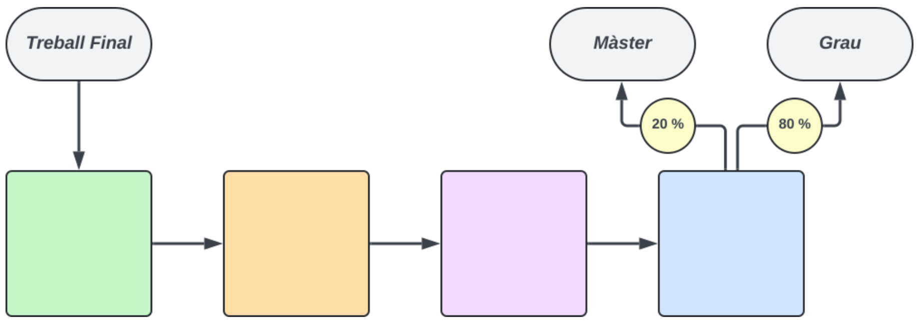
The Underlying Mathematics of Natural Language Processing

Bachelor's Thesis in Mathematics



Pablo Tikas Pueyo

Supervised by Roberto Rubio Núñez

July 2024

UAB
Universitat Autònoma de Barcelona

Abstract

In this thesis, we present a precise mathematical definition of the transformer model in Natural Language Processing, exploring its main components: word embeddings, positional encoding, attention mechanisms, and position-wise neural networks. Our work includes original examples, insights into the connection between mathematical formalization and the model's requirements, and independent calculations.

Acknowledgements

I would like to express my gratitude to my tutor Roberto Rubio Núñez for his valuable guidance and insightful feedback throughout the development of this thesis. His expertise and dedication have been pivotal in the completion of this work.

I am also profoundly grateful to my close ones for their unconditional support during this journey. Their patience and encouragement have been a constant source of strength. Thanks for everything.

Contents

1 Introduction 1

2 Preliminary concepts 2
- 2.1 Supervised learning 2
- 2.2 Neural networks 4

3 Text representation techniques 8
- 3.1 Word embeddings 8
- 3.2 Positional encoding 16

4 Transformer architecture 21
- 4.1 Attention mechanisms 21
- 4.2 Position-wise neural networks and normalization 26
- 4.3 Output distribution and training 28

5 Conclusions 31

1 Introduction

Natural Language Processing (NLP) is a branch of computer science focused on developing models and systems that equip computers with the ability to understand and reproduce human language.

The origins of NLP date back to the 1950s with the development of machine translation models [Joh+21]. These early models relied on manually programmed sets of complex "underlying rules" governing human language. However, by the late 1980s, a paradigm shift occurred within the theoretical framework of NLP with the conceptualization of deep learning (DL) models. These models proposed the use of neural networks to learn these "underlying rules" of natural language autonomously, without the need for explicit programming [Bas+22]. It was not until the late 2000s that technological advancements enabled the practical implementation of these theoretical DL language models, leading to remarkable results in NLP tasks.

Over the years, various DL models have been proposed for NLP, being recurrent neural networks (RNNs) [Sch19] and Long-Short Term Memory (LSTMs) [WJ16] the most implemented examples. Despite their popularity, RNNs and LSTMs face limitations when handling long sequences of text. To overcome these limitations, Vaswani et al. introduced a novel architecture known as transformers in their seminal paper "Attention is All You Need" (2017) [Vas+23]. Since then, transformers have become the foundation of most large language models (LLMs) we have today, including the well-known GPT (Generative Pre-Trained Transformer).

Despite the growing interest in transformer-based models and their various applications [Udd+24; NCK20], the underlying mathematics behind these models have not been thoroughly explored [Bec23; PH22]. In this work, we aim to provide a mathematical perspective on transformers, delving into the key mathematical concepts that justify their success in sequence processing, and particularly in NLP.

This work is structured to show how to successively build the different components constituting the transformer's architecture, culminating in a mathematical definition of transformers. We will see that transformers are essentially parametric functions that take a snippet of text as an input and return a probability distribution over the words of a fixed vocabulary. This distribution can then used to sample the next word during text generation tasks.

We begin by reviewing supervised learning and neural networks, two foundational concepts for understanding the transformer model. Then, we discuss text representation techniques, delving into word embeddings and positional encoding. We continue with a detailed discussion of the main components of the transformer, with a particular focus on attention mechanisms. Finally, we explore how the transformer generates the output distribution and we briefly discuss its training process.

2 Preliminary concepts

In this section, we review the foundational concepts of supervised learning and neural networks. These two topics provide a solid theoretical background for understanding the more advanced concepts discussed later in this work.

2.1 Supervised learning

Supervised learning (SL) is one of the three primary paradigms in machine learning (ML), alongside unsupervised learning and reinforcement learning. In the most general setting of SL, we are given a finite set of labeled data:

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N} \subset \mathcal{X} \times \mathcal{Y}$$

where $\mathcal{X}$ is called the data space, and $\mathcal{Y}$ is the label space [1]. We work with the hypothesis that there exists an underlying relationship between the elements of $\mathcal{X}$ and $\mathcal{Y}$, and our objective is to infer this relationship based on the elements of $\mathcal{D}$.

From a mathematical standpoint, we identify $\mathcal{X}$ and $\mathcal{Y}$ with Euclidean subspaces, $\mathcal{X} \subset \mathbb{R}^m$ and $\mathcal{Y} \subset \mathbb{R}^n$, as this allows us to translate the problem of inferring this underlying relationship between $\mathcal{X}$ and $\mathcal{Y}$ into a problem of approximating an underlying function

[1]The label set can either be a discrete or continuous set of labels.

$f^* : \mathcal{X} \to \mathcal{Y}$ given a set of known points:

$$f^*(x_i) = y_i \quad \forall \ (x_i, y_i) \in \mathcal{D}.$$

In order to approximate this underlying function $f^* : \mathcal{X} \to \mathcal{Y}$, we propose a model, which is essentially a family of parametrized functions:

$$\{f : \Theta \times \mathcal{X} \to \mathcal{Y}\}$$

where $\Theta \subset \mathbb{R}^p$ is the parameter space. The choice of model is highly context-specific, but in general we require f to be continuous and differentiable with respect to parameters in Θ.

Definition 2.1. *Given a model $\{f_\theta : \mathcal{X} \to \mathcal{Y}, \quad \theta \in \Theta\}$, a loss function is defined as any function $\mathcal{L} : \Theta \longrightarrow \mathbb{R}$ that quantifies the error between the outputs of f_θ and f^* for the elements of the labeled data set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$:*

$$\mathcal{L}(\theta) = \textit{Error}\left[\{(f_\theta(x_i), y_i)\}_{i=1}^N\right].$$

The choice of loss function is also context-specific, but, again, we generally require continuity and differentiability with respect to the parameters. In this context, training the model refers to finding the optimal parameters $\theta^* \in \Theta$ that minimize the loss function:

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

The idea is that, if the choice of model is appropriate and $\mathcal{D}$ contains enough labeled data points, then f_{θ^*} approximates f^* well, and, given an arbitrary $x \in \mathcal{X}$, we can "guess" its corresponding label accurately, i.e. $f_{\theta^*}(x) \approx f^*(x)$.

Example 2.2. *Suppose we have a data set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, where x_i is the square footage of a house "i", and y_i is its actual market price. We propose a linear model:*

$$\{f_{(a,b)}(x) = ax + b, \quad (a, b) \in \Theta \subset \mathbb{R}^2\}$$

and we choose the mean-squared error as the loss function:

$$\mathcal{L}(a,b) = \frac{1}{N}\sum_{i=1}^{N}\left[(ax_i+b)-y_i\right]^2.$$

If there is indeed a linear relationship between square footage and house prices, then by minimizing $\mathcal{L}$ we would find a pair of parameters (a^*,b^*) for which the loss function tends to 0, and, given the square footage x of a new house, we could accurately predict its market price by computing a^*x+b^*.

Proposition 2.3 (Gradient descent algorithm)**.** *Let $F:\mathbb{R}^p\to\mathbb{R}$ be a continuous and differentiable function in a neighbourhood of a point $\mathbf{a}_0\in\mathbb{R}^p$. Then, the direction of steepest decrease of F is known to be given by $-\nabla F(\mathbf{a}_0)$. From this, it follows that, for a small enough step size $\gamma\in\mathbb{R}^+$, the following statement holds true:*

$$\mathbf{a_1} \coloneqq \mathbf{a_0} - \gamma\nabla F(\mathbf{a_0}) \quad\Longrightarrow\quad F(\mathbf{a}_1)\le F(\mathbf{a}_0).$$

We can iterate this process obtaining $\{\mathbf{a}_i\}_{i=0}^m$, where

$$\mathbf{a}_{i+1} \coloneqq \mathbf{a}_i - \gamma_i\nabla F(\mathbf{a}_i)$$

and we have:

$$F(\mathbf{a}_0)\ge F(\mathbf{a}_1)\ge\cdots\ge F(\mathbf{a}_m).$$

Proposition 2.3 provides a heuristic method, known as the gradient descent (GD), for finding a local minimum of an arbitrary function $F:\mathbb{R}^p\to\mathbb{R}$ in the neighbourhood of a point $\mathbf{a}_0\in\mathbb{R}^p$ where F is differentiable. Since any loss function $\mathcal{L}(\theta)$ is required to be differentiable in Θ, applying the GD algorithm is a common approach for optimizing models in SL.

