ℝami Luisto

A study on the embedding spaces of the BERT language model

Master's Thesis in Information Technology

September 4, 2024

University of Jyväskylä

Faculty of Information Technology

Author: ℝami Luisto

Contact information: rami.luisto@gmail.com

Supervisors: Sami Äyrämö, and Ida Toivanen

Title: A study on the embedding spaces of the BERT language model

Työn nimi: BERT-kielimallin sisäisestä maailmankuvasta

Project: Master's Thesis

Study line: CS

Page count: 95+0

Abstract: This thesis considers a subfield of artificial intelligence called *Natural Language Processing* (NLP). More specifically we study a language model named *BERT* based on the so called *transformer* architecture, and the internal language representation of BERT called *embedding vectors*.

Keywords: NLP, AI, BERT, Pro gradu theses

Suomenkielinen tiivistelmä: Tässä työssä käsitellään *luonnollisen kielen käsittelynä* tunnettua tekoälyn osa-aluetta. Työssä keskitytään niin kutsuttuun *transformer*-arkkitehtuuriin pohjautuvaan *BERT*-nimiseen tekoälymalliin. Erityisesti työssä tarkastellaan tämän mallin *upotusvektoreita*, jotka kuvastavat mallin sisäistä luonnollisen tekstin esitysmuotoa.

Avainsanat: NLP, tekoäly, BERT, pro gradu -tutkielmat

Preface

Acknowledgements

I thank my advisors Ida Toivanen and Sami Äyrämö for their guidance in my work. With my background I have a few narrow scopes of very high expertise, but I often lack the wide intellectual basis on which to ground my research. With their knowledge Ida and Sami have provided me with the groundwork of expertise required for this thesis. They have furthermore helped me narrow down the scope of this work and trim my sometimes overly tangential expositions. This work and my understanding of the field have greatly benefitted from their help.

In alphabetical order: Pedram Daee, Razieh Ehsani, Jeong-Hyun Kim and Kseniia Palin. Thank you all for your mentoring, your guidance and for showing me how AI is done in the real world. Thanks also to my employer Digital Workforce Services for providing me a place to learn and apply AI to provide value to customers.

I thank my wife Anne for providing intelligent commentary and questions from outside of my technical bubble. Having someone to talk to who is intelligent, suspicious and not afraid to show it is invaluable for my intellectual happiness.

Finally, and most importanty, I dedicate this work to my daugher Aada. Many of the complex ideas here appeared during the many nights when I was holding your hand and easing you to sleep. These are not your dreams, but they are strange thoughts that appeared next to you when you were at the horn gates of dream. I give them to you now, and hope that many of your own will follow in the future.

Espoo, September 4, 2024

$\mathbb{R}$ami Luisto

A few words on the workflow of the writing of this thesis

We have approached many topics discussed here by making up hypotheses, running simulations, trying to understand what happens, and only afterwards checking up on the literature. This has been very fruitful, but might have resulted in an offbeat organization of topics at times. So far all the good ideas we've had and all the interesting observations we've made have then been found in the literature – except that with better insight, results and prestige. The few ideas we have here that we have not been able to find a pre-existing source should be seen as proof of the author's lack of familiarity with the literature and not as a claim of originality.

Disclosure on AI-generated content:

AI-tools have been used in producing this thesis. Their role has been akin to that of a proof-reader or a search engine and not that of a co-author. The scientific merits, if any, of this work rest wholly on the human author and not on AI. We wish, however, to point out exactly where generative AI has been used:

1. Essentially all of the Python code has been written in VSCode with the Github Copilot enabled. This has sped up the author by autocompleting trivial parts of some of the scripts. The main algorithm design and system planning has been done by the author.
2. chatGPT4 has been used to produce many first drafts of code related to API calls and in scraping e.g. the Wikipedia data.
3. On a few occasions, we've used chatGPT4 directly to generate collections of synonymous words or sentences.
4. chatGPT4 has also been used to produce first versions of various scripts that try to do complex linear algebra effectively. This has often resulted in hard-to-notice errors in the calculations and has been more detrimental than useful for this work. (Except as advanced exercises in debugging and frustration.)

Any and all text, code, and other material, AI-generated or otherwise, has been read and verified by the author. The author takes full responsibility of the whole content of this thesis.

Glossary

AI	Artificial Intelligence
BERT	Bidirectional Encoder Representations from Transformers
DL	Deep Learning
ELMo	Embeddings from Language Model
FC NN	Fully Connected Neural Network
FF NN	Feed-Forward Neural Network
GPT	Generative Pretrained Transformer
LLaMA	Large Language Model Meta AI
LLM	Large Language Model
LSTM	Long Short Term Memory
ML	Machine Learning
NLP	Natural Language Processing
PCA	Principal Component Analysis
PORO	A family of LLMs focused on European languages
RNN	Recurrent Neural Network
RoBERTa	A Robustly Optimized BERT
SLM	Small Language Model
TurkuNLP	The University of Turku NLP research group
TDA	Topological Data Analysis
ULMFiT	Universal Language Model Fine-Tuning for text Classification
UMAP	Uniform Manifold Approximation and Projection for dimension reduction
t-SNE	t-distributed Stochastic Neighbor Embedding

List of Figures

Figure 1. The Hopf link and its PCA projection. 11
Figure 2. The t-SNE and UMAP projections of the Hopf link with various parameters. ... 12
Figure 3. PCA-based dimension estimation. 14
Figure 4. Image of **the** original transformer architecture from Vaswani et al. 2017. Encoder on the left, decoder on the right. Used within permissions granted by the authors. 19
Figure 5. Illustration of a tranformer encoder. The "Segment Embeddings" in the figure are present in the BERT architecture but not in the original Transformer architecture. 20
Figure 6. Classical manually defined sinusoidial positional encodings and their pairwise cosine similarities. On the left side each row represents a different positional encoding vector of dimension 512. 29
Figure 7. The self attention mechanism illustrated. 33
Figure 8. Various types of attention patterns as described by (Kovaleva et al. 2019). Image copied from the source files of the article. 35
Figure 9. The BERT main architecture. 40
Figure 10. Distribution visualizations of the context-free embeddings and the positional embedding vectors of the BERT model. 48
Figure 11. Projection visualizations and dimensional analysis of the context-free embeddings and the positional embedding vectors of the BERT model. 50
Figure 12. Histograms on the distributions of the measures studied in Table 3. 54
Figure 13. BERTs positional encodings. 55
Figure 14. Distribution visualizations of the context-free embeddings and the positional embedding vectors of the BERT model with only the 384 positional encodings. Compare this to Figure 10. 56
Figure 15. Positional encoding vectors and their cosine similarities in the FinBERT model. 57
Figure 16. The cosine similarities of the first 10 non-[CLS] token positional embeddings against the next 128 positional encodings. 58
Figure 17. Cosine similarities of BERTs positional encodings, shifted to show relative properties. 58
Figure 18. The UMAP and t-SNE projections of the BERT positional embeddings, with the "bad" positional encodings from 384 onwards highlighted. 59
Figure 19. The helical structure of the first 384 positional embeddings of the BERT model, excluding position 0. PCA projection to 3 components. We do one 'round' once in about every 22 tokens. 60
Figure 20. PCA projection of the evolution of token-level embeddings of two disjoint sentences; "The mouse was gone. The cat sat useless." and "The mouse was gone. The computer sat useless.". Red dot marks the starting embedding. 64
Figure 21. PCA projection of the evolution of only one or two tokens. 64
Figure 22. The Euclidean distances and cosine similarities of consecutive embeddings of the tokens in "The mouse was gone. The cat sat useless.", and the norms of the embeddings at each step. 65

Figure 23. The PCA projection of the [CLS] embeddings of various synonymous sentences. 68
Figure 24. PCA projection of the differences in the embedding vectors. The points represent individual differences and the arrows the average difference vector of each type. 70
Figure 25. Three projections of Wikipedia and Twitter data. 72
Figure 26. Distribution visualizations of a sampling of random Wikipedia and Twitter texts. 73

List of Tables

Table 1. Table showing the predictions of two different model types trained on the context free embeddings to predict whether the token is an adjective, a noun or a verb. The biggest value of each row is boldened. Note that the word "bear" can be either a noun or a verb depending on the context. 28
Table 2. Table showing the cosine similarities of various context-free embeddings with special tokens excluded. 52
Table 3. Table showing some of the extremal token values compared to the average embedding. In these extremal values we've included only the tokens that contain only latin alphabet symbols and have length of at least 3. Largest and smallest value of each column has been highlighted. 53
Table 4. Table showing the four token-position pairs whose cosine similarity is outside of the range $[-0.2, 0.2]$. The final token is a punctuation mark in sanscrit called "*devanagari danda*" and it is nontrivial to display it LaTeXformat. 62
Table 5. Some statistics on the cosine similarities between the sentence type encodings and other pre-encoder embedding vectors. 62
Table 6. Sentence pairs for difference analysis. 69
Table 7. Table showing the cosine similarities of the average difference vectors. We've highlighted some of the more extreme values. 71

Contents

1 INTRODUCTION ... 1
1.1 Contents of this thesis ... 2
1.1.1 What we won't do in this thesis ... 2
1.1.2 What we'll do in this thesis ... 4
1.1.3 Structure of the thesis ... 4

2 PRELIMINARIES ... 6
2.1 Some notation and definitions ... 6
2.1.1 A few words on complex indexing ... 7
2.2 The maskless Shoggoth on the left – the incomprehensible internal world of language models ... 8
2.3 On undertanding and visualizing large dimensions ... 9
2.3.1 Projections to 2D ... 10
2.3.2 PCA-based dimensional analysis ... 13
2.4 On various other things related to this thesis ... 13
2.4.1 On preliminary requirements for the reader ... 14
2.4.2 What to read after this thesis? ... 15
2.4.3 Images and licenses ... 15

3 AN OVERVIEW OF THE TRANSFORMER ARCHITECTURE ... 16
3.1 The technological setting where transformers appeared ... 16
3.2 The architecture of the original transformer ... 18
3.2.1 A bird's eye view of the architecture ... 19
3.2.2 From words to indeces – tokenization in more detail ... 24
3.2.3 Turning "bear" into a concept – input embedding vectors in more detail 26
3.2.4 Not a bag of words – positional embeddings in detail ... 28
3.2.5 Where to look – the attention mechanism in more detail ... 29
3.2.6 Getting clever with linear algebra – multi-head attention in even more detail ... 35
3.2.7 The decoder side of things – masked attention and cross attention ... 36
3.2.8 Putting it all together – a second view to the high-level architecture ... 37
3.3 The BERT model architecture ... 38
3.3.1 Overview of the architecture ... 39
3.3.2 On the weights of BERT ... 40
3.3.3 On the training of BERT ... 42

4 A STUDY OF THE STRUCTURE OF BERT EMBEDDING VECTORS ... 45
4.1 A few words on the research methodology ... 46
4.2 Embeddings before the encoder ... 47
4.2.1 The tokenization ... 49
4.2.2 The input embedding vectors ... 51
4.2.3 The positional encoding vectors ... 54
4.2.4 The (two) sentence type encoding vectors ... 60

4.2.5 The interrelations of the context-free embedding types ... 61
4.3 Evolution of embeddings within the encoder ... 62
4.4 Semantic content and structure of the embeddings after the encoder ... 65
4.4.1 Topology of embeddings ... 67
4.4.2 Algebra of embeddings ... 69
4.4.3 Geometry of embeddings ... 72
4.5 Some practical applications of the embeddings ... 74
4.5.1 Vector databases for text search ... 74
4.5.2 Retrieval Augmented Generation ... 75
4.5.3 Compute-effective NLP ... 76
4.5.4 Interpretability ... 76
4.5.5 Bias detection ... 76

5 CONCLUSION(S) ... 77
5.1 Some post facto reflections and a critical look at the research methodology ... 77
5.2 Some possible further research questions ... 79
5.3 Epilogue ... 80

BIBLIOGRAPHY ... 81

1 Introduction

This thesis considers a subfield of artificial intelligence called *Natural Language Processing* (NLP). More specifically we study a language model named *BERT*, based on the so called *transformer* architecture. Like many other NLP systems, BERT relies on transforming input text into internal representations called *embedding vectors*. These embedding vectors live in a high-dimensional Euclidean space and encode both syntactical and semantical content of the input text. Understanding these embeddings is crucial not only to a deeper understanding of the BERT model and other language models like it, but also for various practical tasks. Indeed, the final BERT output embedding layer can be used as a connection point for further neural network layers, as input features for other machine learning algorithms, or as a numerical representation of a piece of text corpus to enable data search and analysis.

In this thesis we study the structure of the transformer architecture in general and the BERT model in particular, with a deep focus on the embedding vectors and the embedding spaces they reside in. We'll use both synthetic and real-world data to get a better grasp on both the theory and application of the BERT embedding vectors. Though we focus on the particular BERT model, many of the main ideas here should transfer to aid in understanding other transformer based models like FinBERT, RoBERTa, GPT or LLaMA.

To focus our efforts, we list the following concrete research questions for our work. We see these as three sides of the same coin.

1. The transformer architecture contains conceptually different types of embedding vectors like *positional encodings* and semantic *input embeddings of individual tokens*. Can we detect the difference between these types of vectors?
2. As we'll learn in the following chapters, the transformer forms the embedding vectors as a sequence of embeddings that gain more and more context awareness as they evolve. Can we see this progression?
3. The transformer produces embedding vectors that encode the semantic content of both the individual tokens and the input text as a whole. Suppose we alter the semantic content of the input text. Can we then detect changes in:

(a) The topological structure of the embedding vectors?
(b) The linear algebraic structure of the embedding vectors?
(c) The geometrical structure of the embedding vectors? (In the setting of larger collections of text and the data clouds their embedding vectors form.)

1.1 Contents of this thesis

We'll begin here by describing the aims and context of this thesis in a bit more detail. Before describing what we do, we'll tell first what we won't do; i.e. we give an exposition on the context of this study.

1.1.1 What we won't do in this thesis

In mathematics there is sometimes less value in a theorem than in the counterexample that shows the limits of what can be proven. In the same spirit: what we do in this thesis is only a small part of the area we are working in, and so in this subsection we'll look what other things we could do but don't.

Understanding how modern language models work is a topic of active research. Even if we ignore the so called Large Languge Models (LLMs) like GPT 3+, LLaMA or PORO, the "small" language models like BERT, RoBERTa, ELMo, GPT-2, T5 or ERNIE of "only" a few hundred million parameters are still quite opaque with respect to their inner workings.

There are various ways that one can approach in trying to understand the behaviour of these models. All of these approaches are based on the idea that the models are too big to understand by studying them from first principles, e.g. by looking at the model "one neuron at a time". Instead we are more in the realm of natural science where we run experiments and create theories from the results. For approach methodologies we list the following, but we emphasize that this is *not* a comprehensive or authorative list on the topic.

1. We can study the embedding vectors of various text collections in the model. As mentioned, the embedding vectors form an internal representation of text for the model, and understanding both the formation of singular embeddings and the shape or geom-

etry of larger embedding point clouds can be very useful for understanding the models. See e.g. Ethayarajh 2019 or Reif et al. 2019.

2. We can study the so called *attention mechanisms* of the model. These are the internal system of the transformer models that have a crucial role for the generation of embedding vectors.[1] See e.g. Kovaleva et al. 2019
3. We can do so called *ablation* studies where we retrain the model while removing some parts of the model, training data or training schema. The changes to the performance then yield some information on the inner workings. See e.g. Michel, Levy, and Neubig 2019, or again Kovaleva et al. 2019.
4. We can try to *prune* our models, that is, we can try to see how many weights or neurons can be removed from the model after the training without hindering its behaviour. Removing neurons and observing the changes to performance should yield information on which parts of the model are responsible which tasks. See e.g. Li et al. 2020.
5. We can test how the model performs on various benchmark tasks and on varying data and then try to infer why observed behaviour happens.[2]
6. We could use comprehensive existing tools like BERTviz[3] (Vig 2019) or SLIPMAP (Björklund, Seppäläinen, and Puolamäki 2024) to analyze and visualize the model structures.

These are of course not all disjoint approaches and many research approches combine many of them.

So there are several different interesting approaches to study language models. In this thesis we'll focus only on a narrow sector of the possibilities, namely the study of embedding spaces via looking at the embedding vectors of natural text.

1. We'll go through these in detail in the next chapter.
2. This is too generic of an idea for us to point to a single paper. E.g. Rogers, Kovaleva, and Rumshisky 2020 contains descriptions of several benchmarks and the performance of various models on them.
3. https://github.com/jessevig/bertviz

1.1.2 What we'll do in this thesis

As mentioned, our topic of interest in this thesis are the embeddings. But how shall we study them? Many great things in the world have been brought about from the idea of "let's disturb it a bit and see what happens" with calculus and snowglobes being prime examples. We will follow this path and, after getting the basics in order, do various experiments where we vary the input and observe the changes.

More concretely; after familiarizing ourselves with the tranformer architecture, we'll first study the basic ideas of the embeddings spaces by looking the embeddings of various synthetic example sentences. We'll also look at the point cloud of the embeddings of a random sampling of texts from a few open source sources to get some feel for the geometric shape of the embedding space. We also probe the formation and evolution of the various types of embeddings and aim to observe some of the classical structures that there are in the embedding spaces and their interrelations.

1.1.3 Structure of the thesis

In the Chapter 2 we'll describe some of the background for this thesis and describe some of the conceptual challenges in trying to comprehend language models. We also discuss about the techniques and tools we use for visualizing and understanding of high dimensional data. We go through the licensing of the datasets and images we use, and finish with some recommendations for prerequisites and further study.

In Chapter 3 we study the transformer architecture. We will begin by going though the history of transformers. We'll set up the context of RNNs and LSTMs where transformers emerged and then study the recent proliferation of the field by various transformers. The chapter continues with a somewhat through study of the architectures of both the original transformer and the BERT model.

