair.first,
        batch_samples_number,
        outputs_number);

    // Forward propagation data

    const pair<type*, dimensions> outputs_pair =
   forward_propagation.get_last_trainable_layer_outputs_pair();

    const Index outputs_depth = outputs_pair.second[2];

    const TensorMap<Tensor<type, 3>> outputs(outputs_pair.first,
                                             batch_samples_number
   ,
                                             outputs_number,
                                             outputs_depth);

    // Back propagation data

    const Index layers_number = back_propagation.neural_network.
   layers.size();

    pair<type*, dimensions> output_deltas_pair = back_propagation
   .get_output_deltas_pair();

    TensorMap<Tensor<type, 3>> output_deltas(output_deltas_pair.
   first, batch_samples_number,

          outputs_number,

          outputs_depth);


    Tensor<type, 3>& exp_outputs = back_propagation.exp_outputs;
    Tensor<type, 2>& sumexp_outputs = back_propagation.
   sumexp_outputs;
    Tensor<bool, 2>& mask = back_propagation.mask;

    const Tensor<type, 0> mask_sum = mask.cast<type>().sum();

    output_deltas.device(*thread_pool_device) = exp_outputs;

    divide_matrices(thread_pool_device, output_deltas,
   sumexp_outputs);

#pragma omp parallel for

    for (Index i = 0; i < batch_samples_number; i++)
        for (Index j = 0; j < outputs_number; j++)
            output_deltas(i, j, Index(targets(i, j))) -= 1;

    multiply_matrices(thread_pool_device, output_deltas, mask.
    cast<type>() / mask_sum(0));
}
```

Listing 20: Output deltas.

B.7 Transformer Back-Propagation

Perceptron layer

```
void PerceptronLayer3D::calculate_activations_derivatives(
     const Tensor<type, 3>& combinations,
     Tensor<type, 3>& activations,
     Tensor<type, 3>& activations_derivatives
     ) const
{
     switch(activation_function)
     {
     case ActivationFunction::Linear: linear_derivatives(
    combinations,

    activations,

    activations_derivatives);
          return;

     case ActivationFunction::RectifiedLinear:
    rectified_linear_derivatives(combinations,

                 activations,

                 activations_derivatives);
          return;

     default:

          return;
     }
}
```

Listing 21: Activations derivatives.

```
template <int rank>
void linear_derivatives(
     const Tensor<type, rank>& x,
     Tensor<type, rank>& y,
     Tensor<type, rank>& dy_dx
     ) const
{
     dy_dx.setConstant(type(1));
}
```

Listing 22: Linear activation derivatives.

```cpp
template <int rank>
void rectified_linear_derivatives(
    const Tensor<type, rank>& x,
    Tensor<type, rank>& y,
    Tensor<type, rank>& dy_dx
    ) const
{
    dy_dx.device(*thread_pool_device) = (y > 0).select(x.constant
   (type(1)), x.constant(type(0)));
}
```

Listing 23: ReLU activation derivatives.

```cpp
void PerceptronLayer3D::back_propagate(
    const Tensor<pair<type*, dimensions>, 1>& inputs_pair,
    const Tensor<pair<type*, dimensions>, 1>& deltas_pair,
    LayerForwardPropagation* forward_propagation,
    LayerBackPropagation* back_propagation
    ) const
{
    const TensorMap<Tensor<type, 3>> inputs(inputs_pair(0).first,
                                            inputs_pair(0).second
   [0],
                                            inputs_pair(0).second
   [1],
                                            inputs_pair(0).second
   [2]);

    if (deltas_pair.size() > 1)     add_deltas(deltas_pair);

    const TensorMap<Tensor<type, 3>> deltas(deltas_pair(0).first,
                                            deltas_pair(0).second
   [0],
                                            deltas_pair(0).second
   [1],
                                            deltas_pair(0).second
   [2]);