2.2 Neural networks

Transformers are DL models and, as such, their architecture is based on neural networks. Therefore, providing a solid mathematical background on neural networks is fundamental for constructing a mathematical definition of transformers.

Remark 2.4. *In this work, we use the row vectors notation, i.e.* $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^m$ *is expressed as:*

$$\mathbf{x} = \begin{pmatrix} x_1 & x_2 & \dots & x_m \end{pmatrix}$$

and, for $M \in \mathcal{M}_{m\times q}(\mathbb{R})$*, the matrix acts on the right of* $\mathbf{x}$*:*

$$\mathbf{x}\,M = \begin{pmatrix} x_1 & x_2 & \dots & x_m \end{pmatrix} \begin{pmatrix} M_{11} & M_{12} & \dots & M_{1q} \\ M_{21} & M_{22} & \dots & M_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ M_{m1} & M_{m2} & \dots & M_{mq} \end{pmatrix} = \mathbf{y} \in \mathbb{R}^q.$$

Definition 2.5. *A* p*-depth neural network is a parametrized function* $\mathcal{F}_\theta : \mathbb{R}^m \to \mathbb{R}^n$ *which consist of the composition of* p *functions called layers:*

$$\mathcal{F}_\theta = L_p \circ \dots \circ L_1.$$

Each layer $L_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}$ *for* $i \neq p$ *is the composition of an affine transformation and a non-polynomial function* σ_i*, called the activation function, which is applied element-wise, whereas* $L_p : \mathbb{R}^{n_{p-1}} \to \mathbb{R}^{n_p}$ *is simply an affine transformation. Abusing the notation, this can be expressed as:*

$$\begin{aligned} L_i(\mathbf{x}) &= \sigma_i\left(\mathbf{x}W_i + \mathbf{b}_i\right), \quad i \neq p \\ L_p(\mathbf{x}) &= \mathbf{x}W_p + \mathbf{b}_p \end{aligned}$$

where $W_i \in \mathcal{M}_{n_{i-1}\times n_i}$ *are called weights matrices, and* $\mathbf{b}_i \in \mathbb{R}^{n_i}$ *bias vectors. The parameters of neural networks are:*

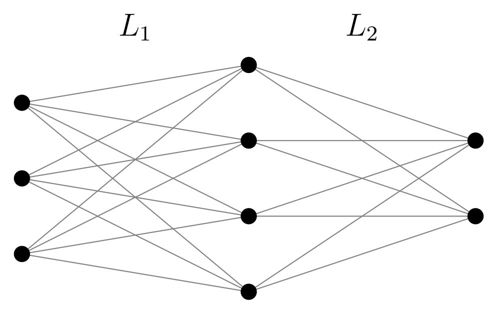
$$\theta = \{W_i^{(j,k)}, b_i^{(l)}\}_{i=1}^p$$

for $j = 1, \dots, n_{i-1}$*, and* $k, l = 1, \dots, n_i$*.*

Remark 2.6 (Graph-like representation of neural networks)**.** *Neural networks can effectively be represented using graphs due to their structural similarities. For instance, for a 2-depth neural network:*

$$\mathcal{F}_\theta : \mathbb{R}^3 \xrightarrow{L_1} \mathbb{R}^4 \xrightarrow{L_2} \mathbb{R}^2$$

can be graphically represented as



where the nodes represent the inputs and the outputs of the layers, and the edges represent the components of the weights matrices. In these representations, the bias vectors and the activation functions are often not explicitly depicted, although they are understood to be present.

Once we have seen the graph-like representations of neural networks, we can provide a brief justification of why they are called "neural networks". The architecture of neural networks is designed to mimic biological neurons: the nodes in the graph correspond to the neurons in the brain, and the weighted edges represent the connections of variable strength between them. Also, biological neurons only fire an electric signal to their neighbours when they are sufficiently stimulated, and this is precisely the role that activation functions play in neural networks.

The following theorem expresses the power and adaptability of neural networks to virtually any type of SL setting:

Theorem 2.7 (Universal Approximation Theorem)**.** *Given a function $f : K \to \mathbb{R}^n$ where $K \subset \mathbb{R}^m$ is a compact set and any $\epsilon > 0$, there exists a 2-depth neural network $\mathcal{F}_\theta : \mathbb{R}^m \to \mathbb{R}^q \to \mathbb{R}^n$ with q sufficiently large such that:*

$$\sup_{\mathbf{x}\in K} \| f(\mathbf{x}) - \mathcal{F}_\theta(\mathbf{x}) \| < \epsilon.$$

From this theorem [Les+92] , it follows that any underlying function $f^* : \mathcal{X} \longrightarrow \mathcal{Y}$ associated to a SL setting can be approximated arbitrarily well by a 2-depth neural network.

To conclude this section, we will discuss the training process of neural networks. Due to the layered structure of neural networks, applying the GD algorithm requires using the chain rule multiple times to compute the gradient of loss functions.

Example 2.8. *Consider the following 2-depth neural network:*

$$\mathbb{R}^2 \xrightarrow{L_1} \mathbb{R} \xrightarrow{L_2} \mathbb{R}$$

with $L_1(\mathbf{x}) = \sigma(z_1(\mathbf{x}))$*,* $L_2(x) = z_2(x)$*, where* σ *is an arbitrary activation function. We have defined* $z_1(\mathbf{x}) = \mathbf{x}\, W_1 + b_1$*,* $z_2(x) = w_2 x + b_2$*, where*

$$W_1 = \begin{pmatrix} W_1^{(1)} \\ W_1^{(2)} \end{pmatrix}.$$

For simplicity, let us assume that our training set consists of a single element $(\mathbf{x}, y)$*. The output of our untrained neural network for the input* $\mathbf{x} = \begin{pmatrix} x_1 & x_2 \end{pmatrix}$ *is:*

$$\hat{y} = L_2(L_1(\mathbf{x})) = z_2(\sigma(z_1(\mathbf{x}))).$$

To compute the gradient of the loss function $\mathcal{L}$*, we need the partial derivatives of* $\mathcal{L}$ *with respect to the weights and biases. For the parameters of the first layer,* $W_1^{(1)}, W_1^{(2)}, b_1$*, we have:*

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W_1^{(j)}} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial \sigma} \frac{\partial \sigma}{\partial z_1} \frac{\partial z_1}{\partial W_1^{(j)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} w_2 \sigma'(z_1) x_j \\ \frac{\partial \mathcal{L}}{\partial b_1} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial \sigma} \frac{\partial \sigma}{\partial z_1} \frac{\partial z_1}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} w_2 \sigma'(z_1). \end{aligned}$$

Now, for the parameters of the second layer, w_2 *and* b_2*:*

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_2} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \sigma(z_1) \\ \frac{\partial \mathcal{L}}{\partial b_2} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}}. \end{aligned}$$

Example 2.8 illustrates how computing the gradient of neural networks often involves redundant calculations. Note how the term $\frac{\partial L}{\partial \hat{y}}$ appears repeatedly in all the partial derivatives of the weights and biases of the network. This redundancy might not be significant in terms of computational complexity for simple neural networks like this one, but for deeper neural networks with many layers, it becomes very inefficient.

Backpropagation is a technique that addresses this inefficiency by computing the partial derivatives in reverse order, i.e. starting with the last layer and ending with the first. This idea of "moving backward" in the neural network allows to reuse intermediate results of the partial derivatives from one layer to compute the partial derivatives of the previous layer (for further information, see [DMC23]).

3 Text representation techniques

In this section, we will delve into text representation techniques. First, we will discuss word embeddings, which will enable us to mathematically represent the concept of semantic similarity. Then, we will explore positional encodings, essential components in transformers due to their non-sequential processing.

3.1 Word embeddings

The first step is defining the basic units of language using a large corpus of text. Let $\mathcal{A}$ denote the alphabet of a language, i.e. the set of all characters in a language, and let C_0 denote a large corpus of text.

Let us note that the corpus of text C_0 is a sequence of characters from $\mathcal{A}$. With the following method, we can create a set $\mathcal{V}$, called the vocabulary, which will contain the most common character patterns in C_0. The elements of $\mathcal{V}$, called the tokens, will be the basic units of language.