Chapter 4 is dedicated fully to the embedding spaces of transformer models. We discuss the structure of the embedding spaces and do various simulations and tests on the various embeddings of both singular sentences or tokens, and of larger data clouds.

Finally in Chapter 5 we will present our conclusions and discuss further interesting avenues of study.

2 Preliminaries

In this chapter we go through some general background. We start with notation and definitions and then move on to various topics in the conceptual space of language models and in some tools that can be used to fathom data in higher dimensions.

2.1 Some notation and definitions

We denote by $\mathbb{N} = \{0, 1, 2, \ldots\}$ the natural numbers, including 0, and likewise use the shorthand of $\mathbb{R}_+$ for the non-negative real numbers $[0, \infty)$.

By $\mathbb{R}^n$ we denote the n-dimensional *Euclidean space* for $n \geq 1$, i.e. the set

$$\{(x_1, x_2, \ldots, x_n) \mid x_i \in \mathbb{R}\}.$$

We often denote the vectors in $\mathbb{R}^n$ by $\mathbf{v}$, $\mathbf{w}$, $\mathbf{u}$ or some other bold font letter. In these cases we implicitly assume that the coordinates of the vector are denoted by subscripts of an unboldened letter, e.g. the components of the vector $\mathbf{w}$ are $(w_1, w_2, \ldots, w_n)$, and we tacitly assume that the dimension n is clear from the context.

We equip $\mathbb{R}^n$ with the *inner product* (also called *the dot product*)

$$\langle \cdot, \cdot \rangle \colon \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}, \quad \langle \mathbf{v}, \mathbf{w} \rangle = \sum_{i=1}^{n} v_i w_i$$

and the *Euclidean norm*

$$\| \cdot \| \colon \mathbb{R}^n \to \mathbb{R}_+, \quad \|\mathbf{v}\| = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle} = \sqrt{\sum_{i=1}^{n} v_i^2} \quad \text{for } \mathbf{v} \in \mathbb{R}^n.$$

A vector with length 1 is called a *unit vector*. We call two vectors $\mathbf{v}$ and $\mathbf{w}$ *orthogonal* if $\langle \mathbf{v}, \mathbf{w} \rangle = 0$, and *orthonormal* if they are orthogonal unit vectors.

With the inner product we can also define a crucial concept in this thesis - *the cosine similarity*.

Definition 2.1.1. Let $\mathbf{v}, \mathbf{w}$ be two vectors in $\mathbb{R}^n$. We set their *cosine similarity* to be the number

$$\text{cs}(\mathbf{v}, \mathbf{w}) = \frac{\langle \mathbf{v}, \mathbf{w} \rangle}{\|\mathbf{v}\| \cdot \|\mathbf{w}\|}.$$

Note that orthogonal vector have cosine similarity of 0, and the cosine similarity of a vector with itself is always 1. Furthermore, scaling of either vector by a non-zero scalar has no effect on the vectors' cosine similarity.

The standard interpretation is that the cosine similarity measures the angle between two vectors, though with the difference that 0 means orthogonality and 1 means having the same direction. The cosine similarity is a sort of distance measure, though it is not a metric - one might call it something like a *similarity* instead. Note also that the cosine similarity is the very same object we see in the cosine rule of planar geometry:

$$\cos(\theta) = \frac{\langle \mathbf{v}, \mathbf{w} \rangle}{\|\mathbf{v}\| \cdot \|\mathbf{w}\|},$$

where θ is the angle between the two vectors – hence the name. Note that this interpretation translates naturally to higher dimensions: any two vectors in $\mathbb{R}^n$ that are not scalar multiples of each other span a 2-dimensional vector space. We can naturally identify that 2-dimensional vector space with $\mathbb{R}^2$ via a linear isometry which preserves both the inner product and the norm, and thus the cosine similarity.

For an $m \times n$ matrix M and an n-dimensional vector v we denote the matrix product of M and $\mathbf{v}$ simply as $M \times \mathbf{v}$, i.e. we set

$$M \times \mathbf{v} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^n a_{1,j} v_j \\ \sum_{j=1}^n a_{2,j} v_j \\ \vdots \\ \sum_{j=1}^n a_{m,j} v_j \end{bmatrix}.$$

2.1.1 A few words on complex indexing

Calculations with vectors and matrices can be somewhat heavy in their indexing syntax. We try to avoid situations that require complex indexing as much as we can. The biggest exception will occur in Chapter 3 when we discuss the so called *attention mechanism* of a transformer. Without going into too many details, there will be transformer blocks that contain several attention heads, but *we do not try to track these with indeces, nor do we index any vector components or matrix values*. So when working through that section we ask the reader to keep in mind that whenever we list matrix products or other vector operations, the

sub-indeces only index the various embedding vectors *within* the block that we are studying at the moment.

As we'll briefly mention in Section 3.2.6 when building these systems for actual pipelines the situation is a much more subtle. In particular when using CUDA or other libraries that get their true power from parallization capabilities and it is paramount to get various dimensions to match up in efficient ways. But all of this is way beyond the scope of our thesis.

2.2 The maskless Shoggoth on the left – the incomprehensible internal world of language models

In the world of the large language models, the difference between the language model GPT and the user-friendly chatGPT system is sometimes described by saying that GPT is "the maskless Shoggoth on the left" and chatGPT is "the masked Shoggoth on the right" – see e.g. (Alexander 2024) and the references therein. This is a nod to the Cthulhu Mythos from the literature of H.P. Lovecraft and refers to the fact that especially the large language models are truly uncomprehensible and *understand language in a completely different way that humans do*.

The so called "small" language models like BERT are not much more transparent, but their nicknames are not references to elder gods. We hypothesize that this is mostly due to the fact that their performance is not as human-like as with the LLMs and thus they create less of an "uncanny valley" effect. Regardless, our point here is that we should not expect language models like BERT to map directly to the same linguistic concepts that humans use.

The BERT architecture is modular and built from components that have clear architectural boundaries, but a given neuron in this 110M parameter neural network doesn't "know" if its inputs and outputs are on the boundaries of these components or not. So even when we mention research that has identified specific blocks to correlate between certain linguistical phenomenon, we should remember that this should not be expected in general. Instead we feel that the architecture simply provides a canvas within which the model can learn a sub-structure that fits the training data. This idea is supported by the various pruning studies, see e.g. (Guo et al. 2019), (Gordon, Duh, and Andrews 2020) and (McCarley 2019), which

suggest that a large portion of the weights in the BERT model can be removed with very little effect on the performance. In particular (Gordon, Duh, and Andrews 2020) find that even 30%-40% of BERT's weights can be dropped with very small detrimental effects. See also (Li et al. 2020), where they argue that it is often better to train a large model and prune it aggressively than to train a smaller model and prune it just a little.

This idea of the full set of model parameters providing an ambient space for the "true" model is mirrored in the embedding spaces. As we shall see, the 768-dimensional spaces are not evenly used to host the embedding vectors.[1] Thus we think it is not unreasonable to assume that there is a "true" embedding space[2] that is situated within the ambient 768-dimensional Euclidean space. This "true" embedding space forms the model's internal model of language, and we study it with the distorting lens of projecting parts of it to human concept space.

2.3 On undertanding and visualizing large dimensions

A large part of our main work will be in trying to understand the structure the embedding spaces of the BERT-style models. Our main tools will be studying either singular or few embedding vectors "by hand" or by analyzing the point clouds we get from embedding larger collections of text data. We can barely visualize four dimensions, so getting a grasp on these 768-dimensional point clouds can be very challenging.

Part of the issue relates to topics mentioned in Section 2.2. We have little hope that the internal concepts of the language model we study are aligned with familiar concepts. The issue is exasparated by the fact that the embedding vectors live in the Euclidean space $\mathbb{R}^{768}$, since high-dimensional geometry is not only hard to visualize but also often unintuitive. Here we also run into the problem that even if the embeddings correspond well to linear algebra, we have no reason to assume that the language model has "chosen" the standard basis for its concepts. By this we mean that the model has a priori no reason to use individual neurons

1. We mean this in the descriptive sense. Naturally the finite set of all possible texts that humans can form before the heat death of the universe cannot be evenly distributed in any unbounded Euclidean space. What we mean here, and what we'll see later on, that the distribution of the embeddings is remarkedly skewed from any uniform distribution that we might manually construct.

2. Well, several embedding space**s** as we'll see.

for specific tasks but can instead encode results in a superposition of many neurons.[3]

In any case, since we can't rely on magically finding a special basis with which to view our embedding spaces, we will have to use other tools to get a better idea of the structure of our high dimensional point clouds. Even with a perfect selection of a basis, though, there would be secondary issues with the fact that the structure of e.g. a 500-dimensional torus embedded in some quadrant of $\mathbb{R}^{768}$ probably wouldn't be evident from looking at the raw coordinate values of some sampling of vectors. For the purposes of this thesis, we have automated some tools to produce various two dimensional graphs and projections of these high dimensional data clouds. The aim of these is to help understand the approximate *dimension*, *shape*, *size* and *location* of the data clouds. We call the combination/superposition of these properties the *geometry* of a datacloud. Though the full geometry of the data still cannot be captured in these few descriptive extracts, we will nevertheless be able to describe some prominant aspects of it.

As mentioned, the geometry of a high-dimensional datacloud is not trivial to comprehend from statistical figures. A powerful, though limited, tool is that of projecting data to smaller dimensions, in particular to 2D. For this we dedicate its own subsection.

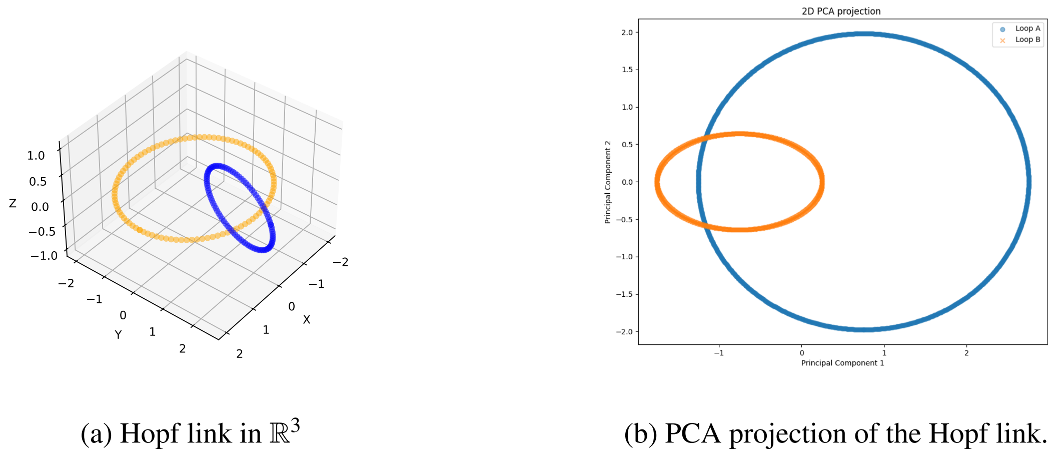
2.3.1 Projections to 2D

A natural way for us to visualize high-dimensional data is to project the data to two dimensions which we can easily show on a screen or a printed page. In this regard the classical tool is *PCA* or *Principal Component Analysis*, see Jolliffe 2002. This method is based on finding the eigenvectors of the covariance matrix of the data and projecting all of the data to the subspace spanned by the two vectors with the largest eigenvalues. If you imagine a flattened ellipsoid in three dimensions, it will have three axes and for a statistically significant data sample from this shape PCA will detect an approximation of the two largest axes. We can then orthogonally project the data to two-dimensional subspace spanned by these two directions. PCA will be one of our major tools, as the result will be a *linear orthogonal projection* of the original data, and thus preserve many of the algebraic quantities we are interested in.

3. See e.g. Olah et al. 2022 for discussion how these superpositions might function in a transformer.

Besides PCA, ther are two non-linear projection methods widely in use: *t-distributed Stochastic Neighbor Embedding* (t-SNE), see Van der Maaten and Hinton 2008 and *Uniform Manifold Approximation and Projection for Dimension Reduction* (UMAP), see McInnes, Healy, and Melville 2018. The t-SNE method is based on modeling the high-dimensional data and a prospective 2D projection dataset by forming a probability distribution of the pairwise distances of points. The method then tries to minimize the Kullback-Leibler divergence of these two distributions.[4] The UMAP is based on modeling the high-dimensional dataset and its prospective projection as nearest neighbour graphs. Like the t-SNE, it then works to minimize various distortions between these structures – we won't go into more detail here and refer the interested reader to the references mentioned.

The crucial point here is remembering that PCA is a linear projection, while both t-SNE and UMAP allow for various distortions. The latter two are often better in creating visualizations where components that are clearly separate in high dimensions can also be seen to differ in the projection. For an example, we study how a *Hopf link* might be projected in these systems. Here a Hopf link is a fancy word for having two circles or rubber bands that are linked in $\mathbb{R}^3$. In Figure 1 we see both the Hopf link and its PCA projection.



(a) Hopf link in $\mathbb{R}^3$

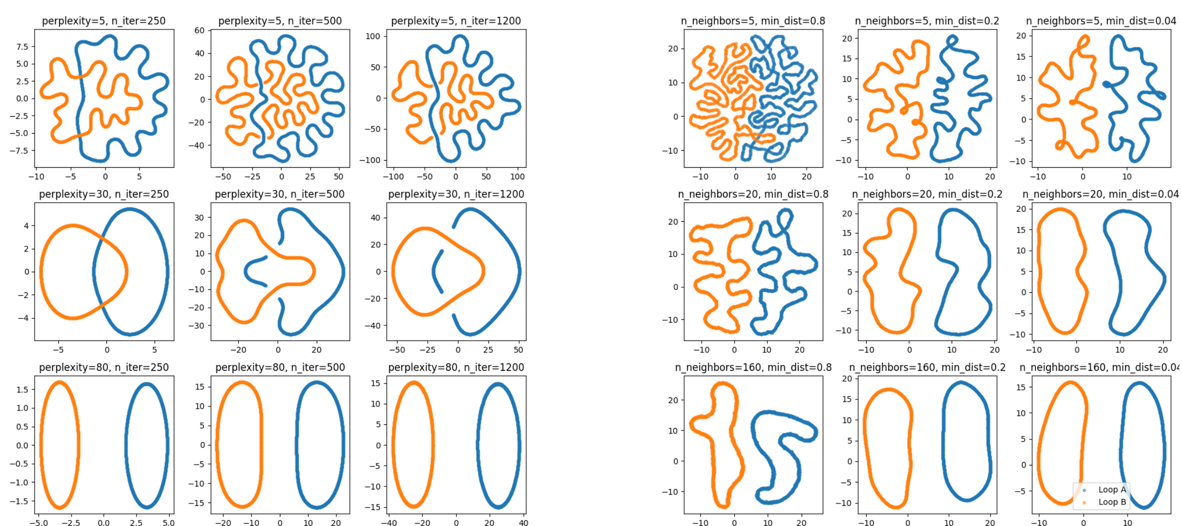
(b) PCA projection of the Hopf link.

Figure 1: The Hopf link and its PCA projection.

4. The Kullback-Leibler divergence is sometimes described as "the average amount of surprise you experience if you think you are sampling from distribution X while you actually are sampling from distribution Y". Here by "surprise" we refer to the information content of a random event. For further information we refer to the classical book MacKay 2003.

To contrast this, in Figure 2 we have the t-SNE and UMAP projections of the Hopf link. Both of these methods allow for several different parameters that affect their behaviour; we've shown here a few. Note that from these projections it is easier to grasp the true fact that there are two separate components in the data, both of which are circular. What is lost completely, however, is the fact that these components are exactly circles and linked.

We wish to emphasize that none of the methods PCA, t-SNE nor UMAP use the labels we have for the data points. All of the projection methods create the projection based on the full data cloud and the coloring of the components we have is *added after the fact*.



(a) t-SNE projections of the Hopf link with a few different parameters.

(b) UMAP projections of the Hopf link with a few different parameters.

Figure 2: The t-SNE and UMAP projections of the Hopf link with various parameters.

We wish to conclude by emphasizing the fact that t-SNE and UMAP are *strongly non-linear* projections. As we saw in the Hopf link example (see Figures 1 and 2), they are able to visualize for us data components that are separated from each other in some intuitive sense, even in cases where the PCA can not detect a separation. The downside here is that this comes with the cost of geometry distortion. For example in the Hopf link example the t-SNE and UMAP methods are able to display that there exists two distinct components in the data that do have some distance between them. But in reality, even though there are two topological components, they cannot be separated by any hyperplane (or even a

toplogical plane). Furthermore any properties related e.g. to distance or cosine similarity are not preserved in the projection. But even with these limitations these tools can be very useful – if we imagine that instead of knotted loops our embedding point clouds in $\mathbb{R}^{768}$ contain several 766-dimensional objects that have been knotted to each other[5], the fact that this structure can be visualized by the non-linear projections can be very illustrative. In any case, we advice caution when interpreting any results that try to condense high-dimensional data to only a few dimensions, be the method UMAP or histograms.

2.3.2 PCA-based dimensional analysis

Besides projection, we can use PCA to estimate the dimension of a data cloud. For an example we look at 10-dimensional dataset with three components. The first one is an embedded 7-dimensional unit cube $[0,1]^7 \times \{0\}^3$ that has been shifted off-origin by the vector $(2,\dots,2)$. The second one is an embedded 8-dimensional unit cube $[0,1]^8 \times \{0\}$ rotated randomly. The third is just an embedded 9-dimensional unit cube $[0,1]^9 \times \{0\}$.

In Figure 3 we plot the results of taking the PCA projections of this dataset, both in total and each component separately to dimensions 1-10 and seeing how much of the variation of the data is explained by the projection.The graph clearly captures the differing dimensional structure of the components since we know that they are there, but from the total dataset curve we can get very little insight in this case.