    // Forward propagation data

    const PerceptronLayer3DForwardPropagation*
   perceptron_layer_3d_forward_propagation =
            static_cast<PerceptronLayer3DForwardPropagation*>(
   forward_propagation);

    const Tensor<type, 3>& activations_derivatives =
   perceptron_layer_3d_forward_propagation->
   activations_derivatives;

    // Back propagation data

    PerceptronLayer3DBackPropagation*
   perceptron_layer_3d_back_propagation =
            static_cast<PerceptronLayer3DBackPropagation*>(
   back_propagation);
```

```

    Tensor<type, 3>& error_combinations_derivatives = 
    perceptron_layer_3d_back_propagation->
    error_combinations_derivatives;

    Tensor<type, 3>& input_derivatives = 
    perceptron_layer_3d_back_propagation->input_derivatives;

    Tensor<type, 1>& biases_derivatives = 
    perceptron_layer_3d_back_propagation->biases_derivatives;
    Tensor<type, 2>& synaptic_weights_derivatives = 
    perceptron_layer_3d_back_propagation->
    synaptic_weights_derivatives;

    const Eigen::array<IndexPair<Index>, 2> 
    double_contraction_indices = { IndexPair<Index>(0, 0), 
    IndexPair<Index>(1, 1) };

    const Eigen::array<IndexPair<Index>, 1> 
    single_contraction_indices = { IndexPair<Index>(2, 1) };

    calculate_activations_derivatives(outputs,
                                      outputs,
                                      activations_derivatives);

    error_combinations_derivatives.device(*thread_pool_device)
        = deltas * activations_derivatives;

    // Parameters derivatives

    biases_derivatives.device(*thread_pool_device)
        = error_combinations_derivatives.sum(Eigen::array<Index, 
    2>({0, 1}));

    synaptic_weights_derivatives.device(*thread_pool_device)
        = inputs.contract(error_combinations_derivatives, 
    double_contraction_indices);

    // Inputs derivatives

    input_derivatives.device(*thread_pool_device)
        = error_combinations_derivatives.contract(
    synaptic_weights, single_contraction_indices);
}
```

Listing 24: Perceptron layer back-propagation.

Normalization layer

```
void NormalizationLayer3D::back_propagate(
    const Tensor<pair<type*, dimensions>, 1>& inputs_pair,
    const Tensor<pair<type*, dimensions>, 1>& deltas_pair,
    LayerForwardPropagation* forward_propagation,
    LayerBackPropagation* back_propagation
    ) const
{
    Index batch_samples_number = inputs_pair(0).second[0];

    const TensorMap<Tensor<type, 3>> inputs(inputs_pair(0).first,
                                            batch_samples_number,
                                            inputs_pair(0).second[1],
                                            inputs_pair(0).second[2]);

    if (deltas_pair.size() > 1)     add_deltas(deltas_pair);

    const TensorMap<Tensor<type, 3>> deltas(deltas_pair(0).first,
                                            deltas_pair(0).second[0],
                                            deltas_pair(0).second[1],
                                            deltas_pair(0).second[2]);

    // Forward propagation data

    const NormalizationLayer3DForwardPropagation* normalization_layer_3d_forward_propagation =
        static_cast<NormalizationLayer3DForwardPropagation*>(forward_propagation);

    const Tensor<type, 3>& normalized_inputs = normalization_layer_3d_forward_propagation->normalized_inputs;

    const Tensor<type, 3>& means = normalization_layer_3d_forward_propagation->means;
    const Tensor<type, 3>& standard_deviations = normalization_layer_3d_forward_propagation->standard_deviations;

    const TensorMap<Tensor<type, 2>> standard_deviations_matrix((type*)standard_deviations.data(), batch_samples_number, inputs_number);

    const type& epsilon = normalization_layer_3d_forward_propagation->epsilon;

    // Back propagation data

    NormalizationLayer3DBackPropagation* normalization_layer_3d_back_propagation =
        static_cast<NormalizationLayer3DBackPropagation*>(back_propagation);