1. Let us begin with an initial set $\mathcal{V}_0 = \mathcal{A}$. Then, we identify the ordered pair of characters (α, β) that occurs most frequently in C_0.
2. We define $\mathcal{V}_1 = \mathcal{V}_0 \cup \{\alpha\beta\}$, and $C_1 = C_0((\alpha, \beta) = \{\alpha\beta\})$. Note how we have included the group $\alpha\beta$ to our vocabulary set, and we have created a new corpus where the ordered pair (α, β) is now a single element $\alpha\beta$.

3. We identify the ordered pair of elements of $\mathcal{V}_1$, (α', β'), that occurs most frequently in the new corpus C_1, and we define $\mathcal{V}_2 = \mathcal{V}_1 \cup \{\alpha'\beta'\}$, $C_2 = C_1((\alpha', \beta') = \{\alpha'\beta'\})$.
4. We iterate this process, producing $\mathcal{V}_n, C_n$ from $\mathcal{V}_{n-1}, C_{n-1}$, with $|\mathcal{V}_n| = |\mathcal{V}_{n-1}| + 1$.
5. We terminate the process at C_m and $\mathcal{V} = \mathcal{V}_m$, with $|\mathcal{V}| = n_{\text{voc}}$, where n_{voc} is a prefixed integer.

Remark 3.1. *We denote the set of tokens with $\{\alpha_i\}_{i=1}^{n_{\text{voc}}}$. These represent the patterns of characters that occur most frequently in the given corpus of text. Therefore, tokens can be single characters, segments of words, full words, or, in some cases, complete phrases. For simplicity, in this work we will usually assume that tokens represent full words, but it is important to bear in mind that the notion of token extends beyond single words.*

Example 3.2. *Let us consider the following alphabet $\mathcal{A} = \{a, b, c\}$, and let us suppose we are given a corpus $C_0 = abacabbc$. The ordered pair of characters that occurs more frequently is (a, b), which we can find twice in the corpus C_0. Thus, we define $\mathcal{V}_1 = \{a, b, c, ab\}$, and we create a new corpus where ab is now a single element, which we will denote with $\overline{ab}$. With this, we have that the new corpus is $C_1 = \overline{ab}ac\overline{ab}bc$. Now, in C_1 all pairs of elements of $\mathcal{V}_1$ occur just once. In this case, we can take the pair that occurs first, $(\overline{ab}, a)$, and define $\mathcal{V}_2 = \{a, b, c, ab, aba\}$. We repeat these steps, updating the vocabulary and the corpus at each iteration, until we reach a vocabulary of a predetermined size, n_{voc}.*

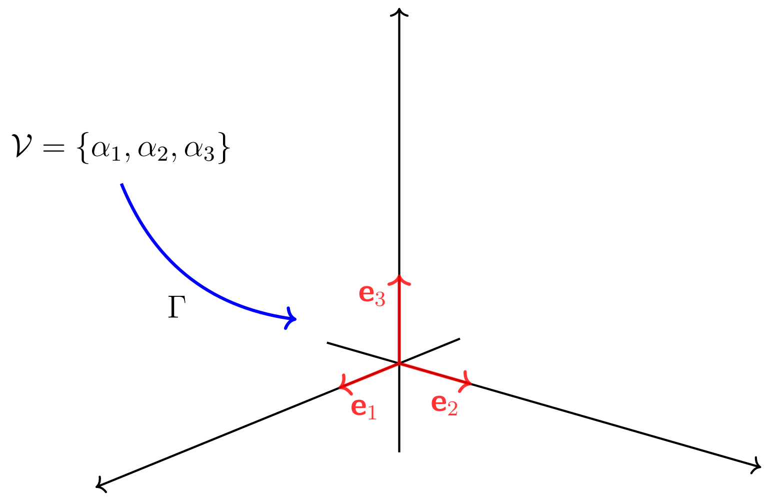
The next step is identifying tokens with vectors in a Euclidean space. One way to do this is by fixing a one-to-one mapping Γ between the set of tokens, $\{\alpha_i\}_{i=1}^{n_{\text{voc}}}$, and the standard orthonormal basis of $\mathbb{R}^{n_{\text{voc}}}$, $\{\mathbf{e}_j\}_{j=1}^{n_{\text{voc}}}$.

Note that there are $n_{\text{voc}}!$ possible mappings, but for simplicity we will consider the mapping:

$$\Gamma(\alpha_i) = \mathbf{e}_i, \quad \forall\, i = 1, \dots, n_{\text{voc}}.$$

This mapping is called one-hot encoding, and $\{\mathbf{e}_i\}_{i=1}^{n_{\text{voc}}}$ is the set of one-hot vectors. The term "one-hot" is used to indicate that these vectors only have a '1' in the component corresponding to the index of the token within the vocabulary, while the rest of the components are '0'.

One-hot vectors $(n_{\mathsf{voc}} = 3)$



One-hot encoding provides a straightforward method for obtaining vector representations of tokens, but it has its limitations. For one thing, one-hot vectors are very sparse in nature, which can lead to inefficient storage and computation, especially for large vocabularies. Furthermore, one-hot encoding lacks a built-in mechanism for capturing semantic similarity[2] between tokens, as each token is represented as an independent index within the vocabulary. To address these limitations, we turn to the concept of word embeddings.

Definition 3.3. *A word embedding is a projection matrix with trainable entries:*

$$W_E : \mathbb{R}^{n_{voc}} \longrightarrow \mathbb{R}^d$$

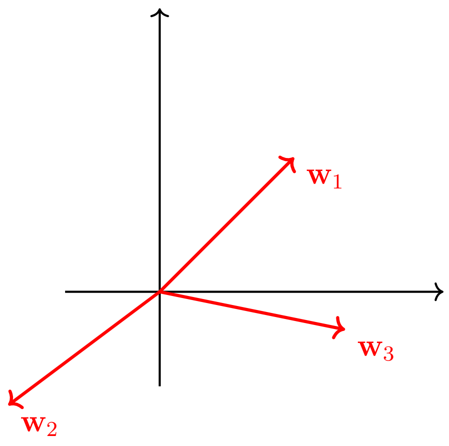
where $d < n_{voc}$ is called the dimension of the embedding.

Remark 3.4. *Notice how a matrix with trainable entries can be understood as a 1-depth neural network with no bias vector.*

[2]Semantic similarity refers to the degree to which two tokens can convey similar meanings in a given linguistic context.

Definition 3.5. *We refer to* $\{\mathbf{w}_i\}_{i=1}^{n_{\text{voc}}}$*, where* $\mathbf{w}_i = \mathbf{e}_i W_E$*, as the set of embedding vectors. Note that embedding vectors are the rows of the embedding matrix* W_E*.*

Embedding vectors $(d = 2)$



Word embeddings reduce the dimensionality of one-hot vectors by projecting them into a lower-dimensional Euclidean space, $\mathbb{R}^d$. More importantly, word embeddings learn to perform these projections in such a way that semantically related tokens are brought "closer" together in $\mathbb{R}^d$. This notion of "closeness" refers to the geometrical proximity between embedding vectors in $\mathbb{R}^d$, typically defined in this context by means of the dot product. By assigning close vectors to related tokens, we ensure that semantic similarity is rooted in the geometry of the new representation space.

There exist various techniques for training word embeddings, each with its own advantages and specific applications [Nas+20]. Two prominent examples of these techniques are Continuous-Bag-of-Words (CBOW) and the Skip-gram, both pertaining to the framework known as Word2Vec, proposed by Mikolov et al. in 2013 [Mik+13].

The Skip-gram model

The Skip-gram model is built upon the distributional[3] hypothesis, which states that tokens frequently co-occurring within a corpus of text tend to be semantically related. Leveraging this hypothesis, the model learns efficient word embeddings by predicting surrounding tokens given a target token within a corpus of text.

The Skip-gram model, although not a classification model in the traditional sense, shares a similar structure, since it involves predicting a set of surrounding tokens from a fixed vocabulary given a target token, which is analogous to predicting classes given an input in classification models. This similarity also extends to the transformer model, where we try to predict the next token for a given sequence. Therefore, it is beneficial to briefly discuss the general procedure in classification models.

In a classification setting, we have a set with N data points, $\{\mathbf{x}_i\}_{i=1}^N$ each belonging to one of k classes. In the notation of the previous section, the label space in this setting is a discrete set $\mathcal{Y} = \{y^{(1)}, \dots, y^{(k)}\}$. For each input example $\mathbf{x}_i$, a classification model typically outputs a k-dimensional vector of the so-called *logits*:

$$\mathbf{z}_i = \begin{pmatrix} z_i^{(1)} & \dots & z_i^{(k)} \end{pmatrix}.$$

Here, $z_i^{(j)}$ is called the logit corresponding to the j-th class for the i-th data point. These logits can be interpreted as scores assigned by the model, representing how "confident" it is that the i-th data point actually belongs to the j-th class. However, logits can be negative or arbitrarily large, which poses issues in terms of interpretability of the model.