2.4 On various other things related to this thesis

We conclude the introductory section with a few miscellaneous topics.

5. Knots and links in $\mathbb{R}^3$ are just 1-dimensional spheres with complex embeddings, and similarly you can "knot" $(n-2)$-dimensional spheres in $\mathbb{R}^n$. We're not going to give an example as we suspect that a giving concrete example of a knotted 766-sphere might is at least a bit non-trivial. For anyone interested, we'd suggest starting with repeated double suspensions of the Hopf link, as these might be knotted.

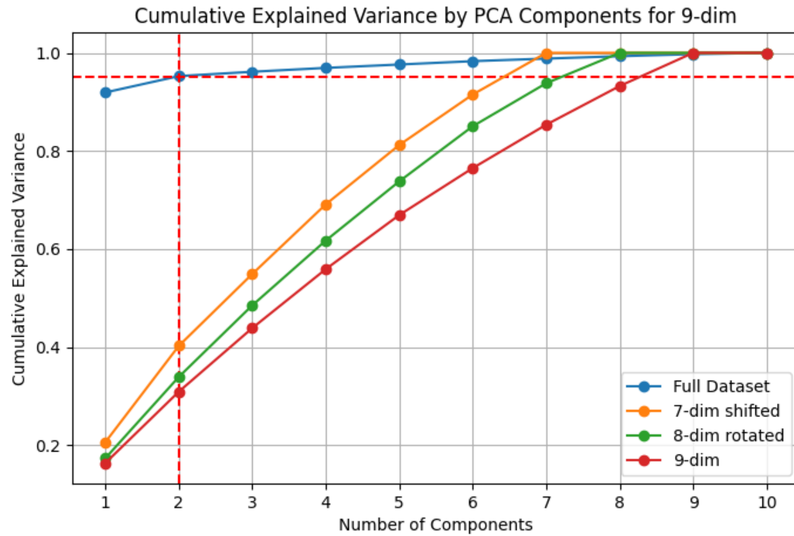


Figure 3: PCA-based dimension estimation.

2.4.1 On preliminary requirements for the reader

The main purpose of this thesis has been to help the author to learn the topic. We have, however, written this thesis in a way that we hope might be useful for other people wishing to study the topic as well. In this spirit we note that for anyone who wishes to thoroughly study this work, you should first have a base knowledge of deep learning and neural networks. At least at the level of topics like what are neurons, backpropagation, hidden layer, gradient descent, layer normalization and so forth.

For anyone feeling insecure in their level of background information, you might try to read this thesis anyway and skip any of the technical parts that are beyond your current grasp and rely on the high-level descriptions. Alternatively, or in parallel, you might want to read up on these topics. To get started, we suggest the following sources, with a roughly increasing order of comprehensiveness of the topic.

1. The "elements of AI" MOOC by the University of Helsinki and Reaktor.
2. The Youtube channel "3Blue1Brown" and their AI-video tutorials.
3. The Coursera course on deep learning by Andrew Ng.
4. The book Prince 2023.

2.4.2 What to read after this thesis?

If this topic seems interesting, we recommend the following sources for further study:

1. "Read the old masters", i.e. look at the original transformer and BERT papers from a few years ago: Vaswani et al. 2017 and Devlin et al. 2019. These are quite readable and well written in our opinion.
2. The excellent survey Rogers, Kovaleva, and Rumshisky 2020 collects many topics of what was known about the behaviour of BERT in 2020.
3. The book Tunstall, Von Werra, and Wolf 2022 for anyone interested in building transformer-based systems in practice.
4. The book MacKay 2003 for the mathematical basics of information theory and then some ML topics.
5. The book Prince 2023 for a comprehensive modern study on deep learning.

2.4.3 Images and licenses

The supermajority of the images and graphs here have been generated by the author with Python, matplotlib, seaborn, MS Visio or Inkscape. The few images borrowed from articles or books are used within their respective licenses, which are pointed out in the Figure captions.

We use various data sources for linguistical text.

1. We scrape a collection of random articles from Wikipedia to study the geometry of embeddings. The scraping script is available at https://github.com/ramiluisto/NLP_toybox/blob/main/src/wikipedia_datafetch.py and uses the Wikipedia API according to their licence.
2. We use WordNet data, see Miller 1995, as a source of a large collection of English nouns, verbs and adjectives. The WordNet licence is available at https://wordnet.princeton.edu/license-and-commercial-use.
3. We use a Kaggle dataset available at https://huggingface.co/datasets/tweet_eval/tree/main/emotion for a source of random tweets. This dataset is connected to the paper (Barbieri et al. 2020) and used within its license.

3 An overview of the transformer architecture

In this chapter we'll focus on the general idea and structure of the transformer architecture, both in general and the BERT model in particular. We'll first shortly discuss the history and context of where the transformers appeared. We then move on to describe the original transformer architecture from Vaswani et al. 2017 in some detail. We continue by studying how the BERT model is put together and finally finish the chapter by discussing how it was trained.

3.1 The technological setting where transformers appeared

To set the scene, we'll start by looking at the history of transformers in the context of neural networks called *Recurrent Neural Networks* (RNNs). Before the transformer revolution started in 2017 the NLP neural networks were heavily focused on RNNs. They were used to the extent that in e.g. 2015 we get blog posts titled "The Unreasonable Effectiveness of Recurrent Neural Networks" from one of the co-founders of OpenAI; see Karpathy 2015.

The base idea of RNNs is that we start feeding them the input text one word[1] at a time. The RNN is a neural network that then processes the input as any feed-forward neural network, but the crucial difference is that besides giving an output for this particular word, they also output an internal state vector that is given as an extra input when the next token is processed. This gives rise to one of the main problems of RNNs: they are hard to parallelize, especially at training time. This is an issue when we want to do very deep RNNs or use very large amounts of data. There are ways to ameliorate the problem, see e.g. Martin and Cundy 2017, but the parallelization is not a natural property of the RNN architecture.

Another defining challenge with RNNs is that it is hard for RNNs to "remember" connections between parts of the text that are far apart. This causes issues both at inference time and training time as gradient decay happens easily and it is hard for the training information to backpropagate throught the RNN. A very important tool to help with this is the so called Long Short-Term Memory (LSTM) architecture (see e.g. Hochreiter and Schmidhuber 1997)

1. Or token or some other datum. We'll use the term word here for simplicity.

that creates a sort of data channel that the RNN cells can use to pass on (or forget) information from the distant past.

The transformer architecture provides an answer to both of these issues. The original transformer paper (Vaswani et al. 2017) appeared in arXiv in June 2017, with the authors all working with Google. The success of the new architecture is evident from the fact that only a year later in June 2018 we get the first GPT-paper (Radford et al. 2018) by OpenAI, and in October 2018 the BERT architecture is published (Devlin et al. 2019). These were then soon followed in the next few years by e.g. the further GPT-models by OpenAI, RoBERTa (Y. Liu et al. 2019), t5 (Raffel et al. 2019) and XLM-R (Conneau et al. 2019), to name a few.

According to (Tunstall, Von Werra, and Wolf 2022) the success and usability of the transformer architecture was very much influenced by the following factors:

1. The immense transferability of these models. The pre-trained base models had encapsulated language understanding on a very deep level, and many of them could be fine-tuned to more specific tasks with only a few hours of extra training. These would often perform better or at least on par with more custom made task-specific models.
2. The HuggingFace library; a portal where large pre-trained ML-models could easily be shared. This enabled the expensive and resource-consuming pre-training of a large transformer model to be only done once by a large actor like Google, Meta, or OpenAI, and the results then fine-tuned by anyone.
3. In (Howard and Ruder 2018) the ULMFiT approach to language model training was introduced. The paradigm introduced here was to train language models in ways that make transfer learning more natural, aiding the transformer revolution. We'll discuss this in slightly more detail in Section 3.3.3.

We refer the reader interested in more of the history of tranformer models to Tunstall, Von Werra, and Wolf 2022.

3.2 The architecture of the original transformer

The original transformer paper Vaswani et al. 2017 was describing a transformer used for translation, see Figure 4. Their system is built especially for translation tasks and, a priori, not much else. The BERT model that appeared a few years later in Devlin et al. 2019 had a different goal. Its whole purpose was to build a foundational model that could be used for various language tasks with no changes to the base architecture – for most tasks adding one additional neural network layer called the "head" is enough. Another impactful model following the original transformer paper was of course the GPT family of models by OpenAI, only first few of which are openly available. Their focus is less on generic language tasks and more on text generation, which admittedly can then be used in a very versatile manner when you have a model as powerful as GPT 3+.

In this section we wish to explain how the transformer architecture works, both in the high level and also in a somewhat detailed manner. We'll begin by giving an overview of the so called *encoder* part of the original transformer model. Our motivation here stems from the fact that the encoder part is the component that is used in our main model of interest, i.e. BERT. After the bird's eye view of the architecture, we'll jump into a more detailed view by studying each part of the encoder part in turn. We conclude the section by discussing how the *decoder* part of the transformer differs from the encoder.

Our description of the transformer architecture is based on several sources. The technical details arise from the original transformer paper (Vaswani et al. 2017), the BERT paper (Devlin et al. 2019) and data extracted from the Huggingface BERT model via the `torchinfo` -package; we'll study this in more detail in Section 3.3.2. Our exposition is also influenced by the video lecture series of *3Blue1Brown*[2] and the semi-interactive website https://bbycroft.net/llm. We won't try to give direct cites of the facts we mention in the following subsections, as the text represents an amalgamation of the knowlegde, but wish to emphasize that the exposition here is based on previous work.

2. https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

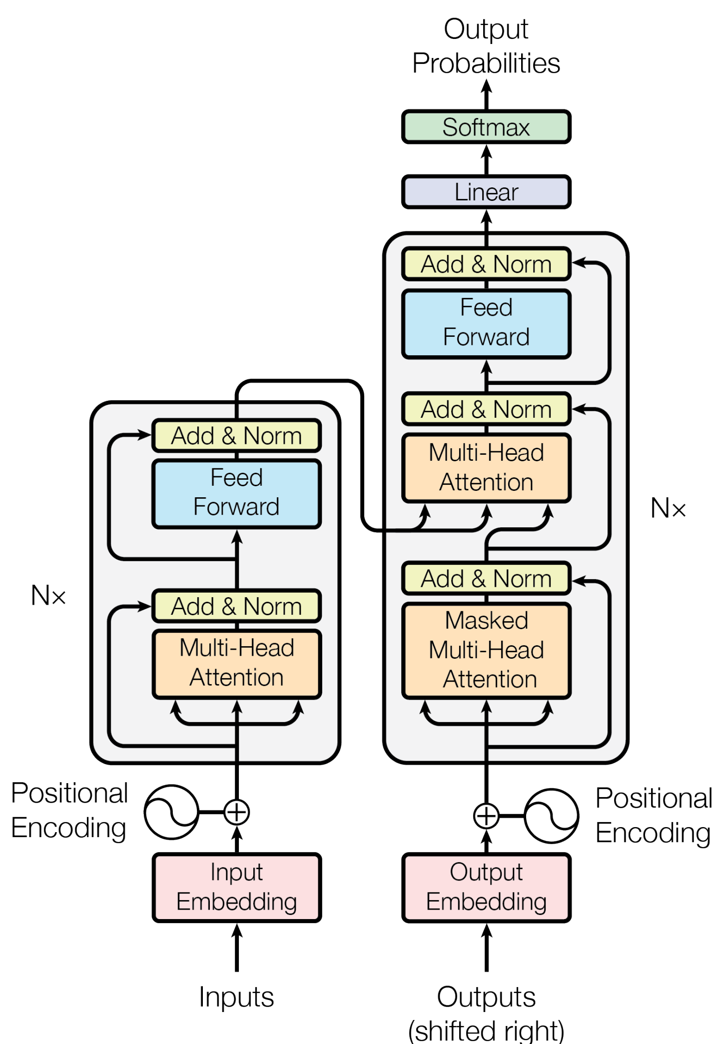


Figure 4: Image of **the** original transformer architecture from Vaswani et al. 2017. Encoder on the left, decoder on the right. Used within permissions granted by the authors.

3.2.1 A bird's eye view of the architecture

The Figure 4 shows the original transformer architecture with the *encoder* on the left and the *decoder* on the right. As mentioned, with only a moderate amount of oversimplification, the major difference between the BERT and GPT families of transformers is that BERTs uses encoders while GPTs use decoders. From Figure 4 we already note that the encoder and decoder are quite similar, except that in the decoder there are two attention layers; a *masked* attention layer at the start and then a second attention layer that takes in inputs both from the encoder component and the masked attention layer. Since our major focus in this thesis is

the BERT style of transformer, we'll focus for now only on the encoder part, and return to the decoder and its slightly differing attention strategies at the end.

We'll first go through the function of the encoder block on high level as numbered in Figure 5. We'll use the string "Then the bear waited for the door to open with a serendipitous smile." as our example input.

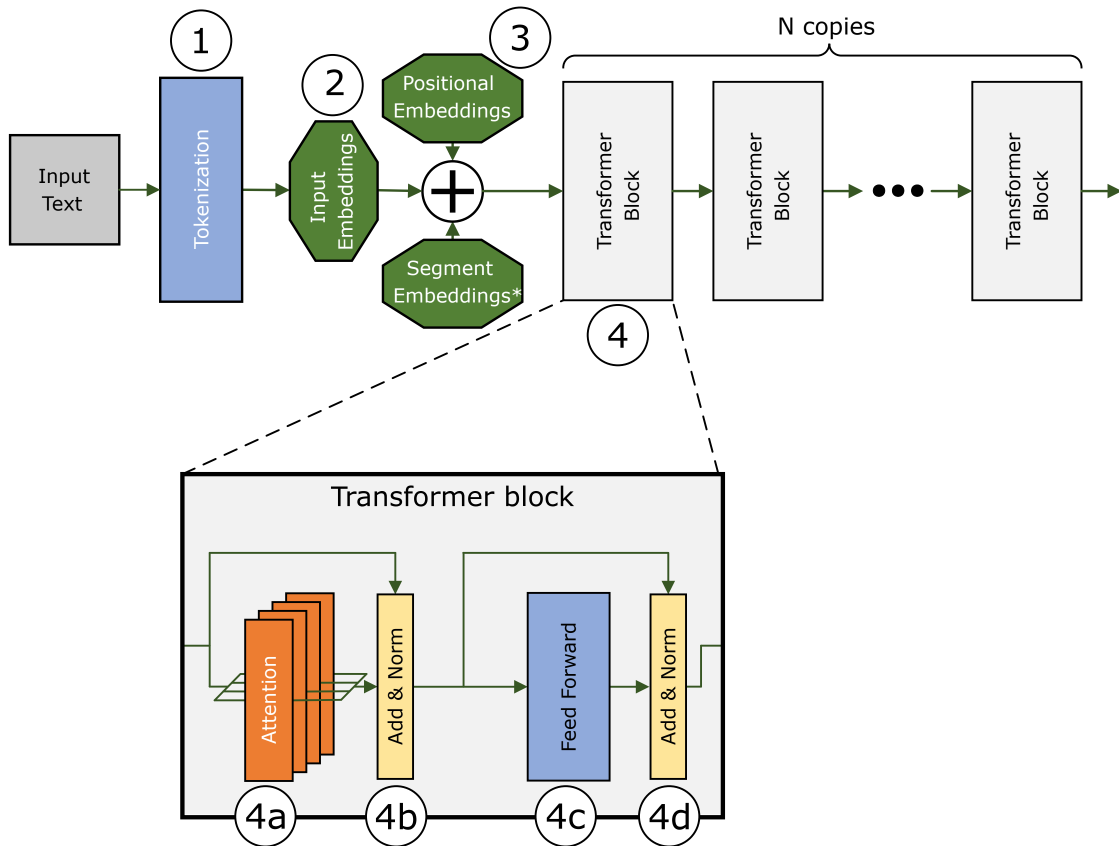


Figure 5: Illustration of a tranformer encoder. The "Segment Embeddings" in the figure are present in the BERT architecture but not in the original Transformer architecture.

1. **Tokenization.** The first step is to *tokenize* the string, i.e. turn the string into a sequence of numbers (or indeces) that the neural network can process. We'll use the example tokenization given by the BERT tokenizer here, the original tranformer paper used a different one so the exact tokenization would be slightly different, but the main idea is the same. We'll return to tokenization details in Subsection 3.2.2.

   In any case, the BERT tokenizer splits our text to following 18 parts: "'Then' 'the'

'bear' 'waited' 'for' 'the' 'door' 'to' 'open' 'with' 'a' 'se' '##ren' '##di' '##pit' '##ous' 'smile' '.'". Here the "##" prefix means that the word is *not* preceded by a space or a sentence start – for other tokens it is assumed to be there. Note that the common words are all one token, but the less common "serendipitous" is split into several *subwords*. Each of these tokens corresponds to a unique *token index* which is what the transformer will actually see. For our sentence, the token index list is as follows[3]; (1599, 1103, 4965, 3932, 1111, 1103, 1442, 1106, 1501, 1114, 1126, 14516, 5123, 3309, 18965, 2285, 2003, 119)

Thus this part has taken in a string S and ouputs a sequence of 18 token indeces. Let's call them $(t_0, t_1, \ldots, t_{17})$. Depending on the ambient system, token index sequences of length below the maximum window width are sometimes padded with special [PAD] tokens, but we'll ignore this topic in this thesis.

2. **Input Embeddings.** With the sequence of tokens we advance in Figure 5 to the "Input Embeddings" box. The input embedding are learned vectors, one for each token (index). For the original transformer paper these vectors have dimension 512, for BERT they have dimension 768 – following Vaswani et al. 2017 we call this dimension d_{model}. These embeddings are called *context-free embeddings*, since they are based only on the token and not on any neighboring tokens or their position in the input text. For example, both of the tokens "the" in our example input text have the exact same input embeddings. We'll analyze the input embedding vectors of the BERT model in more detail in the next chapter in Section 4.2.2.