    Tensor<type, 1>& gammas_derivatives = normalization_layer_3d_back_propagation->gammas_derivatives;
    Tensor<type, 1>& betas_derivatives =
    normalization_layer_3d_back_propagation->betas_derivatives;

    Tensor<type, 3>& scaled_deltas =
    normalization_layer_3d_back_propagation->scaled_deltas;
    Tensor<type, 3>& standard_deviation_derivatives =
    normalization_layer_3d_back_propagation->
    standard_deviation_derivatives;
    Tensor<type, 2>& aux_2d =
    normalization_layer_3d_back_propagation->aux_2d;

    Tensor<type, 3>& input_derivatives =
    normalization_layer_3d_back_propagation->input_derivatives;

    // Parameters derivatives

    gammas_derivatives.device(*thread_pool_device) = (
    normalized_inputs * deltas).sum(Eigen::array<Index, 2>({ 0, 1
    }));

    betas_derivatives.device(*thread_pool_device) = deltas.sum(
    Eigen::array<Index, 2>({ 0, 1 }));

    // Input derivatives

    standard_deviation_derivatives.device(*thread_pool_device) =
    normalized_inputs;

    scaled_deltas.device(*thread_pool_device) = deltas;

    multiply_matrices(thread_pool_device, scaled_deltas, gammas);

    aux_2d.device(*thread_pool_device) = 1 / type(inputs_depth) *
     (scaled_deltas * normalized_inputs).sum(Eigen::array<Index,
    1>({ 2 })) / (standard_deviations_matrix + epsilon);

    multiply_matrices(thread_pool_device,
    standard_deviation_derivatives, aux_2d);

    scaled_deltas.device(*thread_pool_device) = scaled_deltas / (
    standard_deviations + epsilon);

    input_derivatives.device(*thread_pool_device) = scaled_deltas
     - standard_deviation_derivatives;

    aux_2d.device(*thread_pool_device) = 1 / type(inputs_depth) *
     scaled_deltas.sum(Eigen::array<Index, 1>({ 2 }));

    substract_matrices(thread_pool_device, aux_2d,
    input_derivatives);
}
```

Listing 25: Normalization layer back-propagation.

Addition layer

```
void AdditionLayer3D::back_propagate(
    const Tensor<pair<type*, dimensions>, 1>& inputs_pair,
    const Tensor<pair<type*, dimensions>, 1>& deltas_pair,
    LayerForwardPropagation* forward_propagation,
    LayerBackPropagation* back_propagation
    ) const
{
    const TensorMap<Tensor<type, 3>> deltas(deltas_pair(0).first,
                                           deltas_pair(0).second
   [0],
                                           deltas_pair(0).second
   [1],
                                           deltas_pair(0).second
   [2]);

    // Back propagation data

    AdditionLayer3DBackPropagation*
   addition_layer_3d_back_propagation =
        static_cast<AdditionLayer3DBackPropagation*>(
   back_propagation);

    Tensor<type, 3>& input_1_derivatives =
   addition_layer_3d_back_propagation->input_1_derivatives;

    Tensor<type, 3>& input_2_derivatives =
   addition_layer_3d_back_propagation->input_2_derivatives;

    // Inputs derivatives

    input_1_derivatives.device(*thread_pool_device) = deltas;
    input_2_derivatives.device(*thread_pool_device) = deltas;
}
```

Listing 26: Addition layer back-propagation.

Multi-head attention layer

```
void MultiheadAttentionLayer::back_propagate(const Tensor<pair<
   type*, dimensions>, 1>& inputs_pair,
                                                    const
   Tensor<pair<type*, dimensions>, 1>& deltas_pair,

   LayerForwardPropagation* forward_propagation,

   LayerBackPropagation* back_propagation) const
{
    const TensorMap<Tensor<type, 3>> input(inputs_pair(0).first,
                                          inputs_pair(0).second
   [0],
                                          inputs_pair(0).second
   [1],
                                          inputs_pair(0).second
   [2]);
```