Definition 3.6. *Given* $\mathbf{v} = \begin{pmatrix} v^{(1)} & \dots & v^{(k)} \end{pmatrix} \in \mathbb{R}^k$*, we define the softmax function as:*

$$\textit{softmax}(\mathbf{v}) = \frac{1}{\sum_{l=1}^k e^{v^{(l)}}} \begin{pmatrix} e^{v^{(1)}} & \dots & e^{v^{(k)}} \end{pmatrix}.$$

Note that the output of the softmax is a k-dimensional vector whose components are between 0 and 1, and they all sum up to 1. Therefore, the output of the softmax is a valid probability distribution.

[3]Here, the term "distributional" refers to the allocation of tokens within a corpus of text, rather than to probability distributions.

The vector of logits is passed through a softmax function, obtaining a probability distribution over the k classes for the i-th data point:

$$\text{softmax}(\mathbf{z}_i) = \frac{1}{\sum_{l=1}^{k} e^{z_i^{(l)}}} \begin{pmatrix} e^{z_i^{(1)}} & \dots & e^{z_i^{(k)}} \end{pmatrix} := \begin{pmatrix} p(y^{(1)} \mid \mathbf{x}_i) & \dots & p(y^{(k)} \mid \mathbf{x}_i) \end{pmatrix}.$$

Finally, in order to predict the class of $\mathbf{x}_i$, the classification model chooses the class with the highest probability:

$$\text{prediction for } \mathbf{x}_i = \arg\max_j p(y^{(j)} \mid \mathbf{x}_i).$$

These probabilities $p(y^{(j)} \mid \mathbf{x}_i)$ actually depend on the parameters of the model, $\theta \in \Theta$. We will denote this as usual by writing $p_\theta(y^{(j)} \mid \mathbf{x}_i)$. To train the model, we want to maximize the probability of the actual class of $\mathbf{x}_i$, which we will denote with y_i. Therefore, we choose the loss function for a single data point $\mathbf{x}_i$ to be:

$$l_i(\theta) = -\log p_\theta(y_i \mid \mathbf{x}_i).$$

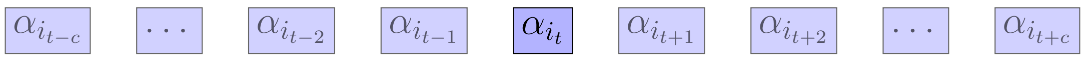
This loss function, known as cross entropy, heavily penalizes the model when the probability assigned to the actual class is close to 0, since the $\log$ makes the expression go to $+\infty$.

Now, considering all the data points $\{\mathbf{x}_i\}_{i=1}^N$, the loss function of the model is:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} l_i(\theta) = -\sum_{i=1}^{N} \log p_\theta(y_i \mid \mathbf{x}_i).$$

This loss function is then minimized using the GD algorithm, obtaining the optimal parameters for classification.

Going back to the Skip-gram model, let us consider a corpus of text $C = (\alpha_{i_t})_{t=1}^T$. For each possible target token α_{i_t} in the corpus , we want to predict its surrounding tokens within a context window of size $2c$, for some $c \in \mathbb{N}$.



The Skip-gram model computes logits for each token in the vocabulary relative to the given target token using the dot product of their embedding vectors:

$$z_t^{(j)} = \langle \mathbf{w}_j, \mathbf{w}_{i_t} \rangle, \quad j = 1, \dots, n_{\mathsf{voc}}.$$

Using the dot product of the embeddings to compute the logits aligns with the distributional hypothesis, assigning higher probability of occurrence to those tokens that are semantically related to the target token.

With this, for each target token we obtain a probability distribution over the vocabulary:

$$p_\theta(\alpha_j \mid \alpha_{i_t}) = \frac{e^{\langle \mathbf{w}_j, \mathbf{w}_{i_t} \rangle}}{\sum_{l=1}^{n_{\mathsf{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle}}, \quad j = 1, \dots, n_{\mathsf{voc}}$$

and the predicted surrounding tokens will be the $2c$ tokens with the highest probabilities.

We are interested in maximizing the probability of predicting all the actual neighbours, and therefore we must consider their joint probability. Assuming that the probabilities are conditionally independent, we have:

$$p_\theta(\alpha_{i_{t-c}}, \dots, \alpha_{i_{t-1}}, \alpha_{i_{t+1}}, \dots, \alpha_{i_{t+c}} \mid \alpha_{i_t}) = \prod_{\substack{j=-c \\ j \neq 0}}^{c} p_\theta(\alpha_{i_{t+j}} \mid \alpha_{i_t}).$$

The loss function for a single target token is then:

$$l_t(\theta) = -\log p_\theta(\alpha_{i_{t-c}}, \dots, \alpha_{i_{t-1}}, \alpha_{i_{t+1}}, \dots, \alpha_{i_{t+c}} \mid \alpha_{i_t}) = -\sum_{\substack{j=-c \\ j \neq 0}}^{c} \log p_\theta(\alpha_{i_{t+j}} \mid \alpha_{i_t})$$

$$= -\sum_{\substack{j=-c \\ j \neq 0}}^{c} \left[\langle \mathbf{w}_{i_{t+j}}, \mathbf{w}_{i_t} \rangle - \underbrace{\log \sum_{l=1}^{n_{\mathsf{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle}}_{\text{no dependence on } j} \right] = 2c \log \sum_{l=1}^{n_{\mathsf{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle} - \sum_{\substack{j=-c \\ j \neq 0}}^{c} \langle \mathbf{w}_{i_{t+j}}, \mathbf{w}_{i_t} \rangle.$$

Therefore, summing over all possible target tokens, we have:

$$\mathcal{L}(\theta) = \sum_{t=1}^{T} l_t(\theta) = \sum_{t=1}^{T} \left[2c \log \sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle} - \sum_{\substack{j=-c \\ j \neq 0}}^{c} \langle \mathbf{w}_{i_{t+j}}, \mathbf{w}_{i_t} \rangle \right].$$

Next, we present the computation of $\nabla \mathcal{L}(\theta)$ performed independently by the author. The partial derivative of $\mathcal{L}(\theta)$ with respect to one component $w_p^{(k)}$ of an arbitrary embedding vector $\mathbf{w}_p$ is given by:

$$\frac{\partial \mathcal{L}}{\partial w_p^{(k)}} = \sum_{t=1}^{T} \left[\underbrace{2c \frac{\partial}{\partial w_p^{(k)}} \log \sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle}}_{\mathbf{I}} - \sum_{\substack{j=-c \\ j \neq 0}}^{c} \underbrace{\frac{\partial}{\partial w_p^{(k)}} \langle \mathbf{w}_{i_{t+j}}, \mathbf{w}_{i_t} \rangle}_{\mathbf{II}} \right].$$

Computing **I**, we obtain:

$$\begin{aligned}
\frac{\partial}{\partial w_p^{(k)}} \log \sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle} &= \frac{1}{\sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle}} \sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle} \frac{\partial}{\partial w_p^{(k)}} \langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle \\
&= \frac{1}{\sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle}} \sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle} \left(\delta_{p,l} w_{i_t}^{(k)} + \delta_{p,i_t} w_l^{(k)} \right) \\
&= \frac{1}{\sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_{i_t} \rangle}} \left(e^{\langle \mathbf{w}_p, \mathbf{w}_{i_t} \rangle} w_{i_t}^{(k)} + \sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{w}_p \rangle} w_l^{(k)} \right)
\end{aligned}$$

where δ_{ij} denotes Kronecker's delta.

Computing **II**, we obtain:

$$\frac{\partial}{\partial w_p^{(k)}} \langle \mathbf{w}_{i_{t+j}}, \mathbf{w}_{i_t} \rangle = \delta_{p,i_{t+j}} w_{i_t}^{(k)} + \delta_{p,i_t} w_{i_{t+j}}^{(k)}.$$

Substituting **I** and **II** into the expression for $\frac{\partial \mathcal{L}}{\partial w_p^{(k)}}$, we obtain an explicit form for the partial derivatives of the loss function in the Skip-gram model. This allows us to apply the GD algoritm to optimize the components of the embedding vectors.