   Thus this component has taken in a sequence of 18 token indeces, $(t_0, \ldots, t_{17})$ and outputs a sequence of 18 vectors of length d_{model}. Let's denote these by $(E_0^{\text{input}}, \ldots, E_{17}^{\text{input}})$. This mapping from indeces to input embedding vector is a straightforward indexing; there is a matrix of size $\#(\text{tokenizer vocabulary}) \times d_{\text{model}}$, and the embedding E_j^{input} is simply the t_j:th row of that matrix.

3. **Positional embeddings.** After the input embeddings have been generated (or, more

3. If you are following along by testing the tokenization, the BERT tokenization starts this sequence with an extra token index 101 and ends the sequence with the extra token index 102. These are the special [CLS] and [SEP] tokens, respectively, and we'll discuss them further when we study the training of BERT in Section 3.3.3.

aptly, selected) we move on in Figure 5 and add to them the *positional encodings*. They are vectors of same length as the input embeddings, i.e. d_{model}. They are added to the input embeddings to provide information to the neural network about the relative and absolute position of the tokens. These are simply summed to the input tokens, and now e.g. the two words "the" have differing embeddings. We'll study the positional encodings in much more detail in Section 3.2.4.

So in short, this layer maps the sequence of input embeddings $(E_0^{\text{input}}, \ldots, E_{17}^{\text{input}})$ sequence of position enhanced embeddings which we denote by $(E_0^{\text{in+pos}}, \ldots, E_{17}^{\text{in+pos}})$. Here the j:th position enhanced embedding is then just $E_j^{\text{in+pos}} = E_j^{\text{input}} + \text{POS}_j$ where POS_j is the j:th position encoding vector.

4. **The Transformer block.** Now we move on in Figure 5 to the core idea, the part that contains the Multi-Head Attention mechanism mentioned in Figure 4. This gray box is often called *a transformer block* and it is basic building block of transformers. There are N of these stacked sequentially. For the original transformer paper $N = 6$, for BERT $N = 12$.

   Each transformer block functions by taking in a sequence of embeddings and outputting a sequence of embeddings of the same dimensions. So each transformer block takes in a sequence $(E_0, \ldots, E_k)$ of d_{model}-dimensional vectors and outputs a sequence $(E'_0, \ldots, E'_k)$ of d_{model}-dimensional vectors.

   For the first transformer block we feed in the positionally enhanced token embeddings. With our example sentence we had 18 of these of dimension d_{model}, and the transformer block will output 18 new embedding vectors of dimension d_{model}. After the first transformer block, these embeddings are called *context-aware embeddings* since they are affected by all the other tokens in the input. For example, the sentences "The cat." and "The hat." would both have different context dependent embedding vector for the word "The" from this point on.

   Now let's look at the transformer block contents in a bit more detail.

   (a) **Multi-head attention.** For the incoming embeddings there is first a multi-head attention layer. The *multi*-part means that we have several attention heads work-

ing in parallel. To describe the general function of a single attention head let's denote the incoming embeddings as $E_0, \dots, E_{17} \in \mathbb{R}^{d_{\text{model}}}$ – one for each of the original 18 input tokens. The purpose of an attention head is to produce for each of the embeddings E_j an "attention-based update" ΔE_j, which will be then used to update the embedding via $E_j \mapsto E_j + \Delta E_j$. The crucial thing here is that the ΔE_j vector will depend on *all of the other embeddings*! In fact what happens is that for each of the E_i the attention head will produce a so called *value vector* V_i, and the vector ΔE_j will be a weighted sum $\sum_i \alpha_i V_i$ of these vectors. So in a sense, each of the other tokens will produce a sort of difference candidate, and the attention mechanism is responsible for selecting which difference candidates should be taken into account with the greatest weights. We will return to this in more detail in Subsection 3.2.5.

From the more functional point of view each of the K attention heads will produce for each of the 18 tokens E_i an update vector ΔE_i^n, $n = 0, \dots, K-1$. The multi-head attention calculates these $18 \cdot N$ update vectors in parallel, and updates the embedding vectors as

$$E_i \mapsto E_i + \Delta E_i^0 + \cdots \Delta E_i^{K-1} =: E_i'.$$

(b) **First sum and normalization.** After this we have what is commonly known as a *skip connection*, which we denote by a large 'plus' for sum, meaning that we add the original vector E_i to the new vector E_i'. After this we normalize the result. So more formally we do

$$E_i' \mapsto \text{Normalize}(E_i + E_i') =: E_i^A,$$

where $\text{Normalize}(\cdot)$ is the layer normalization operation.

(c) **Feed forward neural network.** Now we get our updated embeddings to the feed forward part of the transformer block, where we have a single fully connected neural network with one hidden layer. We have d_{model} input neurons and d_{model} output neurons. The hidden layer has size 2048 in the original tranformer paper and 3072 in BERT. It is a single neural network specific for a single transformer block, and it is applied separately for each embedding E_i^A.

Note that in both the original tranformer architecture and in BERT the hidden layer in the feed forward neural network is much larger than d_{model}.

(d) **Second sum and normalization.** The result of the Feed Forward layer for each embedding is then again added to the input E_i^A and normalized, i.e. we get $\text{Normalize}(E_i^A + \text{FF}(E_i^A))$, where Normalize is again the layer normalization map.

To reiterate the whole transformer block process, we start with an input sequence $(E_0, \ldots, E_{17})$ of embedding vectors. Then for each $j \in \{0, \ldots, 17\}$, each of the K multi-attention heads will create an update vector ΔE_j^k that is a function of all the vectors $E_0, \ldots, E_{17}$. These are all added to E_j, i.e. we get $E_j' = E_j + \sum_k \Delta E_j^k$. Then we add a skip connection and normalize to get $E_j^A = \text{Normalize}(E_j + E_j')$. This is then fed to the FF NN, which is followed by another skip connection and normalization to arrive at

$$E_j^T = \text{Normalize}(E_j^A + \text{FF}(E_j^A))$$

where Normalize is once more the layer normalization map.

And thus we have walked our way through a single pass of a transformer block. Note that after the input token sequence has been turned into a sequence of embedding vectors, each of the transformer block maps the sequence of embeddings to a new sequence of embeddings of the same shape and dimension.[4] In particular, the final output of the encoder will also be a sequence of embeddings.

With this bird's eye view in mind we next turn into looking at each of the parts in more detail.

3.2.2 From words to indeces – tokenization in more detail

There are various ways to tokenize text input to transformers. The base idea here is that we need to turn the incoming string, i.e. a sequence of characters, into a form that a neural network can work with. In tokenization we split the incoming string to pre-defined substrings

4. This 'evolution' of embeddings of identical shape through this process of alternating multi-head attentions and feed-forward neural networks with interim normalized skip connections is sometimes called a *residual stream*.

called *tokens*. The collection of all possible tokens of a given tokenization algorithm is called the *vocabulary*. The tokenization process is strongly tied to the input embeddings of the model, and we can think of the tokenization part of the process as the question of specifying which parts of text 'deserve' their own input embeddings.

One extremal direction would be to use individual characters as tokens, whence we would assign each possible character a unique integer and work with the incoming text on a character-by-character basis. This would result in a small vocabulary size, but it would mean that the model would have to spend a lot of its effort to learn what words are and what they mean. Also we would 'burn' a lot of token bandwidth on simple words: the three character sequences of "the" and "xqq" would be represented by an equal amount of tokens even though we see "the" all the time and "xqq" very rarely.[5]

Another extreme would be to take the collection of all possible words as the token collection. This would mean that we do not have to learn what words mean but then our vocabulary size would be immense. Instead of having to spend a lot of effort in "trying to learn what words are", the neural network would instead have a huge amount of weights to train to learn the meaning of all words independently, including very similar words like "strong", "stonger", "strongest", "strongly" and so on. Also, any word with a typo would either have to be included or then the system would not understand non-perfect text at all.

A standard solution is to find some sort of middle ground between these two extermes. So the idea is to do something called *sub-word tokenization* based on the frequency or importance of different words and their sub-parts. Coarsely the idea is to reserve a token to the common and/or important words like "the", "and" and "bear", and then also take a suitable collection of other syllables like "##ness" that can be used to construct other words like "bearness". In the worst case scenario we also have a dedicated token for each character so that we can express also rare words, typoed words and nonsensical character sequences.

The original transformer paper used so called Byte Pair Encoding (BPE), see Britz et al. 2017, while BERT uses the WordPiece method, see Wu et al. 2016. Both have the same approach

5. For anyone familiar with Huffman coding; there is some sort of analogue here in what we are trying to accomplish.

to take important words to be tokens themselves and break less important words to subwords. The difference is how importance is decided and how the subwords are formed. We will skip the details of these differences in this thesis.

The sizes of the tokenization vocabularies, i.e. the total amount of different words and subwords represented by tokens, vary depending on the model and tokenizer. For e.g. the BERT model the vocabulary is around 30k different tokens, whereas the Finnish language FinBERT[6] has a vocabulary of around 50k tokens, owing probably largely to the complex morphology of the Finnish language.[7]

3.2.3 Turning "bear" into a concept – input embedding vectors in more detail

The idea of the input embeddings is to provide a context free *content* or *meaning* for each word in the tokenizer vocabulary. Even though these are sort of static when compared to e.g. the attention mechanism, we note that for example in the BERT model we have about 110M parameters and about 22M of those[8] are spent to learn and store the input embedding vectors. So this context-unaware listing of the base meanings of tokens is important enough to reserve a whole fifth of all the weights in the model.

What is their purpose then? In a pretend example we would like to have e.g. the input embedding of the token "bear" to contain information on the following topics.

- The token can be a noun. (As in, "I saw a bear in the forest.")
- The token can be a verb. (As in, "We must bear this burden.")
- The token is spelled with the tokens "b", "e", "a" and "r". (This should help detect typoed words.)
- The concept represented by the token can be related to forests and/or honey.
- The concept represented by the token can be related to something hairy and dangerous.
- ...

6. A BERT model trained with Finnish training data, see Virtanen et al. 2019.

7. Without going for a more detailed analysis, we can easily see that 22% of the BERT tokenizer tokens start with "##", compared to 26% in the FinBERT tokenizer. See https://github.com/ramiluisto/NLP_toybox/blob/main/notebooks/vocabulary_comparison.ipynb.

8. We'll study this in more detail in Section 3.3.2.

This is a manually made up list of things that pop to our mind when we think of the word "bear", and the actual information content of the input vector will probably not translate naturally to a discrete list of human concepts. Also some (or many?) of the properties of bearness are probably encoded in the various transformer blocks as some aspects of bearness depend on the context.[9] But for many of the tokens a lot of the information still needs to be encoded in the input vector – if a sentence is discussing a bear then the bear token is the only one guaranteed token to be present[10] and thus it really needs to contain at least a *key* to the data that the model has about bearness. So this is quite non-trivial, and we can't make very strong claims about exactly what we should expect the input embedding vector to encode. But can we say *something* at least?

To take a more simple example, let's ask if the embedding vectors really contain "noun-ness" in them? A standard approach to test something like this is called *probing*. In probing we take the embedding vectors of some dataset, and to test if the embeddings carry information about a topic X, we try to train various ML methods to extract X from the embedding data. We did a quick probing test for the context-free embeddings by sampling random nouns, verbs and adjectives from the WordNet data, extracting their input embeddings in the BERT model and training an XGBoost, a single hidden layer FC NN and a no hidden layer FC NN on the data. All of the methods achieved a 83-86% accuracy on the test set that had an even split of all of the three classes. So this would imply that there is some statistically relevant information about e.g. "noun-ness" learned in the context-free token embeddings.

For a further notion we looked at the output distribution of the two neural networks with the embeddings of the tokens "bear" and "Bear" as inputs,[11] see Table 1. Both of the no hidden layer and one hidden layer models predict both words to be a verb, but note that the capitalized version is considered much more likely to be a noun.

---

9. This claim is supported by the fact that advanced language models will also understand that "king of the forest", or "a large hairy mammal that eats honey and hibernates" point to the concept of a bear. To us this seems to imply that some parts of the concept of "bear" are stored outside the singular input embeddings.

10. In the BERT model we always have, as we shall see, also two special tokens called [CLS] and [SEP]. But these do not have the bandwidth to carry information about all possible other tokens.

11. We are using here the "cased" version of BERT that differentiates between capitalized and non-capitalized words.

Word	Model	adjective pred.	noun pred.	verb pred.
bear	No hidden layer	0.163	0.177	**0.660**
Bear	No hidden layer	0.225	0.374	**0.401**
bear	One hidden layer	0.0004	0.004	**0.996**
Bear	One hidden layer	0.0060	0.283	**0.711**

Table 1: Table showing the predictions of two different model types trained on the context free embeddings to predict whether the token is an adjective, a noun or a verb. The biggest value of each row is boldened. Note that the word "bear" can be either a noun or a verb depending on the context.

Trying to figure out exactly what all properties are encoded in a given input embedding is a very complex task and way beyond this scope of this thesis. We will continue our exploration later in Section 4.2.2, though we'll be largely focusing on the behaviour of the input embeddings as a point cloud.

3.2.4 Not a bag of words – positional embeddings in detail

One important property of the transformer models is that since the sequentiality of the inputs is removed, the attention layers do not 'know' what is the order of the inputs they have.[12] This is suboptimal as in natural language the order of words can carry crucial information, especially since we are using subword tokenization and longer words can be broken up to a collection of consequtive tokens. The classical solution to this is to include *position encodings* to the input vectors, and a standard way both in the original transformer and in BERT is to add the n:th positional vector to the n:th token embedding in the input sequence.

Positional encoding vectors can be designed by hand, and Vaswani et al. 2017 uses a classical technique based on trigonometric functions. In the original transformer we have the dimension of the embedding vectors $d_{\text{model}} = 512$ and the n:th token we set the k:th coordinate to

12. We'll explore this in more detail in the next section when we focus on the attention mechanism.

be

$$\mathrm{enc}(n,k) = \begin{cases} \sin\left(\frac{n}{10000^{2k/512}}\right), & k \text{ is even} \\ \cos\left(\frac{n}{10000^{2k/512}}\right), & k \text{ is odd.} \end{cases}$$

This generates a set of vectors that a neural network can learn both to differentiate from one another and to understand the absolute position of a given token. See Figure 6 for an illustration how these position encodings and their pairwise cosine similarities look like for the first 128 positions. We'll return to the position encodings and their design structure with the BERT architecture in Sections 3.3 and 4.2.3. For a more general treatise on the design of position embeddings in the context of the BERT architecture we refer the reader to Wang et al. 2020.

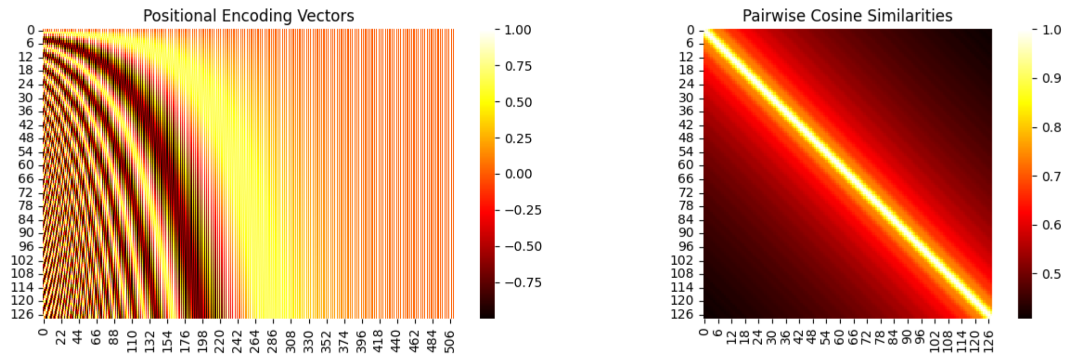


Figure 6: Classical manually defined sinusoidial positional encodings and their pairwise cosine similarities. On the left side each row represents a different positional encoding vector of dimension 512.

3.2.5 Where to look – the attention mechanism in more detail

Let's turn back to our example sentence "Then the bear waited for the door to open with a serendipitous smile." that has been turned into a sequence of 18 position-aware embedding vectors $E_0, \ldots, E_{17}$, each of length d_{model}. Imagine that we are at the first transformer block and we're looking at a single attention head. There are various ways to implement an attention mechanism, but both the original transformer and BERT use the so called "Query, Key and Value" (QKV) attention mechanism. To study this, suppose we are looking at generating the attention-based update vector ΔE_i for an embedding E_i.

What we do first is to generate a *query vector* Q_i for the embedding E_i and *key vectors* K_j, $j = 0, \dots, 17$ for all the embedding vectors. These vectors have the same dimension which we denote d_Q and for both the original transformer paper and BERT $d_Q = 64$. These are formed by attention-head specific matrices M_Q and M_K of dimension $d_{\text{model}} \times d_Q$ together with additive bias vectors B_Q and B_K. These Query and Key matrices with their biases are learned weights for the model.

So we form the query and key vectors as $Q_i = M_Q \times E_i + B_Q$ and $K_j = M_K \times E_j + B_K$, $j = 1, \dots, 17$. These query and key vectors live in a different conceptual domain as the embedding vectors, and the mental model here is that they encode a sort of query that a given token might "ask" in a sentence. To make this more concrete, we'll pretend that the attention head we are looking at is focused on the task of "figure out if the nouns of the text have definite or indefinite articles connected to them".[13]

In our example sentence "Then the bear waited for the door to open with an serendipitous smile." the third word/token "bear" is embedded as E_2, and the query Q_2 we form represents the question "Do I have an article related to me?". But how does the token say "I" or "me"? Well, it has the positional encoding included, so the query could be something like "Does the token at index 2 have an article related to it?". Then the key vectors we get for the first two tokens, "Then" and "the" might look something like $K_0 =$ "I'm not an article, ignore me." and $K_1 =$ "I'm an article at index 1." (The latter tokens would state similar things.) So how could we form such a query from E_2 and key from E_1? Recall from Section 3.2.2 that the context-free embedding vectors seem to contain some aspect of "noun-ness" in them, and we were even able to detect those with a neural net of no hidden layers. Thus this attention head with a focus on finding articles related to nouns would probably be able to learn parameters that highlight both nouns and articles in the query and key vectors.