```

  const TensorMap<Tensor<type, 3>> context(inputs_pair(1).first
,
                                           inputs_pair(1).
second[0],
                                           inputs_pair(1).
second[1],
                                           inputs_pair(1).
second[2]);

  const TensorMap<Tensor<type, 3>> deltas(deltas_pair(0).first,
                                          deltas_pair(0).second
[0],
                                          deltas_pair(0).second
[1],
                                          deltas_pair(0).second
[2]);

  Index batch_samples_number = inputs_pair(0).second[0];

  type* query_weights_data = (type*)query_weights.data();
  type* key_weights_data = (type*)key_weights.data();
  type* value_weights_data = (type*)value_weights.data();
  type* projection_weights_data = (type*)projection_weights.
data();

  // Forward propagation data

  const MultiheadAttentionLayerForwardPropagation*
multihead_attention_layer_forward_propagation =
      static_cast<MultiheadAttentionLayerForwardPropagation*>(
forward_propagation);

  const Tensor<type, 4>& attention_weights =
multihead_attention_layer_forward_propagation->
attention_weights;
  const Tensor<type, 4>& attention_outputs =
multihead_attention_layer_forward_propagation->
attention_outputs;

  const Tensor<type, 4>& query =
multihead_attention_layer_forward_propagation-> query;
  const Tensor<type, 4>& key =
multihead_attention_layer_forward_propagation->key;
  const Tensor<type, 4>& value =
multihead_attention_layer_forward_propagation->value;

  type* attention_weights_data = (type*)attention_weights.data
();
  type* attention_outputs_data = (type*)attention_outputs.data
();

  type* query_data = (type*)query.data();
  type* key_data = (type*)key.data();
```

```
    type* value_data = (type*)value.data();

    // Back propagation data

    MultiheadAttentionLayerBackPropagation*
   multihead_attention_layer_back_propagation =
        static_cast<MultiheadAttentionLayerBackPropagation*>(
   back_propagation);

    Tensor<type, 3>& projection_weights_derivatives =
   multihead_attention_layer_back_propagation->
   projection_weights_derivatives;

    Tensor<type, 4>& error_attention_scores_derivatives =
   multihead_attention_layer_back_propagation->
   error_attention_scores_derivatives;
    Tensor<type, 4>& error_attention_weights_derivatives =
   multihead_attention_layer_back_propagation->
   error_attention_weights_derivatives;
    Tensor<type, 4>& error_attention_output_derivatives =
   multihead_attention_layer_back_propagation->
   error_attention_output_derivatives;

    Tensor<type, 2>& sample_deltas =
   multihead_attention_layer_back_propagation->sample_deltas;

    Tensor<type, 4>& error_query_derivatives =
   multihead_attention_layer_back_propagation->
   error_query_derivatives;
    Tensor<type, 4>& error_key_derivatives =
   multihead_attention_layer_back_propagation->
   error_key_derivatives;
    Tensor<type, 4>& error_value_derivatives =
   multihead_attention_layer_back_propagation->
   error_value_derivatives;

    Tensor<type, 3>& query_weights_derivatives =
   multihead_attention_layer_back_propagation->
   query_weights_derivatives;
    Tensor<type, 3>& key_weights_derivatives =
   multihead_attention_layer_back_propagation->
   key_weights_derivatives;
    Tensor<type, 3>& value_weights_derivatives =
   multihead_attention_layer_back_propagation->
   value_weights_derivatives;

    Tensor<type, 3>& input_derivatives =
   multihead_attention_layer_back_propagation->input_derivatives;
    input_derivatives.setZero();
    Tensor<type, 3>& context_derivatives =
   multihead_attention_layer_back_propagation->
   context_derivatives;
    context_derivatives.setZero();