Now that we have seen how to train word embeddings, for any given snippet of text $s = (\alpha_{i_1}, \dots, \alpha_{i_N})$, $N \in \mathbb{N}$, we can use the trained word embedding W_E to represent s as a sequence of trained embedding vectors:

$$W(s) = (\mathbf{w}_{i_1}, \dots, \mathbf{w}_{i_N})$$

where $W(s)$ is called the semantic representation of the text snippet s. It is common to write $W(s)$ in matrix form:

$$W(s) = \begin{pmatrix} \leftarrow & \mathbf{w}_{i_1} & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & \mathbf{w}_{i_N} & \rightarrow \end{pmatrix}.$$

3.2 Positional encoding

Transformers will offer a key feature known as parallel processing. Parallelization is a paradigm in computer science that involves performing operations simultaneously on a set of elements. In our context, this means that the embedding vectors $\mathbf{w}_{i_k}$ corresponding to the rows of $W(s)$ will not be processed by the transformer one by one, but rather all at once. This approach provides a significant speed-up in both output generation and training time compared to preceding NLP architectures [SHT24]. However, because of this non-sequential processing, the transformer has no reference of the position of the tokens within the sequence. This suggests that positional information must be added to the semantic representation $W(s)$.

In order to do this, we take the embedding vector $\mathbf{w}_{i_k}$ in the sequence, and we add a d-dimensional real-valued vector $\mathbf{p}_{i_k}$. This vector $\mathbf{p}_{i_k}$ is called the positional vector of α_{i_k}, and it encodes positional information of α_{i_k}. By adding this positional vector to the embedding vector, we obtain a new representation:

$$\mathbf{u}_{i_k} = \mathbf{w}_{i_k} + \mathbf{p}_{i_k}$$

which now encodes semantic and positional information of α_{i_k}. In matrix form, this corresponds to adding a positional matrix $P(s)$ to the embedding representation $W(s)$, obtaining a new representation:

$$U(s) = W(s) + P(s) = \begin{pmatrix} \leftarrow & \mathbf{w}_{i_1} & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & \mathbf{w}_{i_N} & \rightarrow \end{pmatrix} + \begin{pmatrix} \leftarrow & \mathbf{p}_{i_1} & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & \mathbf{p}_{i_N} & \rightarrow \end{pmatrix} = \begin{pmatrix} \leftarrow & \mathbf{u}_{i_1} & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & \mathbf{u}_{i_N} & \rightarrow \end{pmatrix}.$$

The matrix $U(s)$ is called the semantic-positional representation of s, and $\mathbf{u}_{i_k}$ is called the semantic-positional vector of α_{i_k}.

A first approach would be to add the absolute position of each token within the sequence to the embedding representation. Namely, the matrix $P(s)$ would be:

$$P(s) = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 2 & 2 & \dots & 2 \\ \vdots & \vdots & \ddots & \vdots \\ N & N & \dots & N \end{pmatrix}.$$

However, for long sequences with $N \gg 1$, the positional vectors for the last tokens in the sequence would have very large components compared to the corresponding embedding vectors. This imbalance can cause semantic information to become less relevant or even neglegible for the model. Therefore, one desired property of positional vectors is that they are properly scaled.

To this end, we can normalize the elements of $P(s)$ by dividing all the components by N:

$$P(s) = \begin{pmatrix} 1/N & 1/N & \dots & 1/N \\ 2/N & 2/N & \dots & 2/N \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{pmatrix}.$$

This would certainly solve the scaling problem, but it raises another issue. Note that, for a sequence of 3 tokens, that is $N = 3$, the positional information added to the second token would be $p_2 = 2/3$. Now, for a sequence of 6 tokens, $N = 6$, the positional information added to the second token would be $p'_2 = 2/6 = 1/3$, and the positional information added to the fourth token would be $p'_4 = 4/6 = 2/3 = p_2$. With this example, we can see that this approach leads to a positional encoding which is not position-unique. This can cause problems if we want our model to handle sequences of variable length.

Sinusoidal positional encoding

Let P_{ij} denote the components of the positional matrix. Sinusoidal positional encoding is defined as:

$$\begin{cases} P_{i,2j} = \sin\left(\omega_j \cdot i\right) \\ P_{i,2j+1} = \cos\left(\omega_j \cdot i\right) \end{cases}$$

where $\omega_j = \frac{1}{M^{2j/d}}$, for M a fixed integer[4].

For an even dimension of the embedding, i.e. $d = 2m$ for some $m \in \mathbb{N}$, the sinusoidal positional matrix reads:

$$P(s) = \begin{pmatrix} \sin(1) & \cos(1) & \sin\left(\frac{1}{M^{2/d}}\right) & \cos\left(\frac{1}{M^{2/d}}\right) & \sin\left(\frac{1}{M^{4/d}}\right) & \dots & \sin\left(\frac{1}{M^{(m-1)/d}}\right) \\ \sin(2) & \cos(2) & \sin\left(\frac{2}{M^{2/d}}\right) & \cos\left(\frac{2}{M^{2/d}}\right) & \sin\left(\frac{2}{M^{4/d}}\right) & \dots & \sin\left(\frac{2}{M^{(m-1)/d}}\right) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \sin(N) & \cos(N) & \sin\left(\frac{N}{M^{2/d}}\right) & \cos\left(\frac{N}{M^{2/d}}\right) & \sin\left(\frac{N}{M^{4/d}}\right) & \dots & \sin\left(\frac{N}{M^{(m-1)/d}}\right) \end{pmatrix}.$$

Note how the sine and cosine functions keep the components bounded to ± 1, and we can also observe that each position has a unique positional vector. However, the most interesting property of sinusoidal positional encoding is that it allows the transformer to understand *relative* positions of tokens. This means that the transformer will be able to infer the relative distance between two tokens without having to explicitly learn their absolute positions within the sequence.

[4]The value of M is determined empirically, based on model performance. In [Vas+23], the value used is $M = 10000$.

Proposition 3.7. *For any pair of tokens whose absolute position differs an offset $l \in \mathbb{N}$, there exists a linear transformation T such that:*

$$\mathbf{p}_{i_{k+l}} = T\,\mathbf{p}_{i_k}$$

where $T = T(l)$ only depends on the offset l.

Proof. The sinusoidal positional vector $\mathbf{p}_{i_k}$ for a given token of the sequence α_{i_k} reads:

$$\mathbf{p}_{i_k} = \left(\sin(k) \quad \cos(k) \quad \dots \quad \sin\left(\tfrac{k}{M^{2j/d}}\right) \quad \cos\left(\tfrac{k}{M^{2j/d}}\right) \quad \dots \quad \sin\left(\tfrac{k}{M^{(m-1)/d}}\right)\right).$$

Now, for a fixed offset $l \in \mathbb{N}$, the positional vector of $\alpha_{i_{k+l}}$ reads:

$$\mathbf{p}_{i_{k+l}} = \left(\sin(k+l) \quad \cos(k+l) \quad \dots \quad \sin\left(\tfrac{k+l}{M^{2j/d}}\right) \quad \cos\left(\tfrac{k+l}{M^{2j/d}}\right) \quad \dots \quad \sin\left(\tfrac{k+l}{M^{(m-1)/d}}\right)\right).$$

For any j, we have that:

$$\sin\left(\frac{k+l}{M^{2j/d}}\right) = \sin\left(\frac{k}{M^{2j/d}} + \frac{l}{M^{2j/d}}\right).$$

Using the angle addition formula for the sine:

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta).$$

We can rexpress this as:

$$\sin\left(\frac{k+l}{M^{2j/d}}\right) = \sin\left(\frac{k}{M^{2j/d}}\right)\cos\left(\frac{l}{M^{2j/d}}\right) + \cos\left(\frac{k}{M^{2j/d}}\right)\sin\left(\frac{l}{M^{2j/d}}\right).$$

Similarly, we can use the addition formula for the cosine, and we obtain:

$$\cos\left(\frac{k+l}{M^{2j/d}}\right) = \cos\left(\frac{k}{M^{2j/d}}\right)\cos\left(\frac{l}{M^{2j/d}}\right) - \sin\left(\frac{k}{M^{2j/d}}\right)\sin\left(\frac{l}{M^{2j/d}}\right).$$

Therefore, for any j, we can write:

$$\begin{pmatrix}\sin\left(\frac{k+l}{M^{2j/d}}\right) & \cos\left(\frac{k+l}{M^{2j/d}}\right)\end{pmatrix} = \begin{pmatrix}\sin\left(\frac{k}{M^{2j/d}}\right) & \cos\left(\frac{k}{M^{2j/d}}\right)\end{pmatrix} \underbrace{\begin{pmatrix}\cos\left(\frac{l}{M^{2j/d}}\right) & -\sin\left(\frac{l}{M^{2j/d}}\right) \\ \sin\left(\frac{l}{M^{2j/d}}\right) & \cos\left(\frac{l}{M^{2j/d}}\right)\end{pmatrix}}_{\mathcal{R}_j}.$$

where notice that $\mathcal{R}_j = \mathcal{R}_j(l)$ is a rotation matrix of an angle $\phi(l,j) = \frac{l}{M^{2j/d}}$.