Now the next step is that we start to compare all of the key vectors K_j with the query vector Q_i to see which of them are most applicable to the situation. The K_0 key vector should be quite dissimilar to Q_2, whereas K_1 should be much more apt for Q_2. For K_j, $j > 2$ the similarity should be negligible due to their position. The actual way we compare the similarities is by the *dot product of the query and key vectors*. Note that this is computationally beneficial,

13. I.e. we are asking if any given instance of a noun has "the", "a" or "an" attached to it.

since we want to parallelize as much as possible, and so we can calculate all the vectors Q_i, pack them into a matrix Q, do the same for the keys to get K, then we can simply calculate all the pair-wise dot products by calculating the matrix product $Q \times K^\top$. See again Figure 7.

Here we note again that this space of query and key vectors is conceptually distinct from the embedding space(s), but there are no distinct "key space" and "query space". Thus the example query we have here would probably look more like $Q_2 =$ "Article near and to the left of index 2." and our key vector might then look like $K_1 =$ "Article at position 1." Though of course the actual real-life attention keys and queries are not something we can turn into natural English expressions.

Anyway. With the similarities between the keys and the queries, we now know which other embeddings are most relevant for the task at hand. We then need to create the vector used to update E_2. Before going into the details, we note that there is yet another conceptual difference in play here. And that is the difference between *absolute amounts* and *differences/directions*. It's not as obvious a difference as the error you make if you try to add litres to meters or apples to oranges, but let's image that we are orienteering and talking to a friend over the radio. They say that their coordinates, relative to a far away fixed point, are "15km to north and 20km to east". You tell that your coordinates are "16km to north and 18km to east". It would make sense to look at e.g. the difference of your coordinates: $(15,20) - (16,18) = (-1,2)$ and tell your friend to move this difference i.e. add the difference to their position. But it would not be very meaningful to study what is the sum of your coordinates, $(15,20) + (16,18) = (31,38)$. So in a sense even though the math looks similar, some of the pairs of numbers here are absolute *positions* while other are differences or *directions*. We can get directions from substracting positions, we can add directions to absolute positions to get absolute positions and we can add directions together to get more directions, but *we should not add absolute positions*. And this is now relevant for our situation with the embedding vectors. Even if we find out from our query-key comparison that the embedding E_1 is relevant to the update of vector E_2 based on the query, we should not simply add a (weighted) E_1 to E_2 as these are, in some sense, positions and not differences. For this reason we turn to the third component of the QKV-attention: the *value vector*.

The value vector V_i for an embedding E_i has the same dimension d_{model}. But the idea here

is that the value vector is a difference or a direction and not an absolute position, so that we can add it to an embedding vector E_j. The value vectors are calculated with learned matrices and biases like the key and query vectors, but they use two matrices instead of one. The reason here is that the query and key matrices have dimension $d_{\text{model}} \times d_Q$, whereas a naive implementation of the value vector through a single matrix product would require a matrix of size $d_{\text{model}} \times d_{\text{model}}$. Since usually $d_Q << d_{\text{model}}$, this would require a lot more parameters to learn. Instead we implement the value vector by using two matrices of shapes $d_{\text{model}} \times d_Q$ and $d_Q \times d_{\text{model}}$. This drops the amount of parameters needed drastically, and has some other clever computational advances as well – see Section 3.2.6. In any case, we generate a value vector V_i for each of the embedding vectors and form a weighted sum of these to form the attention update vector ΔE_2.

We get the weights by taking the softmax of the dot products of the key vectors K_j with the query vector Q_2. We thus finally get $\Delta E_2 = \sum_j \alpha_j V_j$, and repeat this for all the embeddings E_i. For our example case the weights would most likely concentrate very heavily on the value vector V_2, which would probably have content in the vein of "increase noun-ness and definiteness of this token". Once more see Figure 7.

The whole QVD attention is very succintly expressed in Vaswani et al. 2017 as

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_Q}}\right) V,$$

where Q, K and V are the collections of query-, key- and value vectors concatenated into matrices. Note that they divide the dot products with the square root of the query space dimension before taking the softmax. They mention that for larger values of d_Q there seems to be an advantageous effect to the learning, which they hypothesize to be related to gradient propagation problems for softmaxed large dot product values.

And so we get to the end of a single attention head. To reiterate the attention process; we update each embedding vector with a weighted sum of the value vectors of the other embedding vectors. The weights are based on the dot product similarity of the query-key vectors of the embeddings, and the query-, key- and value vectors are formed by matrix products (including an additive bias vector) of an attention-head specific matrices and the embedding vector in question.

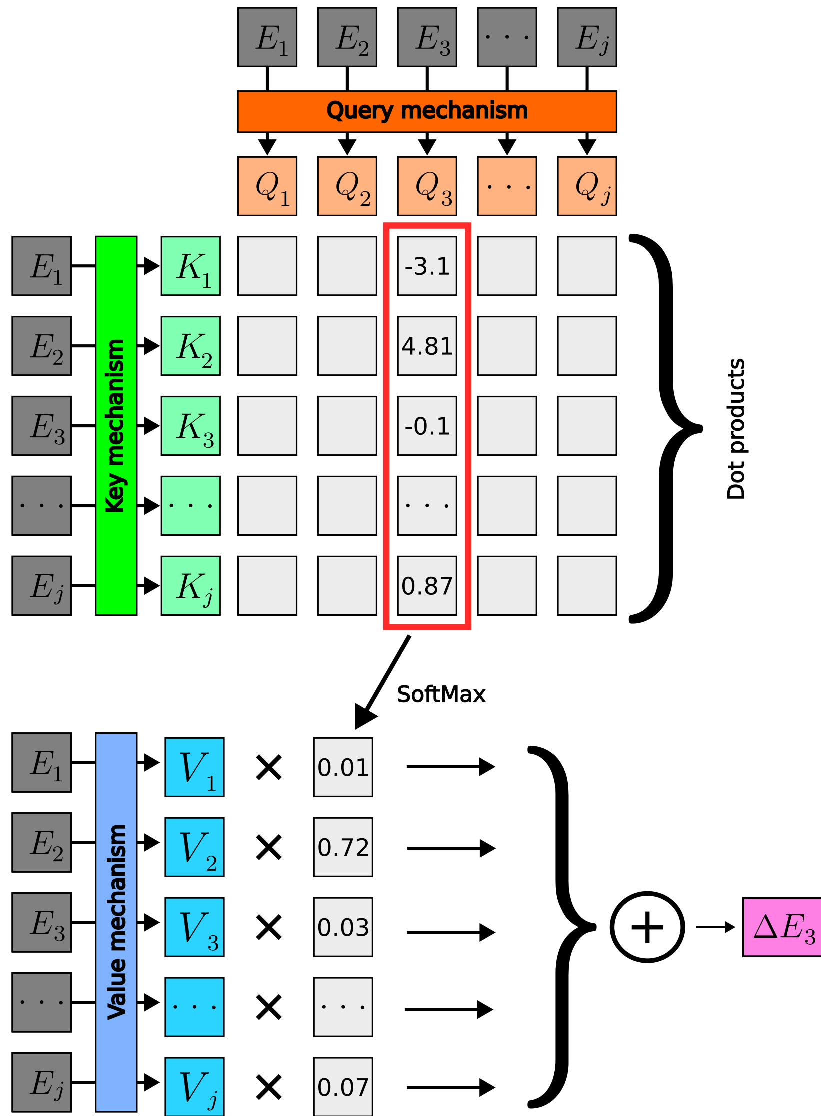


Figure 7: The self attention mechanism illustrated.

With the multi-attention head we simply do the attention head process in parallel with several attention heads. The k:th attention head produces, in the end, an update vector ΔE_j^k for each embedding vector E_j. These update vectors can be formed independently from one another and at the end step we just add them all to E_j. We wish to emphasize that it is *only* in the attention heads where the embeddings of different tokens can affect each other.

We remark that with our hypothetical "looking for articles related to nouns" attention wouldn't work fully by itself to figure out which nouns have definite or non-definite articles in them. This is due to the fact that even in our simple example sentence "Then the bear waited for the door to open with a serendipitous smile." there are situations like "a serendipitous smile" where the article and the noun are separated with a whole word. (And actually by quite many tokens.) Furthermore the sentence we study has several articles, and it is nontrivial to match them to their nouns. So in practice this kind of article detection would probably require the combined effort of several attention heads together with the fully connected neural network of a transformer block.

Now the final question in this subsection is about how fantastical was our example attention head that is "looking for articles related to nouns". The study of attention heads is a field of its own, see e.g. (Goldberg 2019), (N. F. Liu et al. 2019), (Rogers, Kovaleva, and Rumshisky 2020), (Tenney et al. 2019), (van Schijndel, Mueller, and Linzen 2019), (Warstadt et al. 2019), (Wu et al. 2020), but suffice it to say that various linguistical concepts have been identified in the attention layers, especially for the BERT model. The division of tasks does not seem to be so cleanly divided between different attention heads (or even different transformer blocks at times), but there do seem to be attention heads whose tasks can be mapped to (or at least seem to be correlated with) the linguistical concepts used by humans. In particular, (Htut et al. 2019) and (Clark et al. 2019) report that there are BERT attention heads that clearly attend to certain identifiable syntactic positions. See also Figure 8 from (Kovaleva et al. 2019), where Kovaleva et al. describe different "types" of attention heads.

We refer to (Rogers, Kovaleva, and Rumshisky 2020) and the references therein for a more thorough exposition on the topic of BERT attention interpretation.

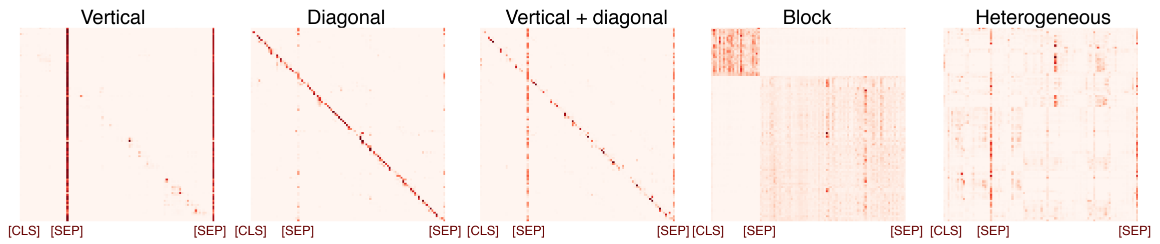


Figure 8: Various types of attention patterns as described by (Kovaleva et al. 2019). Image copied from the source files of the article.

3.2.6 Getting clever with linear algebra – multi-head attention in even more detail

We don't want dissect the complete technical implementation of the multi-head attention here, but we want to say a few words for those who will go on and read the original Vaswani et al. 2017 for more details. In there the organization of the value matrices in particular is a bit more complex than what we let on in here. Part of the reason is that there is *a lot* of linear algebra optimization going on to make this all run fast. As we noted, the attention mechanism was reduced to various matrix products and softmaxes, and these can be cast and performed in very efficient way with e.g. Matlab, Numpy or other even more powerful CUDA libraries. But there is even more things going on, so the implementation details are less straightforward.

A single example we point out here is that we are taking a weighted sum of the value vectors. These value vectors were created by the multiplying embedding vectors with two matrices, one that has dimension $d_{\text{model}} \times d_Q$ and thus "drops" the dimension of the embedding vector from d_{model} to d_Q and another that "lifts" it back up. Weighted sum is a linear operation, meaning that we can actually do the weighted sum in the smaller-dimensional space d_Q and then map the results back to the embedding space with the "lifting" matrix. This is computationally quite beneficial.

Note also that many of the dimensions "line up" nicely here – for example in the BERT model we have 12 attention heads that each have $d_Q = 64$, and $12 \cdot 64 = 768 = d_{\text{model}}$. These kinds of consistent sizes of vectors and matrices can be very useful in getting a full utilization of the immensely parallelizable CUDA framework.

3.2.7 The decoder side of things – masked attention and cross attention

As we promised, we now take a quick peek at the *decoder* part of Figure 4. In particular, we take a look on how the *masked* self-attention differs from the vanilla attention mechanism and how the so called *cross*-attention works. We won't go into detail about GPT architectures, but suffice it to say that the main difference to the BERT architecture is the usage of masked multi-head attention instead of the basic one.

We start by remarking on the context here. The original transformer was built for machine translation, and it was a job for the encoder to "read" the incoming piece of text and a job for the decoder to produce the translation. The reading of the incoming piece of text can be done very well in parallel by the encoder, but the generation of the translated text is done from left to right one word at a time. Instead of turning back to RNNs for this task, the idea is that we use the same attention mechanisms as before, but simply *mask* all the tokens to the right of the current embedding vector so that they cannot influence the deductions. One way is to set the dot product values to $-\infty$ which the softmax will set to 0, and thus the value vectors related to those embedding vectors will be zeroed out. Note that even though this makes the system more sequential than parallel, at training time we can do a lot of parallelization by taking a training sentence and turning it into several training data by masking more and more of the words at once. For example, from the English to Finnish sentence pair ("Bears are hairy.", "Karhut ovat karvaisia.") we can create the training data points of ("Bears are hairy.", "[?]"), ("Bears are hairy.", "Karhut [?]"), ("Bears are hairy.", "Karhut ovat [?]") and ("Bears are hairy.", "Karhut ovat karvaisia[?]") where in each pair the "[?]" symbol represents the token that the system should predict.

The *cross*-attention here is contra to the attention mechanism that we studied within the encoder which is called *self*-attention. Here the idea is that when working through the QKV attention, we do not need to form the query-, key- and value vectors from the same set of embeddings! In the original transformer paper, as we are working on translation, we have on one hand a complete sentence in some language and on the other hand a partial translation of that sentence in another language. But instead of texts in two languages, we could have two texts in the same language whose interrelations we are trying to parse, or a piece of vectorized audio and a text, or images and video, and so on. Essentially kind of data that is

turned into embedding vectors goes here.

So for now suppose we have embedding vectors $E_1, \ldots, E_K$ from domain 1 and embedding vectors $\hat{E}_1, \ldots \hat{E}_N$ from domain 2, and that the aim of the attention mechanism is to update the domain 1 vectors based on the domain 2 vectors. Note that the vectors do not need to have the same dimension. What happens in the cross-attention is that the Query meachanism maps $E_j \mapsto Q_j \in \mathbb{R}^{d_Q}$ and the Key mechanism maps $\hat{E}_i \mapsto K_i \in \mathbb{R}^{d_K}$ with $d_K = d_Q$. Thus mapping these vectors from two different domains to something that we can compare with the softmaxed dot product! Then the role of the Value mechanism is to map $\hat{E}_i \mapsto \Delta\hat{E}_i$ in such a way that $\hat{E}_i$ is an absolute position in domain 2 while $\Delta\hat{E}_i$ is a direction in domain 1.[14] Thus the query-key mechanism enables the comparison of embeddings of different domains, and the value vector enables the domain 2 to affect vectors in domain 1.

3.2.8 Putting it all together – a second view to the high-level architecture

After all the details in mind, we can look at the big picture again with a better understanding. We wish to highlight here a few things.

The first of these is that the attention mechanism is agnostic to the *order* and *amount* of inputs. If we have input encodings E_0, E_1 and E_2, then feeding in to the self-attention mechanism the sequence (E_0, E_1, E_2) will produce the output (E_0', E_1', E_2'), while an input sequence (E_2, E_0, E_1) would produce the output (E_2', E_0', E_1'). This is due to the fact that the attention mechanism updates each embedding based on a weighted sum of the value vectors, and these weights were calculated from a softmaxed dot product – if you read through the details again, nothing changes here if you permutate the orderings. Furthermore the feed-forward neural network is the same neural net applied to each output embedding simultaneously but individually, so there is no order present here either. So the system really is a "bag of words" system without the positional encoding vectors added to the input embeddings in the very beginning. Each transformer block is completely order agnostic and thus the positional embeddings are crucial.

14. Depending on the various dimensions involved, this would probably be again split to two matrix operations like in the self-attention mechanism.

The second thing we wish to highlight is that when training or using this system, besides the amount of positional encoding vectors there is no inherent limitation on the amount of tokens that can be fed to the network. With varying amount of tokens the attention layer will have a varying amount of dot products to weigh on. For the original transformer paper the encoding vectors were manually created trigonometric functions, so in principle we could use any number of them. The problem that emerges is that the amount of attention dot products grows $\mathcal{O}(N^2)$ and together with it the backpropagation calculations[15]. Also, for the system to learn what the various positional encoding vectors signify, it needs to have a good collection of them presented during training.[16] Meaning that if we wish to do inference on embedding sequences of length 10M, then we better have trained the system with many sequences of comparable length. Note that e.g. in BERT the positional embeddings are learned at training time, so the so called "window width" of the system, e.g. the maximum length of the input embedding sequence, is fixed at 512.

A final thing we wish to mention here is that this fluidity in position-freeness also gives us the chance to train the system with different length of sequences. As we will see, BERT model was trained 90% of the time with text of length 128 tokens and 10% of the time with length 512 tokens. The only place where these training styles have any effect will be at the position embedding vectors, as we will also see.

3.3 The BERT model architecture

Now we move on to the architecture of the BERT transformer (**B**idirectional **E**ncoder **R**epresentations from **T**ransformers) introduced in Devlin et al. 2019. Their paper lists several main goals for their work, for us the most crucial ones are as follows:

1. Create an NLP model whose base architecture does not need to be altered for downstream tasks. The BERT model has been pre-trained with quite a large effort, but it

15. Though there are other attention mechanisms besides the KVD attention. Some of these are particularly targeted to avoid this quadraticity.