    Tensor<type, 1>& aux_rows =
multihead_attention_layer_back_propagation->aux_rows;

    Tensor<type, 2>& query_biases_derivatives =
multihead_attention_layer_back_propagation->
query_biases_derivatives;
    Tensor<type, 2>& key_biases_derivatives =
multihead_attention_layer_back_propagation->
key_biases_derivatives;
    Tensor<type, 2>& value_biases_derivatives =
multihead_attention_layer_back_propagation->
value_biases_derivatives;
    Tensor<type, 1>& projection_biases_derivatives =
multihead_attention_layer_back_propagation->
projection_biases_derivatives;

    type* projection_weights_derivatives_data =
projection_weights_derivatives.data();

    type* error_attention_scores_derivatives_data =
error_attention_scores_derivatives.data();
    type* error_attention_weights_derivatives_data =
error_attention_weights_derivatives.data();
    type* error_attention_output_derivatives_data =
error_attention_output_derivatives.data();

    type* error_query_derivatives_data = error_query_derivatives.
data();
    type* error_key_derivatives_data = error_key_derivatives.data
();
    type* error_value_derivatives_data = error_value_derivatives.
data();

    type* query_weights_derivatives_data =
query_weights_derivatives.data();
    type* key_weights_derivatives_data = key_weights_derivatives.
data();
    type* value_weights_derivatives_data =
value_weights_derivatives.data();

    type* query_biases_derivatives_data =
query_biases_derivatives.data();
    type* key_biases_derivatives_data = key_biases_derivatives.
data();
    type* value_biases_derivatives_data =
value_biases_derivatives.data();

    for (Index head_index = 0; head_index < heads_number;
head_index++)
    {
        type* head_query_weights_data = query_weights_data +
head_index * depth * hidden_depth;
        type* head_key_weights_data = key_weights_data +
head_index * depth * hidden_depth;
```

```
        type* head_value_weights_data = value_weights_data + head_index * depth * hidden_depth;

        type* head_query_data = query_data + head_index * input_size * hidden_depth * batch_samples_number;
        type* head_key_data = key_data + head_index * context_size * hidden_depth * batch_samples_number;
        type* head_value_data = value_data + head_index * context_size * hidden_depth * batch_samples_number;

        type* head_projection_weights_data = projection_weights_data + head_index * hidden_depth * depth;
        type* head_attention_weights_data = attention_weights_data + head_index * context_size * input_size * batch_samples_number;
        type* head_attention_outputs_data = attention_outputs_data + head_index * input_size * hidden_depth * batch_samples_number;

        type* head_projection_weights_derivatives_data = projection_weights_derivatives_data + head_index * hidden_depth * depth;

        type* head_attention_scores_derivatives_data = error_attention_scores_derivatives_data + head_index * context_size * input_size * batch_samples_number;
        type* head_attention_weights_derivatives_data = error_attention_weights_derivatives_data + head_index * context_size * input_size * batch_samples_number;
        type* head_attention_output_derivatives_data = error_attention_output_derivatives_data + head_index * input_size * hidden_depth * batch_samples_number;

        type* head_query_derivatives_data = error_query_derivatives_data + head_index * input_size * hidden_depth * batch_samples_number;
        type* head_key_derivatives_data = error_key_derivatives_data + head_index * context_size * hidden_depth * batch_samples_number;
        type* head_value_derivatives_data = error_value_derivatives_data + head_index * context_size * hidden_depth * batch_samples_number;

        type* head_query_weights_derivatives_data = query_weights_derivatives_data + head_index * depth * hidden_depth;
        type* head_key_weights_derivatives_data = key_weights_derivatives_data + head_index * depth * hidden_depth;
        type* head_value_weights_derivatives_data = value_weights_derivatives_data + head_index * depth * hidden_depth;