Therefore, we have that $\mathbf{p}_{i_{k+l}} = \mathbf{p}_{i_k} T(l)$ where:

$$T(l) = \left(\begin{array}{c|c|c|c|c|c} \mathcal{R}_0 & 0 & \dots & 0 & \dots & 0 \\ \hline 0 & \mathcal{R}_1 & \dots & 0 & \dots & 0 \\ \hline \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \hline 0 & 0 & \dots & \mathcal{R}_j & \dots & 0 \\ \hline \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \hline 0 & 0 & \dots & 0 & \dots & \mathcal{R}_{m-1} \end{array}\right).$$

$\square$

The existence of a linear relationship $T(l)$ between positional vectors of any two tokens separated by an offset l will enable the transformer to infer the relative position of two tokens based on the linear mapping that relates their positional vectors, without having to explicitly learn their absolute positions.

4 Transformer architecture

Word embeddings and positional encoding are often viewed as preprocessing components within the transformer architecture. In this section, we will delve into the main components of the transformer, with a particular focus on attention mechanisms, which are its most distinctive and powerful features.

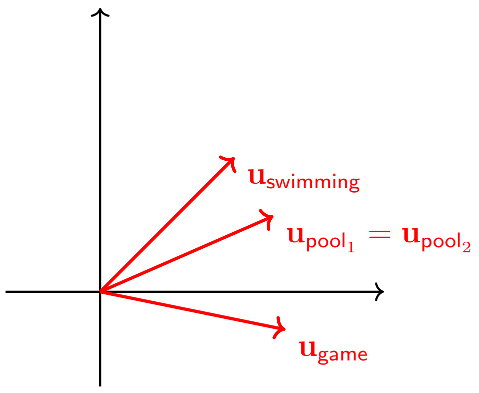
4.1 Attention mechanisms

Attention mechanisms enhance the model's ability to understand context-specific relationships within the input text by refining the generic semantic-positional representations. This allows the transformer to better grasp the precise meanings of tokens in certain contexts. As a motivating example, let us consider the following sentences:

He quickly dived into the swimming pool.
He brilliantly won a game of pool.

Because of how word embeddings are constructed, they do not learn different embedding vectors for each possible meaning of the token *pool*; instead, they learn a single generic embedding vector $\mathbf{w}_{\text{pool}}$ that somehow encapsulates all its possible meanings. Furthermore, both occurrences of *pool* appear in the seventh position of their respective sentences, which means that they also have the same positional vector $\mathbf{p}_{\text{pool}}$. Therefore, despite *pool* having entirely different meanings in each context, both instances share an identical semantic-positional vector:

$$\mathbf{u}_{\text{pool}_1} = \mathbf{u}_{\text{pool}_2} = \mathbf{w}_{\text{pool}} + \mathbf{p}_{\text{pool}}.$$


$\mathbf{u}_{\text{swimming}}$
$\mathbf{u}_{\text{pool}_1} = \mathbf{u}_{\text{pool}_2}$
$\mathbf{u}_{\text{game}}$

To address this limitation, attention mechanisms update these generic representations by incorporating information from the rest of the tokens in the sequence, thereby providing each token with a new representation that is context-aware. From the various types of attention mechanisms [BF23], in this work we focus on dot-product attention [Vas+23].

Let us consider the semantic-positional representation of some snippet of text s:

$$U(s) = \begin{pmatrix} \leftarrow & \mathbf{u}_{i_1} & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & \mathbf{u}_{i_N} & \rightarrow \end{pmatrix}.$$

Definition 4.1. *For a semantic-positional vector $\mathbf{u}_{i_k}$ in $U(s)$, we define its associated value vector as:*

$$\mathbf{v}_{i_k} = \mathbf{u}_{i_k} W_V$$

where $W_V \in \mathcal{M}_{d\times d}$ is a matrix with trainable entries.

Intuitively, value vectors encode the information necessary for updating the representations of the rest of the tokens in the sequence. In the case of the introductory example, the updated representations of *pool* might resemble:

$$\mathbf{u}_{\text{pool}_1} \mapsto \mathbf{u}_{\text{pool}_1} + \mathbf{v}_{\text{swimming}} + \mathbf{v}_{\text{dived}} + \mathbf{v}_{\text{into}} + \dots$$
$$\mathbf{u}_{\text{pool}_2} \mapsto \mathbf{u}_{\text{pool}_2} + \mathbf{v}_{\text{game}} + \mathbf{v}_{\text{pool}} + \mathbf{v}_{\text{brilliantly}} + \dots$$

However, let us note that not all tokens should contribute equally to refining the meaning of a specific token. In particular, we naturally understand that tokens such as *swimming*, *dived*, *won* and *game* are significantly more relevant than the rest of tokens when determining the specific meaning of the token *pool*. This suggests that we require a mechanism for quantifying relative contextual relevance, so that we can update the representations accordingly. To quantify this contextual relevance, attention mechanisms use the concepts of queries, keys and attention scores.

Definition 4.2. *For a fixed integer $d^* < d$, let us consider two projection matrices with trainable parameters:*

$$W_Q : \mathbb{R}^d \longrightarrow \mathbb{R}^{d^*}$$
$$W_K : \mathbb{R}^d \longrightarrow \mathbb{R}^{d^*}$$

For a context-positional vector $\mathbf{u}_{i_k}$ in $U(s)$, we define its associated query and key vectors as:

$$\mathbf{q}_{i_k} = \mathbf{u}_{i_k}\, W_Q$$
$$\mathbf{k}_{i_k} = \mathbf{u}_{i_k}\, W_K$$

Query and key vectors play complementary roles in determining contextual relevance between tokens. We can think of query vectors as "questions" each token asks about the rest of the tokens in the sequence, representing what each token is "looking for" in others. Conversely, key vectors are like the "answers" each token provides when queried by another token, representing what each token can "offer" to the other tokens. Based on this intuition, the projection matrices W_Q and W_K are trained to generate representations in $\mathbb{R}^{d^*}$ that optimally encode these "questions" and "answers".

Remark 4.3. *Bear in mind that this interpretation is just a humanized oversimplification of how attention mechanisms work. In reality, the information encoded in query and key vectors is much more abstract and complex, capturing deep dependencies within the sequence that go far beyond simple question-and-answer analogies.*

Building on this intuition, we define the attention score between two tokens as the measure to how well the questions posed by the query align with the answers provided by the key. Mathematically, this is captured by the dot product in $\mathbb{R}^{d^*}$ between the query vector and the key vector of two tokens:

$$\text{AttentionScore}(\alpha_{i_k}, \alpha_{i_j}) = \langle \mathbf{q}_{i_k}, \mathbf{k}_{i_j} \rangle.$$

This attention score quantifies how relevant the token α_{i_j} is for the token α_{i_k}.

Definition 4.4. *The* **attention pattern** *is defined as an $N \times N$ matrix:*

$$A(s) = \textit{softmax}\left[\begin{pmatrix} \langle \mathbf{q}_{i_1}, \mathbf{k}_{i_1} \rangle & \dots & \langle \mathbf{q}_{i_1}, \mathbf{k}_{i_N} \rangle \\ \langle \mathbf{q}_{i_2}, \mathbf{k}_{i_1} \rangle & \dots & \langle \mathbf{q}_{i_2}, \mathbf{k}_{i_N} \rangle \\ \vdots & \ddots & \vdots \\ \langle \mathbf{q}_{i_N}, \mathbf{k}_{i_1} \rangle & \dots & \langle \mathbf{q}_{i_N}, \mathbf{k}_{i_N} \rangle \end{pmatrix}\right] := \begin{pmatrix} a_{1,1} & \dots & a_{1,N} \\ a_{2,1} & \dots & a_{2,N} \\ \vdots & \ddots & \vdots \\ a_{N,1} & \dots & a_{N,N} \end{pmatrix}$$

where the softmax function is applied row-wise.

The softmax function normalizes the attention scores for a fixed query $\mathbf{q}_{i_k}$ across all possible keys, thereby providing the token α_{i_k} with a set of attention weights $\{a_{k,l}\}_{l=1}^{N}$ that represent the *relative* contextual importance of all tokens in the sequence with respect to α_{i_k}.