16. In theory this is not completely true. We could, in theory, build a positional encoding system where e.g. the concept of "vector B is the next one from vector A" is extracted with one specific linear operator that the network can learn. But in practice this might be a challenging task and its performance unclear.

can be fine-tuned to various specific tasks in the matter of a few hours.

2. The authors wish to show the benefits of bidirectionality, i.e. not limiting the model with e.g. only seeing tokens to the left of the current position. So in particular, we'll be using the encoder part of the original transformer architecture, not the decoder part.

3.3.1 Overview of the architecture

The bird's eye view of the BERT core architecture is as in Figure 9, see again Figure 5 for details of the encoder block. The main architecture of BERT is *almost* identical to the original Transformer encoder part, but there are a few differences. We'll be describing the BERT architecture by explaining the differences to the original transformer encoder.

1. As we briefly mentioned, the tokenizer used is different than in the original Transformer. In BERT we use the WordPiece method instead of BPE, but we won't go into detail about the differences in this thesis and refer an interested reader to Prince 2023. A crucial difference here is that BERT will add to any tokenization special tokens; the [CLS] token at the start and a [SEP] token at the end. The purpose of the [CLS] token is to function as a "summary" of the whole text while the [SEP] operates as a segment end marker. We'll discuss these in more detail in Section 3.3.3
2. The positional encodings are learned vectors instead of hardcoded sinusoidal ones. There are 512 positional encodings, and we will look at them in more detail in Section 4.2.3.
3. Besides the positional encodings, we also add to each input embedding one of two possible *segment embedding vectors*. The purpose of these is related to the training of BERT – we'll go in to more detail in Section 3.3.3, but in short BERT has a 'mode of operation' where it predicts if two sentences are next to each other in a piece of text or not. The segment embeddings identify to the model the different sentences to compare.
4. The BERT model uses the same architecture of transformer blocks as the original transformer paper. We have 12 transformer blocks, each containing a 12-layered multi-attention head. The d_{model} dimension for the standard BERT is 768, the key and query vectors are 64-dimensional and the value vector generation is also factored through

$\mathbb{R}^{64}$.

5. After the final transformer block there is the "head". This is usually a simple fully connected neural network layer that is changed and trained specifically for a given downstream task. It is usually connected either to the final embedding corresponding to the [CLS] token or to all of the embeddings in the final layer.

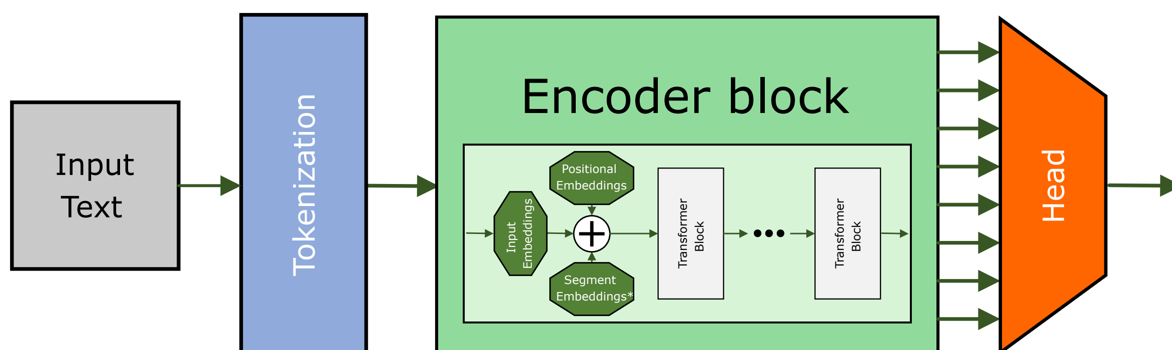


Figure 9: The BERT main architecture.

So with that we see that the BERT model really is mostly just the encoder part of the original transformer architecture. We'll next turn to explore the BERT architecture a bit with bespoke model analysis tools.

3.3.2 On the weights of BERT

One way to get a technical overview of the architecture of BERT is with the `torchinfo` library. Running the following code[17] with various `depth` and `verbose` parameters will give information on varying resolutions.

```
from transformers import BertModel
from torchinfo import summary

model = BertModel.from_pretrained('bert-base-cased')
print(summary(model, depth=3, verbose=1))
```

With the parameters `depth=3, verbose=1` the printout looks something like below, though we've added comments to the end of the lines:

17. Note that this will download the BERT base model locally, taking up about 416Mb of space.

```
==============================================================
Layer (type:depth-idx)                    Param #
==============================================================
BertModel                                 --
+-BertEmbeddings: 1-1                     --
|    +-Embedding: 2-1                     22,268,928 # Input embs
|    +-Embedding: 2-2                     393,216 # Position embs
|    +-Embedding: 2-3                     1,536 # Segment embs
|    +-LayerNorm: 2-4                     1,536
|    +-Dropout: 2-5                       --
+-BertEncoder: 1-2                        -- # Encoder
|    +-ModuleList: 2-6                    --
|    |    +-BertLayer: 3-1                7,087,872 # 1:st Transf. block
|    |    ...
|    |    +-BertLayer: 3-12               7,087,872 # 12:th Transf. block
+-BertPooler: 1-3                         -- # Head
|    +-Linear: 2-7                        590,592
|    +-Tanh: 2-8                          --
==============================================================
Total params: 108,310,272
Trainable params: 108,310,272
Non-trainable params: 0
==============================================================
```

With a vocabulary of 30k tokens, each with a 768-dimensional input emebedding, we see the $30k \cdot 768 \approx 22M$ input embedding parameters at `Embedding 2-1`, the $512 \cdot 768 \approx 400k$ positional embedding weights, and the $2 \cdot 768 = 1536$ weights of the segment embeddings. The various normalization layers also use a set of 768 weight and 768 bias weights. In each of the `BertLayer3-Xs` we see 7M weights, and we can open up their distribution by increasing the `depth` and `verbosity` parameters. By setting `depth=4, verbose=1` we get the model counts at the following resolution.

```
   ...
|    +-BertLayer: 3-1                     --
|    |    |    +-BertAttention: 4-1              2,363,904
|    |    |    +-BertIntermediate: 4-2           2,362,368
|    |    |    +-BertOutput: 4-3                 2,361,600
   ...
```

Here the `BertIntermediate` and `BertOutput` are the two layers of the feed-forward neural network with a hidden layer of 3072 neurons. (So we have $2,361,600 = 3,072 \cdot 768$ weights for the output layer, and with the 768 bias vectors of the hidden layer, $2,362,368 = 3,072 \cdot 768 + 768$ weights for the intermediate layer.)

Increasing the verbosity to 2 gives even more detail about the attention layer.

```
...
+-BertEncoder: 1-2                                 --
|    +-layer.0.attention.self.query.weight         +-589,824
|    +-layer.0.attention.self.query.bias           +-768
|    +-layer.0.attention.self.key.weight           +-589,824
```

```
|    +-layer.0.attention.self.key.bias            +-768
|    +-layer.0.attention.self.value.weight        +-589,824
|    +-layer.0.attention.self.value.bias          +-768

|    +-layer.0.attention.output.dense.weight      +-589,824
|    +-layer.0.attention.output.dense.bias        +-768
|    +-layer.0.attention.output.LayerNorm.weight  +-768
|    +-layer.0.attention.output.LayerNorm.bias    +-768
...
```

We have a 12-head multi-attention system with $d_Q = 64$. Thus for each of the attention heads we have a Query matrix of size $64 \cdot 768 = 49,152$, and $49,152 \cdot 12 = 589,824$. There's also a learned query bias vector that we add to the result. Same holds for the key vectors. For the value vectors, recall that we split the mapping of embedding vectors to value vectors into two matrices of dimensions 768×64 and 64×768. We see this split here as having separate `attention.self.value` and `attention.ouput.dense` layers.

3.3.3 On the training of BERT

Besides the architecture breakthrough of transformers, the field of NLP somewhat simultaneously got a large boost from improved methods of pretraining language models. In particular, the ULMFiT (Howard and Ruder 2018) approach to language model training was important. The idea of the ULMFiT approach is somewhat similar to the idea of transfer learning. In ULMFiT we train a language model in three phases:

1. Pre-training. This is the heavy training phase where we use e.g. all of the CommonCrawl or Wikipedia data to train the model to understand a general idea of language.
2. Domain fine-tuning. Here we take the base model and extend the training with a corpus from the domain of application.
3. Task fine-tuning. Finally here we train the model to the specific task we need to understand. With the BERT architecture, recall Figure 9, this usually involves the change of the final head layer to something task-specific.

When we talk about *the* training of BERT, we refer to the most compuationally expensive pre-training part, though the task fine-tuning is identical in technical nature.

One of the great issues in ML is the fact that we usually need a lot of training data, and it is often very laborous to annotate the ground truth to training data. A crucial part of the

success of transformer models is the applicability of so called Masked Language Modeling (MLM)[18], in which training data is generated by having a language model predict a masked word from existing text. This MLM approach means that any large corpus of text can be turned into a considerable amount of training data.

Besides the MLM task, the BERT training contains also something called *next sentence prediction*. In this task we again take long pieces of text from a corpus and split them in two. We then combine these halfs with either the original pair or some other pair and ask the ML model to predict if the two given text pieces were originally next to each other.

The BERT training does both of these tasks, the MLM task and the next sentence prediction task, "in one go". The idea is that we take sentence pair ("The bear was majestic.", "We took a lot of pictures of it."), tokenize the texts and extend the tokenization with special tokens as follows:

[CLS] The bear was majestic. [SEP] We took a lot of pictures of it.[SEP]

Furthermore besides adding the special [CLS] and [SEP] tokens, we also add to the token embeddings the sentence type embeddings shown in the diagram in Figure 5. There are two sentence type embeddings, and the first one is applied to all the tokens up to and including the first [SEP] token, while the latter is applied to the rest of the tokens. Thus we 'inform' the model in two ways where the separation of the sentences is – both with the [SEP] token separating them and the segment vectors labeling the parts.

The next sentence prediction is read from a simple binary classification head attached to the final embedding of the [CLS] token. In particular, after the BERT pre-training the [CLS] token is fine-tuned for this task and this task only. The name "CLS" stands for *classification*, but the authors recommend fine tuning whenever the [CLS] embedding is used for a classification task since at the base version it is not specialized for differentiation. Regardless, the next-sentence prediction task does 'force' the [CLS] token to be imbued with general level information of the whole text.

With the next sentence prediction task set up, we then alter the tokenized input text

18. Also known as *the Cloze task* in the literature.

[CLS] The bear was majestic. [SEP] We took a lot of pictures of it.[SEP]

with the MLM task. This is done by selecting 15% of the tokens for masking. For these tokens we replace the token in question with a special [MASK] token 80% of the time, replace it with a random token 10% of the time and leave it to be 10% of the time. Regardless of which of these three actions we perform, the model is then tasked, for this training datum, to predict the original word. The reasons for not only using the [MASK] tokens are discussed in detail both in (Vaswani et al. 2017) and (Rogers, Kovaleva, and Rumshisky 2020), so we won't go into them here. So our final training datum might look like this:

[CLS] The **[MASK]** was majestic**duck** [SEP] We took **[MASK]** lot of **pictures** of it.[SEP]

and the model would need to predict the emboldened tokens and that the sentences are in fact from a text where they are next to each other.

4 A study of the structure of BERT embedding vectors

As we saw in Chapter 3, the BERT architecture we study can be seen as a system that produces a sequence of embeddings that encode more and more complex information the further we advance.[1] We start with a sequence of context-free embeddings of individual tokens and end up with embeddings whose purpose is to encode complex semantic content of the whole input text. In this Chapter our aim is to study in detail how the embedding vectors look like at various levels of the encoder with a special focus on the final output layer. We're also guided in this Chapter by the research questions 1 through 3 stated in the introduction; these ask in a few different ways if we can observe differerences in these various embeddings.

After a quick look into the tokenization part of the system, we'll continue by looking at the context-free embeddings that are the input to the actual encoder stack. When then look at how these embeddings evolve through the 12 transformer blocks of the system before turning to study the final output embeddings. After that we turn to study the semantic content of the embedding vectors and if the various embedding spaces have conceptual relations. We also look how and if any semantic properties persist in the "evolution" of an embedding through the transformer blocks. We finish the Chapter with a detailed look at some application possibilities of the embeddings.

Throughout the chapter we'll be using the *cosine similarity* of two vectors as a measure of their similarity. The cosine similarity is simply the cosine of the angle between the two vectors; cosine similarity of 1 means that they have the same direction, cosine similarity of 0 means that they are orthogonal and cosine similarity of -1 means that they are pointing at opposite directions. It is in no way obvious that this metric would be the best one to measure embedding vector similarities, but it is one of the most widely used ones. We refer to Rogers, Kovaleva, and Rumshisky 2020 for further discussion on the suitability of various metrics.

1. This evolution of the embeddings through the transformer blocks is sometimes called *the residual stream* of a transformer in the literature.

4.1 A few words on the research methodology

Our methodology, if it should even be called that, has no name. Not because it is new, unique or original, but because it is so old and fundamental that it requires no special name. It is simply that of starting with curiosity, planning an experiment, trying to guess the results before running it, and then updating our worldview based on the discrepancy between our a priori guess and how the world turned out to be. After this we repeat. We don't aim for a particular goal, but look at the world through experiments and see where they lead us. In the end we report the most fascinating path of results.

We feel that it would be intellectually dishonest to name and claim here a particular *research methodology* like "exploratory data analysis". Our approch has been nothing if not exploratory, but we feel that fixating on a single title would be a disservice to the reader. Many of the research methodologies have a somewhat fixed format and definition. Though being restricting, such standardization has several benefits and e.g. supports the replicability and extendability of scientific research. This work does not, however, aim to be reproducible. It aims to be a master's thesis. It fulfills its purpose primarily in the process of it being written as this is when our main goal of learning the topic is being fulfilled – the text you are reading is just our witness report after the fact. We are furthermore devote believers of Feynman's idea that a great way to learn is to try to do everything by yourself before checking how it's supposed to be done. It is these ideas that drive our choice of a research methodology. If pressed, we would name this simply as the base scientific method.

So in this thesis we have set out to better understand the internal model transformers. Our avenue of attack was to focus on BERTs embedding spaces, which first lead us to study the transformer architecture in depth. After that the task turned into starting to analyze how the embedding vectors look like with various concrete inputs. We fed the system both synthetic and real-world data and tried to find interesting points to look and probe at. Many approaches and ideas turned out to be either not interesting, or somewhat interesting but requiring too much exposition to explain to be worth included in this particular thesis with its limited scope. But some approaches bore fruit and these are reported here in the next sections. We will return to the topic of research methodology in a sort of post mortem analysis in Section 5.1.

4.2 Embeddings before the encoder

In this section we start with a more detailed analysis of both the tokenization and the triplet of *input embeddings*, *positional embeddings* and *sentence type embeddings*. We then look at how the three types of pre-encoder embedding vectors relate to each other. Note that the crucial point in this Section is that these vectors are all *context-free*, meaning that they do not depend on surrounding data. This means that, unlike with the latter encodings, we can study the embeddings in each set of embedding types independently from any other embedding.

Recall that with the BERT architecture all of these embedding vectors are learned parameters. The vocabulary size of BERT is 30k, and each of those 30k tokens has their own input embedding vector of dimension 768. This means that the matrix of all 30k input embeddings has around 23M values[2]. Then there are 512 positional embedding vectors of length 768, netting us around 39k parameter values. Finally we have two sentence type embedding vectors, both with length 768.

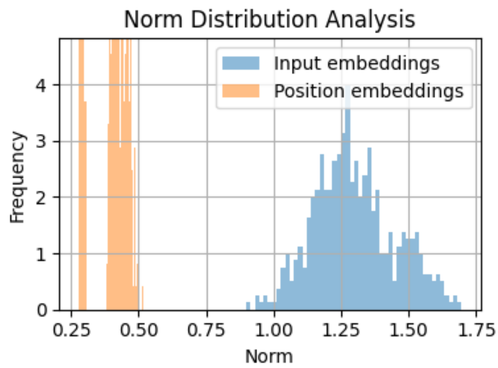
Now, getting e.g. the cosine similarities between any two input embedding vectors results in a matrix of 900M values – as a numpy matrix this takes around 3.5 Gb of memory on our laptop. This means that most calculations can still be brute-forced, but they can be quite compute-intensive.

To start off our analysis, we have in Figure 10 plotted the basic statistical distributions of the input embeddings and the positional encodings embeddings. We've not included the two sentence type embedding vectors, but their norms are 0.70 and 0.72, their cosine similarity is -0.065, their dot product is -0.033 and their Euclidean distance is 1.04.

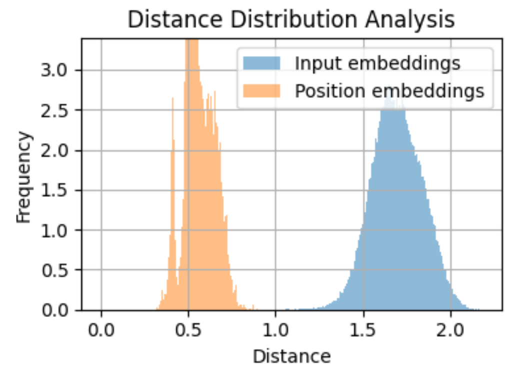
What we note in Figure 10 is that the input embeddings and positional embeddings seem to have quite different structures and distributions. The positional embeddings have norms concentrated to just below 0.5, with the input embeddings have a wider distribution of norms between 1.0 and 1.75. For the sentence type embeddings we had norms around 0.7, so they are also separated from the other types. We see similar properties in other statistics as well.