        type* head_query_biases_derivatives_data =
query_biases_derivatives_data + head_index * hidden_depth;
        type* head_key_biases_derivatives_data =
key_biases_derivatives_data + head_index * hidden_depth;
        type* head_value_biases_derivatives_data =
value_biases_derivatives_data + head_index * hidden_depth;

        const TensorMap<Tensor<type, 2>> head_query_weights(
head_query_weights_data, depth, hidden_depth);
        const TensorMap<Tensor<type, 2>> head_key_weights(
head_key_weights_data, depth, hidden_depth);
        const TensorMap<Tensor<type, 2>> head_value_weights(
head_value_weights_data, depth, hidden_depth);

        const TensorMap<Tensor<type, 3>> head_query(
head_query_data, input_size, hidden_depth,
batch_samples_number);
        const TensorMap<Tensor<type, 3>> head_key(head_key_data,
context_size, hidden_depth, batch_samples_number);
        const TensorMap<Tensor<type, 3>> head_value(
head_value_data, context_size, hidden_depth,
batch_samples_number);

        const TensorMap<Tensor<type, 2>> head_projection_weights(
head_projection_weights_data, hidden_depth, depth);

        const TensorMap<Tensor<type, 3>> head_attention_weights(
head_attention_weights_data, context_size, input_size,
batch_samples_number);
        const TensorMap<Tensor<type, 3>> head_attention_outputs(
head_attention_outputs_data, input_size, hidden_depth,
batch_samples_number);

        TensorMap<Tensor<type, 2>>
head_projection_weights_derivatives(
head_projection_weights_derivatives_data, hidden_depth, depth)
;

        TensorMap<Tensor<type, 3>>
head_attention_scores_derivatives(
head_attention_scores_derivatives_data, context_size,
input_size, batch_samples_number);
        TensorMap<Tensor<type, 3>>
head_attention_weights_derivatives(
head_attention_weights_derivatives_data, context_size,
input_size, batch_samples_number);
        TensorMap<Tensor<type, 3>>
head_attention_output_derivatives(
head_attention_output_derivatives_data, input_size,
hidden_depth, batch_samples_number);

        TensorMap<Tensor<type, 3>> head_query_derivatives(
head_query_derivatives_data, input_size, hidden_depth,
batch_samples_number);
        TensorMap<Tensor<type, 3>> head_key_derivatives(
head_key_derivatives_data, context_size, hidden_depth,
batch_samples_number);
     TensorMap<Tensor<type, 3>> head_value_derivatives(
head_value_derivatives_data, context_size, hidden_depth,
batch_samples_number);

     TensorMap<Tensor<type, 2>> head_query_weights_derivatives
(head_query_weights_derivatives_data, depth, hidden_depth);
     TensorMap<Tensor<type, 2>> head_key_weights_derivatives(
head_key_weights_derivatives_data, depth, hidden_depth);
     TensorMap<Tensor<type, 2>> head_value_weights_derivatives
(head_value_weights_derivatives_data, depth, hidden_depth);

     TensorMap<Tensor<type, 1>> head_query_biases_derivatives(
head_query_biases_derivatives_data, hidden_depth);
     TensorMap<Tensor<type, 1>> head_key_biases_derivatives(
head_key_biases_derivatives_data, hidden_depth);
     TensorMap<Tensor<type, 1>> head_value_biases_derivatives(
head_value_biases_derivatives_data, hidden_depth);

     // WEIGHTS DERIVATIVES

     // Projection weights derivatives

     head_projection_weights_derivatives.device(*
thread_pool_device)
         = head_attention_outputs.contract(deltas,
projection_weights_derivatives_contraction_indices);

     // Attention output derivatives

     for (Index sample_index = 0; sample_index <
batch_samples_number; sample_index++)
     {
         type* sample_attention_output_derivatives_data =
head_attention_output_derivatives_data + sample_index *
input_size * hidden_depth;