Remark 4.5 (Attention pattern as a bilinear form)**.** *Alternatively, if we define $W_{QK} = W_Q(W_K)^T$ as the query-key matrix, the attention pattern $A(s)$ can also be seen as the result of applying a non-symmetric bilinear form $\langle \mathbf{v}, \mathbf{w} \rangle_{QK} := \mathbf{v} W_{QK} \mathbf{w}$ to each pair of rows in $U(s)$ and then followed by the softmax function applied row-wise:*

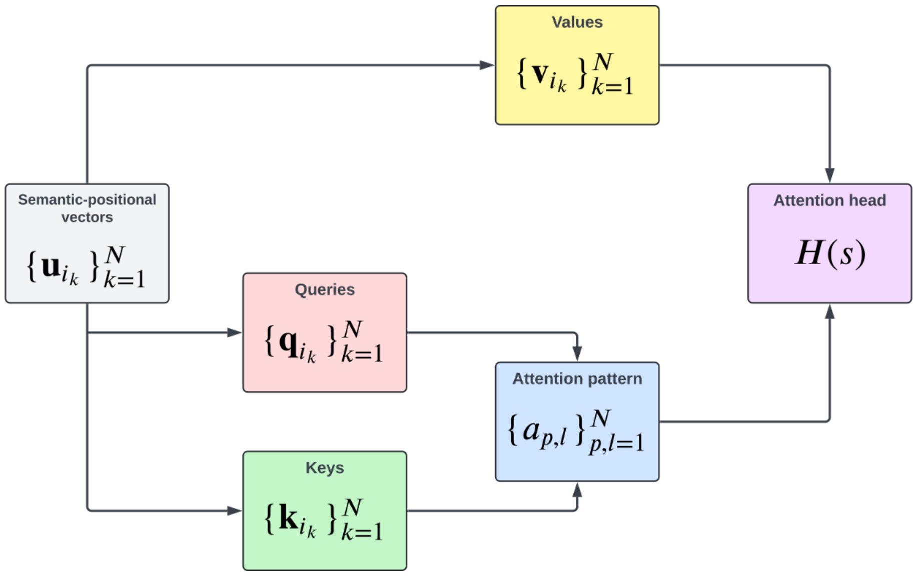
$$A(s) = \textit{softmax}\left[\begin{pmatrix} \langle \mathbf{u}_{i_1}, \mathbf{u}_{i_1} \rangle_{QK} & \dots & \langle \mathbf{u}_{i_1}, \mathbf{u}_{i_N} \rangle_{QK} \\ \vdots & \ddots & \vdots \\ \langle \mathbf{u}_{i_N}, \mathbf{u}_{i_1} \rangle_{QK} & \dots & \langle \mathbf{u}_{i_N}, \mathbf{u}_{i_N} \rangle_{QK} \end{pmatrix}\right].$$

Finally, for each token in the sequence we can compute weighted sums of the value vectors, where the attention weights ensure that the contribution of each value vector is based on the corresponding contextual relevance.

We collect all these weighted sums in an $N \times d$ matrix called the attention head:

$$H(s) = \begin{pmatrix} \sum_{l=1}^{N} a_{1,l}\mathbf{v}_{i_l} \\ \vdots \\ \sum_{l=1}^{N} a_{N,l}\mathbf{v}_{i_l} \end{pmatrix}$$

where the k-th row of $H(s)$ represents the contextual information that will be used to refine the representation of α_{i_k}.

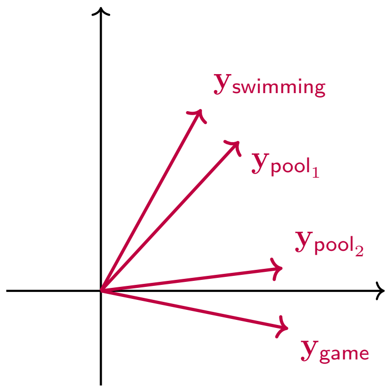

Values
$\{\mathbf{v}_{i_k}\}_{k=1}^{N}$
Semantic-positional vectors
$\{\mathbf{u}_{i_k}\}_{k=1}^{N}$
Attention head
$H(s)$
Queries
$\{\mathbf{q}_{i_k}\}_{k=1}^{N}$
Attention pattern
$\{a_{p,l}\}_{p,l=1}^{N}$
Keys
$\{\mathbf{k}_{i_k}\}_{k=1}^{N}$

By adding the attention head $H(s)$ to the semantic-positional representation $U(s)$, we obtain a context-aware representation of the text snippet s:

$$Y(s) = U(s) + H(s) = \begin{pmatrix} \leftarrow & \mathbf{y}_{i_1} & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & \mathbf{y}_{i_N} & \rightarrow \end{pmatrix}$$

where $\mathbf{y}_{i_k} = \mathbf{u}_{i_k} + \sum_{l=1}^{N} \alpha_{k,l}\mathbf{v}_{i_l}$ is called the context-aware vector of α_{i_k}.

Going back to the introductory example, the context-aware representations of *pool*$_1$ and *pool*$_2$ allow the model to understand their specific meaning in their respective contexts.


$\mathbf{y}_{\text{swimming}}$
$\mathbf{y}_{\text{pool}_1}$
$\mathbf{y}_{\text{pool}_2}$
$\mathbf{y}_{\text{game}}$

4.2 Position-wise neural networks and normalization

Even though context-aware representations are already very informative, they might not sufficiently capture higher-level abstractions required for complex tasks. Position-wise neural networks introduce non-linearity to the model, enhancing the transformer's ability to capture complex patterns and dependencies in the sequence.

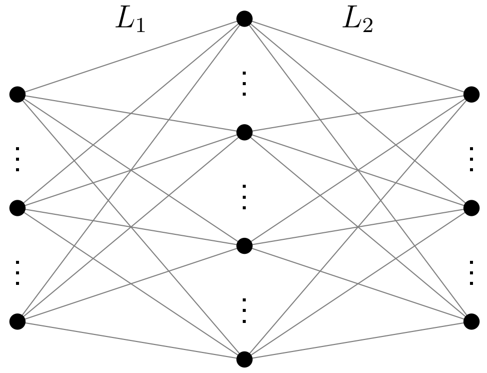
The term "position-wise" refers to the fact that a neural network is applied identically and independently to each position in the sequence. This neural network consists of two layers:

$$\mathcal{F} : \mathbb{R}^d \xrightarrow{L_1} \mathbb{R}^{d_F} \xrightarrow{L_2} \mathbb{R}^d, \quad d_F > d.$$

Its explicit expression is given by:

$$\mathcal{F}(\mathbf{x}) = \max\left(0, \mathbf{x}W_1 + \mathbf{b}_1\right) W_2 + \mathbf{b}_2$$

with $W_1 \in \mathcal{M}_{d\times d_F}$, $W_2 \in \mathcal{M}_{d_F\times d}$, $\mathbf{b}_1 \in \mathbb{R}^{d_F}$ and $\mathbf{b}_2 \in \mathbb{R}^{d}$.



Applying the position-wise neural network to the context-aware representation gives a new representation:

$$\tilde{Z}(s) = \begin{pmatrix} \leftarrow & \mathcal{F}(\mathbf{y}_{i_1}) & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & \mathcal{F}(\mathbf{y}_{i_N}) & \rightarrow \end{pmatrix} := \begin{pmatrix} \leftarrow & \tilde{\mathbf{z}}_{i_1} & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & \tilde{\mathbf{z}}_{i_N} & \rightarrow \end{pmatrix}.$$

These vectors $\tilde{\mathbf{z}}_{i_k}$ are called enhanced vectors, and they are richer representations of the context-aware vectors $\mathbf{y}_{i_k}$, capturing higher-level abstractions and dependencies within the sequence. Typically, these vectors undergo a normalization process before generating the output distribution.

Normalization

Including normalization steps in DL models offers significant advantages in speed and performance by stabilizing the learning process [BKH16]:

$$\mathbf{z}_{i_k} = \frac{\tilde{\mathbf{z}}_{i_k} - \boldsymbol{\mu}_{i_k}}{\sigma_{i_k} + \epsilon}, \quad \boldsymbol{\mu}_{i_k} = (\mu_{i_k}, \dots, \mu_{i_k})$$

where $\mu_{i_k} = \frac{1}{d}\sum_{j=1}^{d} \tilde{z}_{i_k}^{(j)}$, $\sigma_{i_k}^2 = \frac{1}{d}\sum_{j=1}^{d}(\tilde{z}_{i_k}^{(j)} - \mu_{i_k})^2$, and $\epsilon > 0$ is an added constant to avoid division by 0. Here, the term σ_{i_k} refers to the standard deviation of the components of $\tilde{\mathbf{z}}_{i_k}$, not to be confused with the activation function from previous sections.

With this, we can finally define the normalized enhanced representation of the text snippet s as:

$$Z(s) = \begin{pmatrix} \leftarrow & \mathbf{z}_{i_1} & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & \mathbf{z}_{i_N} & \rightarrow \end{pmatrix}.$$

This will be the vectors used during the output generation and training parts of the transformer, as we will see in the upcoming section.