The differences between these two embedding styles are also visible in our projection tools

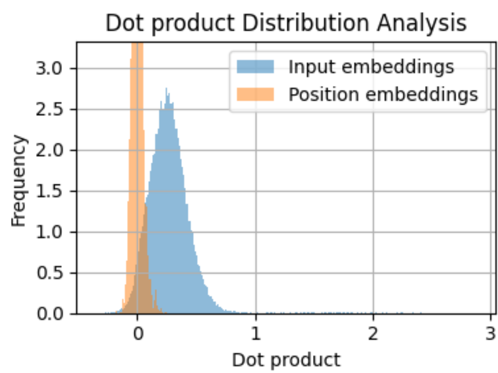
2. See again Section 3.3.2.



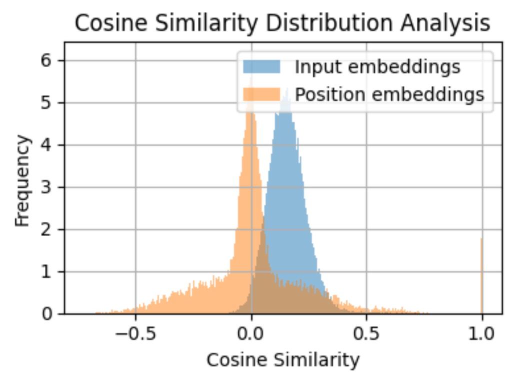
(a) Euclidean norm distribution



(b) Euclidean distance distribution



(c) Dot product distribution



(d) Cosine similarity distribution

Figure 10: Distribution visualizations of the context-free embeddings and the positional embedding vectors of the BERT model.

and PCA dimension analysis, as shown in Figure 11. In here we see several things:

(a) The PCA projection is the most geometry preserving of our projections, and here we see both a distinct distribution between the two embedding types, and a non-trivial shape of the input embedding vectors. There seems to be a "Pacman"-shaped main component in the input embeddings together with a towering offshoot at the top. The positional embeddings are much more narrowly distributed as we saw from the charts in Figure 10, so the input embeddings dominate the PCA projection.[3]

(b) From the dimensionality analysis we see that the curve of the positional embeddings increases very fast compared to the input embeddings, implying that they are much more focused on a lower-dimensional subset of the embedding space. This is to be expected as the concept they encode - position - has a lot less information than the input embeddings that encode semantic meaning of (almost) all words. We'll do more specific analysis of the positional encodings later on in Section 4.2.3.

(c) & (d) Both of the t-SNE and UMAP projections that aim at presenting the topology but not the geometry of the data are able to spot that the positional embeddings have two components, one of which is possibly mixed in with the input embeddings. Both of the methods also seem to hint at a long string-like part in both embedding sets. We'll return to these notions in sections 4.2.2 and 4.2.3.

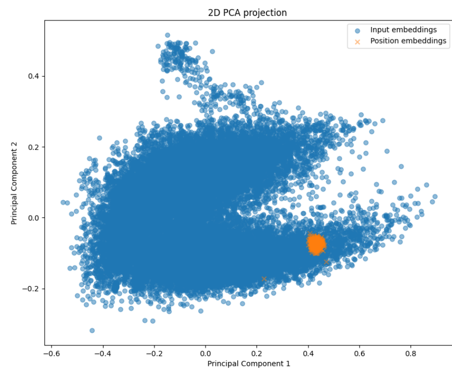
So there are observable geometrical differences between the different types of embeddings. Our first research question asked if we can detect the difference between these types of vectors, which we have now shown to be possible.

We'll next study the various embedding vectors in more detail.

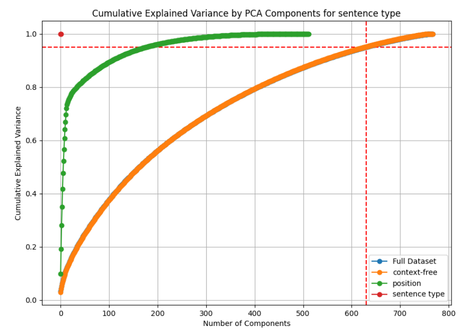
4.2.1 The tokenization

The tokenizer and its results are not embeddings, but we feel that they fit naturally in this subsection as it is the tokenizer that 'decides' which words or parts of words 'deserve' their own input embedding.

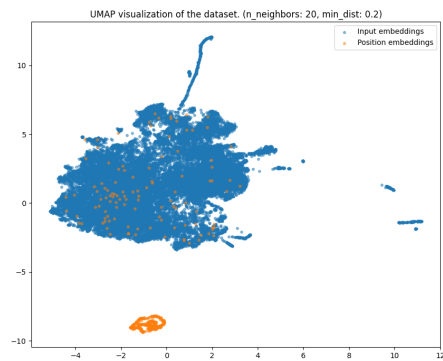
3. Recall that the PCA projection to 2D looks for the two dominant directions of variance and is not aware of any labeling we might have for the data.



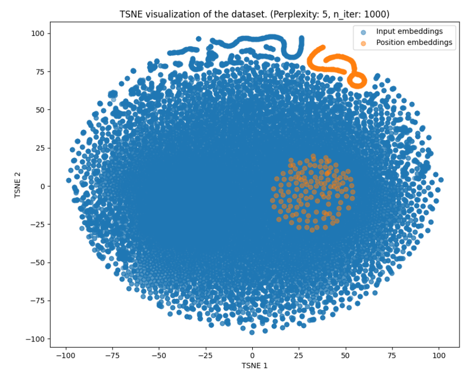
(a) PCA projection



(b) PCA dimension analysis



(c) UMAP projection



(d) t-SNE projection

Figure 11: Projection visualizations and dimensional analysis of the context-free embeddings and the positional embedding vectors of the BERT model.

We can quite easily study the tokenizer contents.

```python
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('bert-base-cased')
for j in range(30000):
    print(j, tokenizer.convert_ids_to_tokens([j]))
```

This will print 30k lines of text so we recommend directing the output to e.g. a file. What we see in the tokens is as follows:

- 0: The special [PAD] token.
- 1–99: The special [unusedX] tokens.
- 100: The special token [UNK] for unknown token.
- 101: The special classification [CLS] token.
- 102: The special [SEP] token.
- 103: The special [MASK] token.
- 104 & 105: Two more special [unused] tokens.
- 106–199: Pretty much the ascii character set.
- 200–1102: Various more exotic single character tokens.
- 1103: 'the'
- 1104: 'of'
- 1105: 'and'
- 1106–28995: The rest of the tokens in no immediately obvious order, though common short words seem to appear higher on the list.

As noted in Section 3.2.2, about 22% of the tokens start with "##", i.e. are the endings of subwords. Furthermore about 77% of the tokens are alphanumeric[4].

4.2.2 The input embedding vectors

As we saw in Figures 10 and 11, the input embeddings are not uniformly distributed to the unit sphere or cube in $\mathbb{R}^{768}$. Instead they do seem to have some complex geometry. Let's

4. See https://github.com/ramiluisto/NLP_toybox/blob/main/notebooks/vocabulary_comparison.ipynb for further comparisons.

first look at some extremal values and look for the extremal pairs in the cosine similarities between input embedding pairs. Due to some page count comments from our advisors, we've decided to omit the full list of 900M pairs of tokens and their pairwise cosine similarities, and simply show some of the token pairs with the more extremal cosine similarities. See Table 2.

Token 1	Token 2	Cosine similarity
£1	£2	0.9328
northward	southward	0.9095
197	196	0.8891
1986	1985	0.8876
southbound	northbound	0.8866
⋮	⋮	⋮
island	Steele	7.4506E-09
J	nets	5.5879E-09
knee	Mali	3.7253E-09
bed	##nect	0.0000E+00
arms	Via	-1.8626E-09
Southern	fears	-7.4506E-09
⋮	⋮	⋮
and	Quarterfinals	-0.3134
and	vols	-0.3179
and	smirked	-0.3181
and	Nope	-0.3187

Table 2: Table showing the cosine similarities of various context-free embeddings with special tokens excluded.

In the most negative cosine similarities the words "and" and "the" are very much dominating. So the model has learned for these tokens in particular that in some cases they are strongly anticorrelated with various other words.

We can also try to see where the embeddings are centered at by taking the average of all the input embeddings. This should not be considered to be an operation with any semantic meaning in finding "the most average word". We find it and its relation to other embeddings interesting regardless as a sort of description of the "mean and variance" of the distribution of embeddings. In Table 3 we've listed some of the alphanumeric tokens that are closest to the average input embedding with respect to various metrics.

Token	Euclidean Distance	Abs. Cosine Similarity	Abs. Dot Product	Cosine Similarity	Dot Product	Euclidean Norm
Archived	**1.64**	0.38	0.33	0.38	0.33	**1.77**
versa	1.54	0.39	0.33	0.39	0.33	1.67
facilitate	**0.87**	0.48	0.24	0.48	0.24	1.00
innovative	0.92	0.44	0.22	0.44	0.22	1.02
Keyboards	1.06	**0.60**	0.38	**0.60**	0.38	1.28
Freyja	1.18	0.59	0.41	0.59	0.41	1.40
ground	1.32	**0.00**	**0.00**	0.00	0.00	1.22
building	1.17	0.00	0.00	-0.00	-0.00	1.05
Wouldn	1.29	0.57	**0.43**	0.57	**0.43**	1.50
trillion	1.50	0.51	0.43	0.51	**0.43**	1.69
and	1.20	0.37	0.17	**-0.37**	**-0.17**	0.92
the	1.11	0.31	0.13	-0.31	-0.13	**0.85**

Table 3: Table showing some of the extremal token values compared to the average embedding. In these extremal values we've included only the tokens that contain only latin alphabet symbols and have length of at least 3. Largest and smallest value of each column has been highlighted.

For context on the size and distribution of the column values in Table 3 we've plotted histograms of their full distribution in Figure 12. In the histograms, excepting the norm, each histogram plots the values of the particular metric between the average embedding and all the other input embeddings. The norm distribution is simply the norms of the input embeddings,

provided for context. The mean of each distribution is included in the legend.

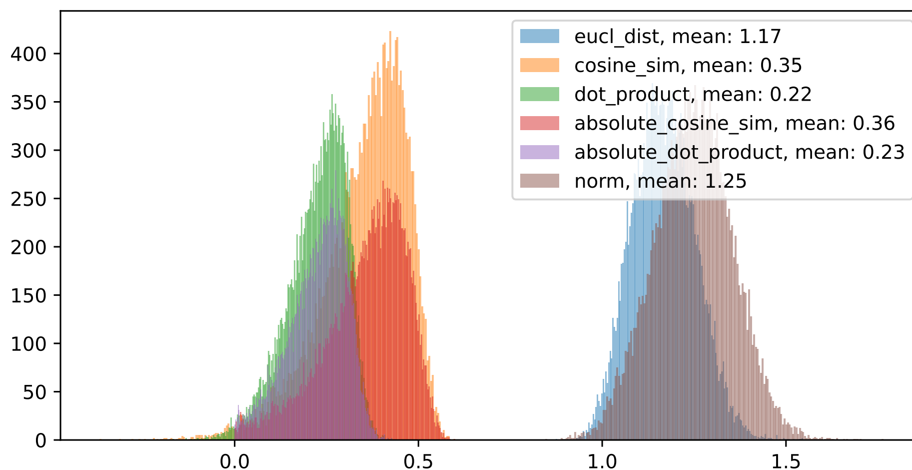


Figure 12: Histograms on the distributions of the measures studied in Table 3.

We can thus conclude that even from the point of view of statistical distributions of the pairwise similarities of the embedding vectors, and their relations to the average vector, the set of context-free embedding vectors exhibit complex structure.

4.2.3 The positional encoding vectors

As we noted in Section 3.2.4, positional encodings can be designed 'by hand', and e.g. the original transformer uses encodings based on trigonometric functions.

For us and BERT, however, these positional encodings are learned properties, i.e. they are initialized as random weights and then learned during the training process. We've plotted the positional encodings and their cosine similarities in Figure 13 – compare this to Figure 6 in Section 3.2.4.

There is a lot to unpack Figure 13. We first of all note that, as expected, there is no visually obvious structure to the positional encodings. This is natural as a selection of e.g. the sinusoidial functions or a standard basis of 768 is a very human approach. There are an uncountable amount of bases we can take for any given vector space, and the neural network has no preference to choose a basis that is quick for us to grasp from images. Though we do

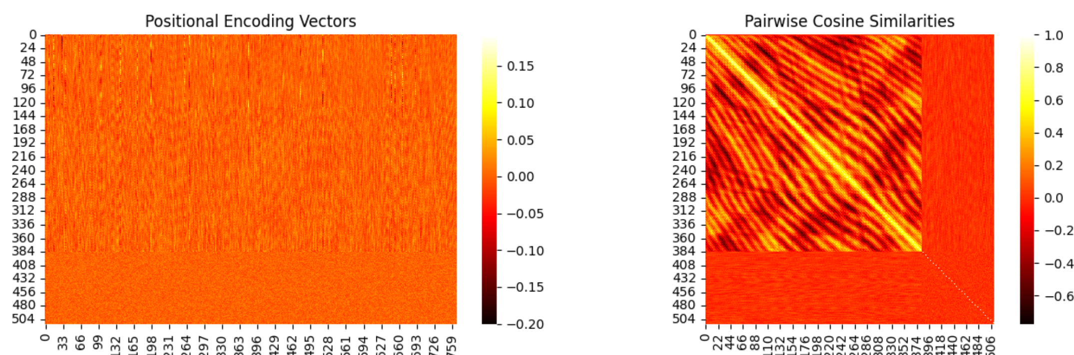


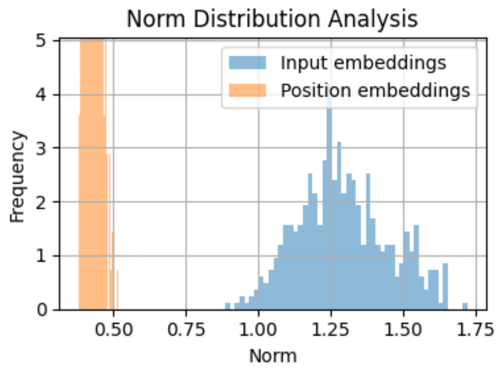
Figure 13: BERTs positional encodings.

see some geometrical wavy patterns in the vectors, which is a sign of some sort of continuous change of parameters over the position.

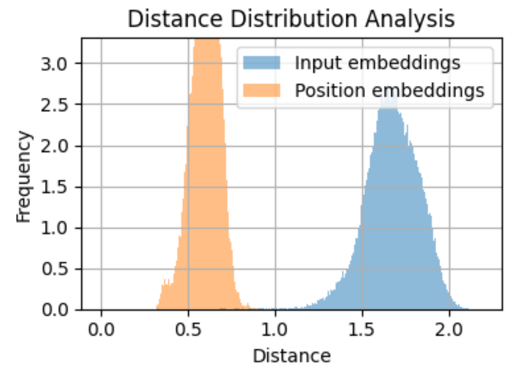
The second thing we note is that there seems to be a phase change in the structure of these vectors at position $384 = 256 + 128 = 768/2$. Some ad hoc entropy and distribution measurements seem to suggest that these values are randomly distributed, but it is hard to say. For now we point out that if we look at the distribution of the values and the pairwise cosine similarities of these positional encodings corresponding to the latter positions, they do not seem to exhibit any clear structure. As noted in Section 3.3.3, during the training of BERT they used training text samples of 128 tokens 90% of the time and text samples of 512 tokens 10% of the time. We do feel that we can also see a small phase shift around the 128:th positional encoding vector in the pairwise cosine similarities in Figure 13, but this does not explain the change at positions 384 onwards. In Figure 14 we've replotted the charts in Figure 10 with only the first 384 positional encodings included. We note in particular the shape of histogram of the pairwise cosine similarities; with all positional embedding vectors we clearly have a combination of the flatter distribution we see with the first 384 embeddings superimposed with a sharp narrow distribution around zero, which should be expected when taking cosine similarities between random high-dimensional vectors.

Furthermore, comparing the situation to the FinBERT[5] model, see Figure 15, we see that there is no similar cutoff, though we again observe a slight phase shift at the 128:th positional

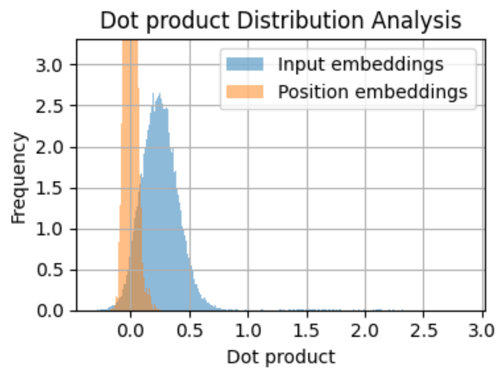
5. A transformer model with identical architecture and training schema to BERT, but trained with Finnish text. See Virtanen et al. 2019.



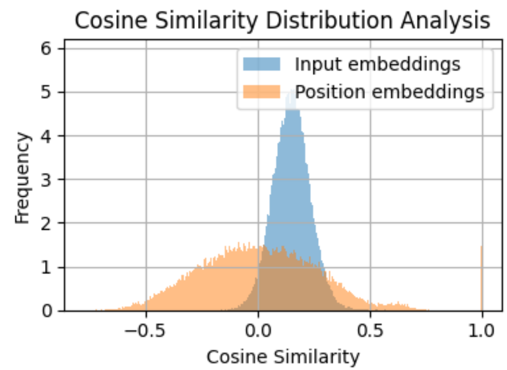
(a) Euclidean norm distribution



(b) Euclidean distance distribution



(c) Dot product distribution



(d) Cosine similarity distribution

Figure 14: Distribution visualizations of the context-free embeddings and the positional embedding vectors of the BERT model with only the 384 positional encodings. Compare this to Figure 10.

encoding. This is likely again due to the 128-512 split in the training schema.