         TensorMap<Tensor<type, 2>>
sample_attention_output_derivatives(
sample_attention_output_derivatives_data, input_size,
hidden_depth);

         sample_deltas = deltas.chip(sample_index, 0);

         sample_attention_output_derivatives.device(*
thread_pool_device)
             = sample_deltas.contract(head_projection_weights,
 A_BT);
     }

     // Value derivatives

     batch_matrix_multiplication(thread_pool_device,
head_attention_weights , head_attention_output_derivatives ,
head_value_derivatives , A_B);

    // Value weights derivatives

    head_value_weights_derivatives.device(*thread_pool_device
)
        = context.contract(head_value_derivatives ,
transformation_weights_derivatives_contraction_indices);

    // Attention weights derivatives

    batch_matrix_multiplication(thread_pool_device ,
head_value , head_attention_output_derivatives ,
head_attention_weights_derivatives , A_BT);

    // Attention scores derivatives

    softmax_derivatives_times_tensor(head_attention_weights ,
head_attention_weights_derivatives ,
head_attention_scores_derivatives , aux_rows);

    head_attention_scores_derivatives.device(*
thread_pool_device) = head_attention_scores_derivatives *
scaling_factor;

    // Query derivatives

    batch_matrix_multiplication(thread_pool_device ,
head_attention_scores_derivatives , head_key ,
head_query_derivatives , AT_B);

    // Key derivatives

    batch_matrix_multiplication(thread_pool_device ,
head_attention_scores_derivatives , head_query ,
head_key_derivatives , A_B);

    // Query weights derivatives

    head_query_weights_derivatives.device(*thread_pool_device
)
        = input.contract(head_query_derivatives ,
transformation_weights_derivatives_contraction_indices);

    // Key weights derivatives

    head_key_weights_derivatives.device(*thread_pool_device)
        = context.contract(head_key_derivatives ,
transformation_weights_derivatives_contraction_indices);

    // INPUTS DERIVATIVES

    for (Index sample_index = 0; sample_index <
batch_samples_number; sample_index++)
        {
            type* sample_query_derivatives_data =
head_query_derivatives_data + sample_index * input_size *
hidden_depth;
            type* sample_key_derivatives_data =
head_key_derivatives_data + sample_index * context_size *
hidden_depth;
            type* sample_value_derivatives_data =
head_value_derivatives_data + sample_index * context_size *
hidden_depth;

            const TensorMap<Tensor<type, 2>>
sample_query_derivatives(sample_query_derivatives_data,
input_size, hidden_depth);
            const TensorMap<Tensor<type, 2>>
sample_key_derivatives(sample_key_derivatives_data,
context_size, hidden_depth);
            const TensorMap<Tensor<type, 2>>
sample_value_derivatives(sample_value_derivatives_data,
context_size, hidden_depth);

            // Input derivatives

            input_derivatives.chip(sample_index, 0).device(*
thread_pool_device)
                += sample_query_derivatives.contract(
head_query_weights, A_BT);

            // Context derivatives
            context_derivatives.chip(sample_index, 0).device(*
thread_pool_device)
                += sample_key_derivatives.contract(
head_key_weights, A_BT)
                + sample_value_derivatives.contract(
head_value_weights, A_BT);
        }

        // BIASES DERIVATIVES

        head_query_biases_derivatives.device(*thread_pool_device)
 = head_query_derivatives.sum(biases_derivatives_sum_indices);

        head_key_biases_derivatives.device(*thread_pool_device) =
 head_key_derivatives.sum(biases_derivatives_sum_indices);

        head_value_biases_derivatives.device(*thread_pool_device)
 = head_value_derivatives.sum(biases_derivatives_sum_indices);
    }

    projection_biases_derivatives.device(*thread_pool_device) =
deltas.sum(projection_biases_derivatives_sum_indices);
}
```

Listing 27: Multi-head attention layer back-propagation.

Embedding layer