4.3 Output distribution and training

Similarly to the Skip-gram model, the transformer outputs a n_{voc}-dimensional vector of logits, which is then passed through a softmax function to obtain a probability distribution. Given a snippet of text $s = (\alpha_{i_1}, \dots, \alpha_{i_N})$, the probability of $\alpha_j \in \mathcal{V}$ being the next token in the sequence is given by:

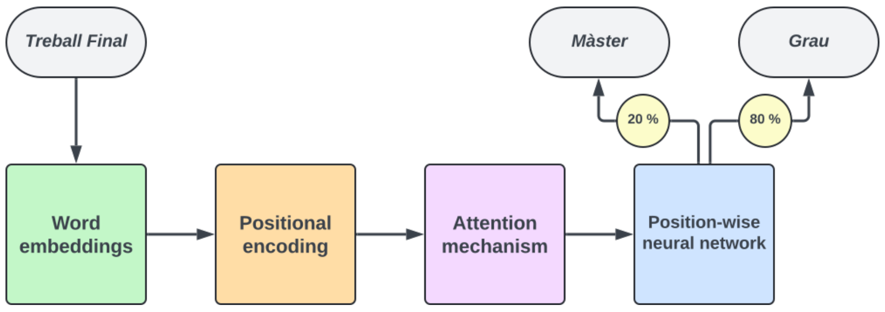
$$p_\theta(\alpha_j \mid s) = \frac{e^{\langle \mathbf{w}_j, \mathbf{z}_{i_N} \rangle}}{\sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{z}_{i_N} \rangle}}.$$

Note how this distribution is very similar to the one discussed in the Skip-gram model. However, instead of using the embedding vector of the last token in s, we are using its normalized enhanced vector to compute the logits. The intuition behind this is that the enhanced representation now incorporates contextual information from the previous tokens in the sequence and captures higher-level abstractions, enabling the model to make more informed and accurate probability assignments.

With this foundation, we can finally provide a mathematical definition of the transformer model.

Definition 4.6. *The transformer model is a family of parametrized functions* $\{\mathcal{T}_\theta : \mathbb{R}^{N\times d} \to \mathbb{R}^{n_{voc}} \mid \theta \in \Theta\}$ *such that, given a snippet of text* $s = (\alpha_{i_1}, \dots, \alpha_{i_N})$, $\mathcal{T}_\theta$ *outputs a probability distribution:*

$$\mathcal{T}_\theta(s) \sim p_\theta(\alpha_j \mid s) = \frac{e^{\langle \mathbf{w}_j, \mathbf{z}_{i_N} \rangle}}{\sum_{l=1}^{n_{voc}} e^{\langle \mathbf{w}_l, \mathbf{z}_{i_N} \rangle}}.$$



The parameters of the transformer model primarily include those of the attention mechanism and the position-wise neural network. Specifically, in the attention mechanism the parameters are the components of the projection matrices W_Q, W_K and the linear transformation W_V, corresponding to the queries, keys and values, respectively. On the other hand, the parameters of the position-wise neural network are the components of the weights matrices W_1, W_2 and the bias vectors $\mathbf{b}_1$ and $\mathbf{b}_2$. Typically, word embeddings are trained separately and independently from the transformer model, and thus they do not contribute to the set of parameters.

Year	Model	N° parameters
2018	GPT	110M
2018	BERT	340M
2019	GPT-2	1.5B
2020	GPT-3	175B
2022	PaLM	540B
2023	GPT-4	1.4T (?)

Table 1: Evolution of total number of parameters in LLMs. Source: [Dou23]

The rapid growth in the number of parameters in LLMs reflects the advancements in computational power and efficiency. More parameters generally translates to better understanding and accuracy of the model in predicting and generating text.

Training

In order to adjust the parameters of the transformer model, we use a set of training text snippets, $\{s_i\}_{i=1}^{N_S}$. For each training sample $s_i = (\alpha_{i_1}, \dots, \alpha_{i_N})$, the transformer computes the output probability distribution passing $\overline{s_i} = (\alpha_{i_1}, \dots, \alpha_{i_{N-1}})$ as input. The goal of the training process is to maximize the probability $p_\theta(\alpha_{i_N} \mid \overline{s_i})$. Therefore, similar to the Skip-gram model, the loss function for a single training sample is:

$$l_i(\theta) = -\log p_\theta(\alpha_{i_N} \mid \overline{s_i}) = -\langle \mathbf{w}_{i_N}, \mathbf{z}_{i_{N-1}} \rangle + \log \sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{z}_{i_{N-1}} \rangle}$$

and then the total loss function of the transformer is the sum of individual losses across all training samples:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N_S} l_i(\theta) = \sum_{i=1}^{N_S} \left(-\langle \mathbf{w}_{i_N}, \mathbf{z}_{i_{N-1}} \rangle + \log \sum_{l=1}^{n_{\text{voc}}} e^{\langle \mathbf{w}_l, \mathbf{z}_{i_{N-1}} \rangle} \right).$$

In this case, deriving a closed-form analytical expression for the gradient of the loss function is virtually impossible due to the complex structure of the transformer model. However, this underscores the critical role of the backpropagation algorithm, providing a highly efficient method for computing gradients in complex DL models like the transformer. By leveraging backpropagation, the transformer model adjusts its parameters using the GD algorithm, enabling to effectively improve its predictive capabilities.

5 Conclusions

In this work, we have defined transformers as families of parametrized functions used for generating probability distributions over a fixed vocabulary given a snippet of text. We have delved into their main components: word embeddings and positional encoding for generic text representations, attention mechanisms for capturing relevant contextual information, and position-wise neural networks for higher-level abstractions.

To answer the title of the thesis, there are three main mathematical frameworks we have used during this work. Linear algebra has played a critical role by performing linear and affine transformations on generic text representations, obtaining enhanced representations that factor in a broader range of natural language aspects. Calculus has also been essential for optimizing the model's parameters using the GD algorithm, ensuring the model improves its capabilities during training. Lastly, probability theory has been fundamental in generating output distributions over the vocabulary, which can then be used for sampling during text generation.

Some of the concepts described in this work have been oversimplified, prioritizing mathematically precise and accessible explanations over technical intricacies. Future work could extend on this introduction to transformers by exploring more advanced versions with richer mathematics.

In conclusion, a deep mathematical understanding of transformers is essential for developing and enhancing their capabilities. Today, many improvements introduced to these models are purely based on empirical observations, lacking a solid theoretical background supporting them. Therefore, future research should focus on establishing fundamental principles to better understand current models, facilitating the development of more efficient and accurate architectures in the field of NLP.

References

[Bas+22] Anil Bas et al. "A Brief History of Deep Learning-Based Text Generation". In: Dec. 2022, pp. 1–4. DOI: 10.1109/ICCA56443.2022.10039545.

[Bec23] S. Becker-Khan. "Notes on the mathematics of large transformer language model architecture". In: (2023).

[BF23] Gianni Brauwers and Flavius Frasincar. "A General Survey on Attention Mechanisms in Deep Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 35.4 (Apr. 2023), pp. 3279–3298. ISSN: 2326-3865. DOI: 10.1109/tkde.2021.3126456. URL: http://dx.doi.org/10.1109/TKDE.2021.3126456.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[DMC23] Saeed Damadi, Golnaz Moharrer, and Mostafa Cham. *The Backpropagation algorithm for a math student*. 2023. arXiv: 2301.09977 [cs.LG].

[Dou23] Michael R. Douglas. *Large Language Models*. 2023. arXiv: 2307.05782 [id='cs.CL'].

[Joh+21] Prashant Johri et al. "Natural Language Processing: History, Evolution, Application, and Future Work". In: Jan. 2021, pp. 365–375. ISBN: 978-981-15-9711-4. DOI: 10.1007/978-981-15-9712-1_31.

[Les+92] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *NYU Working Paper No. IS-92-13* (1992). URL: https://ssrn.com/abstract=1288490.

[Mik+13] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[Nas+20] Usman Naseem et al. *A Comprehensive Survey on Word Representation Models: From Classical to State-Of-The-Art Word Representation Language Models*. 2020. arXiv: 2010.15036 [cs.CL].

[NCK20] David Noever, Matt Ciolino, and Josh Kalin. *The Chess Transformer: Mastering Play using Generative Language Models*. 2020. arXiv: 2008.04057 [cs.CL].

[PH22] Mary Phuong and Marcus Hutter. *Formal Algorithms for Transformers*. 2022. arXiv: 2207.09238 [cs.LG].

[Sch19] Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*. 2019. arXiv: 1912.05911 [id='cs.LG'].

[SHT24] Clayton Sanford, Daniel Hsu, and Matus Telgarsky. *Transformers, parallel computation, and logarithmic depth*. 2024. arXiv: 2402.09268 [cs.LG].

[Udd+24] Mohammad Amaz Uddin et al. *ExplainableDetector: Exploring Transformer-based Language Modeling Approach for SMS Spam Detection with Explainability Analysis*. 2024. arXiv: 2405.08026 [cs.CL].

[Vas+23] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[WJ16] Shuohang Wang and Jing Jiang. *Learning Natural Language Inference with LSTM*. 2016. arXiv: 1512.08849 [id='cs.CL'].