```
void EmbeddingLayer::back_propagate(
    const Tensor<pair<type*, dimensions>, 1>& inputs_pair,
    const Tensor<pair<type*, dimensions>, 1>& deltas_pair,
    LayerForwardPropagation* forward_propagation,
    LayerBackPropagation* back_propagation
    ) const
{
    const Index batch_samples_number = inputs_pair(0).second[0];
    const Index inputs_number = inputs_pair(0).second[1];

    const TensorMap<Tensor<type, 2>> inputs(inputs_pair(0).first, batch_samples_number, inputs_number);

    if (deltas_pair.size() > 1)     add_deltas(deltas_pair);

    const TensorMap<Tensor<type, 3>> deltas(deltas_pair(0).first, batch_samples_number, inputs_number, deltas_pair(0).second[2]);

    // Forward propagation data

    EmbeddingLayerForwardPropagation* embedding_layer_forward_propagation = static_cast<EmbeddingLayerForwardPropagation*>(forward_propagation);

    // Back propagation data

    EmbeddingLayerBackPropagation* embedding_layer_back_propagation = static_cast<EmbeddingLayerBackPropagation*>(back_propagation);

    Tensor<type, 2>& sample_deltas = embedding_layer_back_propagation->sample_deltas;
    Tensor<type, 2>& embedding_weights_derivatives = embedding_layer_back_propagation->embedding_weights_derivatives;

    // Parameters derivatives

    embedding_weights_derivatives.setZero();

    for (Index i = 0; i < batch_samples_number; i++)
    {
        if(positional_encoding)
            sample_deltas.device(*thread_pool_device) = deltas.chip(i, 0) * sample_deltas.constant(sqrt(depth));
        else
            sample_deltas.device(*thread_pool_device) = deltas.chip(i, 0);

        for (Index j = 0; j < inputs_number; j++)
        {
            embedding_weights_derivatives.chip(Index(inputs(i, j)
), 0).device(*thread_pool_device) += sample_deltas.chip(j, 0);
        }
    }
}
```

Listing 28: Embedding layer back-propagation.

B.8 Adaptive Moment Estimation

```
void AdaptiveMomentEstimation::update_parameters(
    BackPropagation& back_propagation,
    AdaptiveMomentEstimationData& optimization_data
    ) const
{
    NeuralNetwork* neural_network = loss_index->
get_neural_network();

    Index& iteration = optimization_data.iteration;

    const Tensor<type, 1>& gradient = back_propagation.gradient;

    Tensor<type, 1>& gradient_exponential_decay =
optimization_data.gradient_exponential_decay;

    Tensor<type, 1>& square_gradient_exponential_decay =
optimization_data.square_gradient_exponential_decay;

    Tensor<type, 1>& parameters = back_propagation.parameters;

    const type bias_correction =
            sqrt(type(1) - pow(beta_2, type(iteration))) /
            (type(1) - pow(beta_1, type(iteration)));

    gradient_exponential_decay.device(*thread_pool_device)
        = gradient * (type(1) - beta_1) +
gradient_exponential_decay * beta_1;

    square_gradient_exponential_decay.device(*thread_pool_device)
        = gradient.square() * (type(1) - beta_2) +
square_gradient_exponential_decay * beta_2;

    if (!use_custom_learning_rate)
    {
        parameters.device(*thread_pool_device)
            -= (learning_rate * bias_correction) *
gradient_exponential_decay / (
square_gradient_exponential_decay.sqrt() + epsilon);
    }
    else
    {
        const type warmup_steps = 4000;
        type& step = optimization_data.step;

        const type custom_learning_rate = learning_rate * min(pow(step, -0.5), step * pow(warmup_steps, -1.5));

        parameters.device(*thread_pool_device)
            -= (custom_learning_rate * bias_correction) * gradient_exponential_decay / (square_gradient_exponential_decay.sqrt() + epsilon);

        step++;
    }

    optimization_data.iteration++;

    // Update neural network parameters

    neural_network->set_parameters(parameters);
}
```

Listing 29: ADAM update rule.