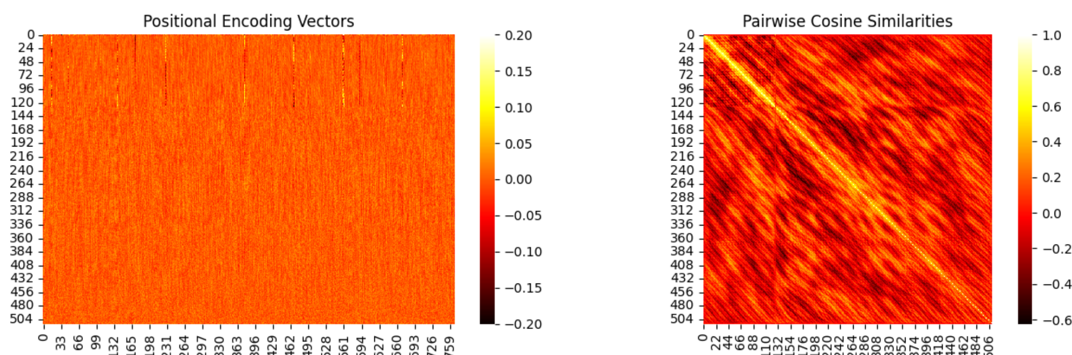


Figure 15: Positional encoding vectors and their cosine similarities in the FinBERT model.

The third thing we note is that the pairwise cosine similarities of the first 384 positional encoding vectors exhibit very strong patterns. Not easily visible from the heatmap of Figure 13 is that the positional encodings corresponding to the first position has very low cosine similarities with the rest of the positional encodings. This is quite natural as the first token in BERT is *always* the special [CLS] token that aims to capture the semantic content of the whole text. For the rest of the positional embeddings we have in Figure 16 plots of cosine similarity of a single positional embedding against the 128 first positional embeddings – i.e. this plots the first 128 values of the rows 1-10 of the heatmap in Figure 13. What we notice is that each of the positional encodings has a very similar structure. They are quite dissimilar to previous positional encoding, very similar to themselves of course, and then the similarity is sort of "oscilating downwards".

We can get a better image here if we change the plotting by shifting these graphs backwards, i.e. plotting the *relative* similarity of a given positional encoding to the encodings coming after it. See Figure 17. Thus we see that there is some non-trivial structure in the cosine similarities of these positional embeddings. We refer to Wang et al. 2020 for further discussion on the details of BERT position encodings.

Next we turn to the *geometry* of the positional encodings. Figure 18 had the UMAP and t-SNE projections of the embeddings with the positions 384-511 both included and excluded. What we notice is both that there seems to be strong 1-dimensional structures in the first 384 embeddings and that the last 128 embeddings (which we call the "bad" embeddings in the

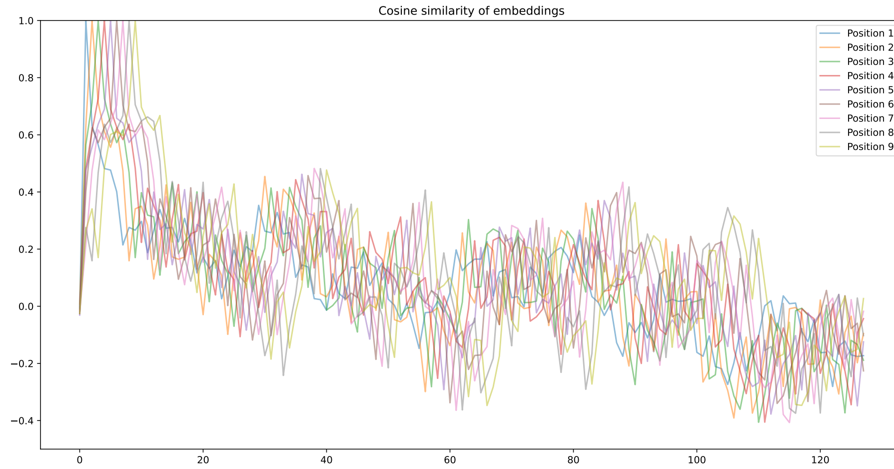


Figure 16: The cosine similarities of the first 10 non-[CLS] token positional embeddings against the next 128 positional encodings.

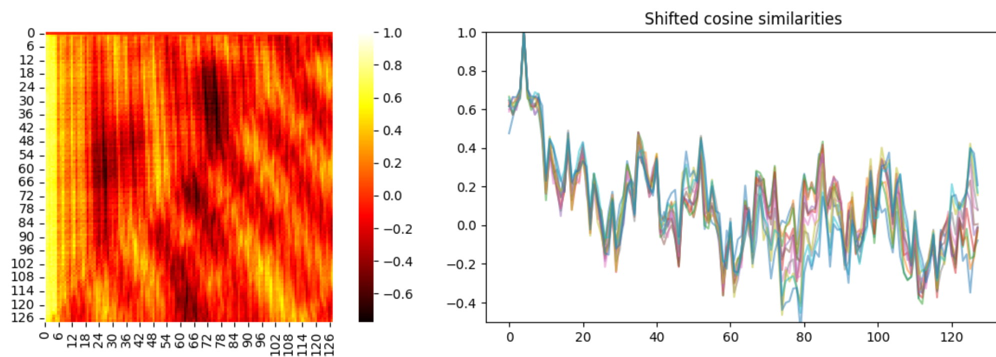


Figure 17: Cosine similarities of BERTs positional encodings, shifted to show relative properties.

figure) look random also here, supporting our previous observation. The linear structure is also to be somewhat expected – the positional encodings do encode the linear ordering of the sequence of input tokens.

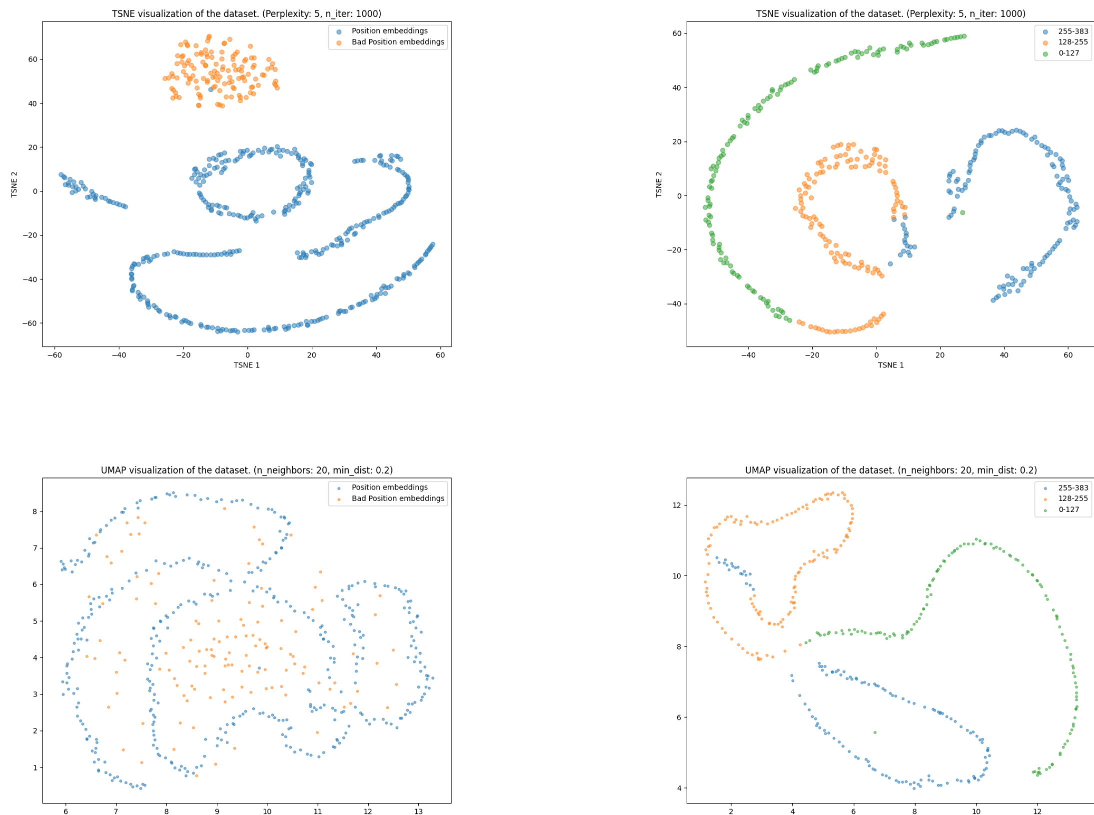
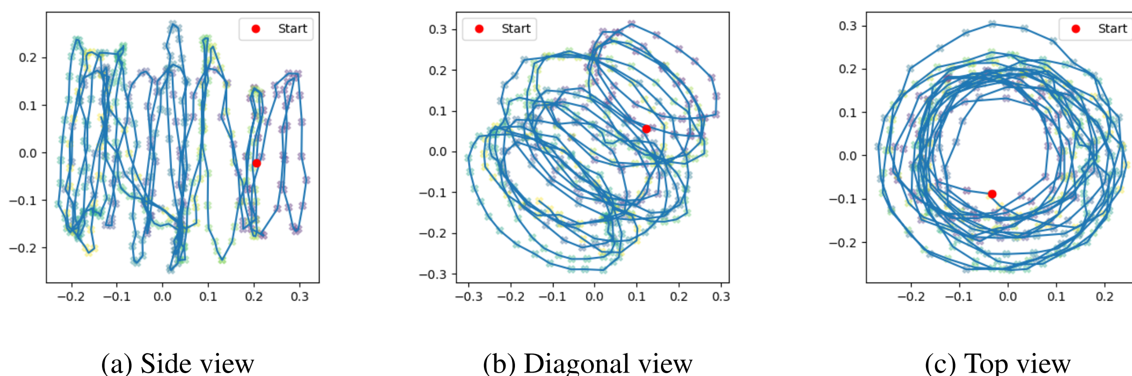


Figure 18: The UMAP and t-SNE projections of the BERT positional embeddings, with the "bad" positional encodings from 384 onwards highlighted.

More striking is the observation that if we take the PCA-projection of the positional embeddings to 3D, a helix-like path emerges; see Figure 19. There are two things we wish to emphasize here:

1. The PCA projection is an orthogonal projection, so it does not 'invent' new geometry unlike t-SNE or UMAP. So there really is some sort of helical structure in the full structure of the embeddings. Furthermore the t-SNE and UMAP projections shown in Figure 18 support the idea that the topological structure of the positional embeddings is strongly 1-dimensional.

2. This is a projection to 3 dimensions, meaning that we are ignoring 509 dimensions[6] of geometry and only looking looking at those three that generate most variance. Recall from the PCA dimensional analysis in Figure 11 that the most dominant 3 dimensions only capture around 30% of the variance visible in the data. So even though the embeddings seem to have 1-dimensional topology, and contain a helical factor, it does not mean that this is all that there is to the geometry of the positional embeddings.



(a) Side view (b) Diagonal view (c) Top view

Figure 19: The helical structure of the first 384 positional embeddings of the BERT model, excluding position 0. PCA projection to 3 components. We do one 'round' once in about every 22 tokens.

We note that the cyclical nature visible in Figure 19 shows that we go around once in about every 22 tokens. This frequency of 22 is not mirrored in Figure 17, which also seems to also suggest that the movement that the positional embeddings do in the rest of the 509 dimensions is non-trivial.

4.2.4 The (two) sentence type encoding vectors

As mentioned, we have only two of these and their norms are 0.70 and 0.72, their cosine similarity is -0.065, their dot product is -0.033 and their Euclidean distance is 1.04.

As discussed in Section 3.3.3, part of the training task is to predict form the final [CLS] token embedding whether two sentences follow each other. These two sentences are distinguished

6. The positional embeddings live in $\mathbb{R}^{768}$, but since we have only 512 of them, they are contained in an (at most) 512-dimensional linear subspace of $\mathbb{R}^{768}$.

both by a separating special [SEP] token and by adding the respective sentence type encoding vectors to the tokens of each sentence. For many down-stream tasks like text classification from the [CLS] head or named entity recognition from other tokens, only the first sentence type encoding vector is used and is thus pretty much a constant vector added in the beginning.

4.2.5 The interrelations of the context-free embedding types

The three types of embeddings – input embeddings, positional embeddings and sentence type embeddings – represent conceptually very different things. In this section our aim is to study if we are able to observe this dissimilarity beyond the statistical differences we saw in Figure 10.

Looking at the cosine similarities between the positional embedding vectors and the context-free embedding vectors, we note that with around 30k context free embeddings and 512 positional encoding vectors, there are around 14M pairwise cosine similarities that can be calculated. From these 14M there are exactly four token-position pairs that have cosine similarity outside of the range $[-0.2, 0.2]$ and we've listed three of these in Table 4. Essentially we see that the special [CLS] token which always appears in the first position, i.e. the zeroeth index, has an exceedingly high cosine similarity with the corresponding position embedding. Indeed, these two vectors have norms of 1.44 and 1.48, their Euclidean distance is 0.42 and their l_∞-distance[7] is 0.06. So these two vectors are almost identical – recall from Table 2 in the previous sector that the two tokens with most similar context-free embeddings were "£1" and "£2" with a cosine similarity of 0.93!

Now, for the differences of the two sentence type encodings against the position embeddings and the context-free embeddings we've calculated the pairwise cosine similarities. Table 5 lists some of the main statistics.

What we note here is that even the extremal values are barely higher than the $[-0.2, 0.2]$ range with the supermajority being pretty much orthogonal. Thus we can, for the second time, safely conclude that we have answered our first research question in the positive.

7. I.e. the maximum difference found in any coordinate.

Token	Position	Cosine similarity
`[CLS]`	0	0.96
`[SEP]`	0	0.48
`[MASK]`	0	0.48
##`U+0964`	0	0.204

Table 4: Table showing the four token-position pairs whose cosine similarity is outside of the range $[-0.2, 0.2]$. The final token is a punctuation mark in sanscrit called "*devanagari danda*" and it is nontrivial to display it LaTeXformat.

Pair	Max	Min	Avg	5% Percentile	95% Percentile
Sent. type vs Position	0.2349	-0.1514	0.0123	-0.0738	0.1412
Sent. type vs Context-free	0.2183	-0.1899	-0.0400	-0.1081	0.0238

Table 5: Some statistics on the cosine similarities between the sentence type encodings and other pre-encoder embedding vectors.

4.3 Evolution of embeddings within the encoder

We now move on to study context-*aware* embeddings by advancing through the attention layers. Studying all possible situations by brute force turns completely unfeasible at this point – instead we turn to inferring properties of the whole by studying samples. In particular we'll be interested to observe any geometrical properties in the embedding vectors generated from text.

A priori there is no law of nature that the different embedding spaces have any geometric relation to each other, nor that they represent the same conceptual space. Though we do note that the various skip layers in the transformer blocks probably create quite a strong connection between two consecutive layers. In this Section we'll put this question to the test and see how the embeddings of some example sentences evolve as they traverse through the layers. For further resources on this topic we do not cite direct papers, but note that the idea of "embedding evolutions" is related to (or better known as) *the residual stream* of a

transformer.

Using the Python `transformers` library we can easily access the interim embedding layers of BERT. With the 12 transformer blocks we get of course 12 output embedding collections, but we also have the input to the first transformer block at index 0, totaling at 13 embedding collections. Recall that the inputs to the first transformer block are the context-free embeddings that has been enhanced with the positional encodings and the sentence encoding.

We've taken two texts to study these token evolutions:

"The mouse was gone. The cat sat useless."

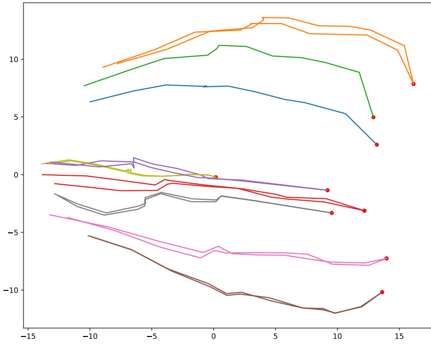
and

"The mouse was gone. The computer sat useless."

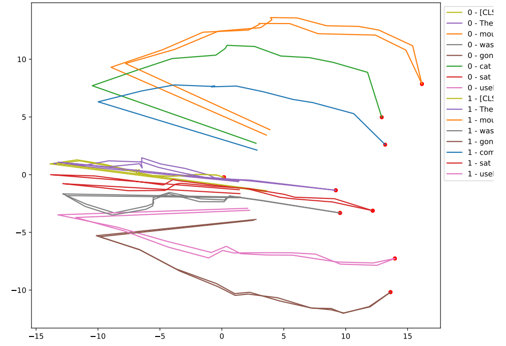
We're using standard BERT, and in these sentences each word is a single token. The idea with these texts is that the token "mouse" in the first sentence will have a different meaning depending on the second sentence – it might be a mammal or a computer peripheral. We've plotted the embeddings of all the tokens as they 'evolve' through the transformer blocks in Figure 20. We noticed that the change in the embeddings at the last transformer block was much larger than in the other layers, and we've plotted the path both with the last jump included and not.

So it seems that the embeddings are traversing together. And excepting the last embedding layer, which is most sensitive to retrainings due to lack of gradient decay[8], in the PCA projection they seem to be going to the same direction. This can be partially illusory, as for the direction to look similar, we just need to have a few coordinates (out of 768) where the changes happen strongly in the same direction, and the PCA will "favor" those directions in the projection direction as it is fitting to our data where we are looking at these evolution paths. In Figure 21 we've plotted the paths of only a few tokens to show that the movements are not quite as uniform as Figure 20 might suggest, especially since here too we should expect the "PCA illusion" to be at play.

8. Discussion on this can be found e.g. in Rogers, Kovaleva, and Rumshisky 2020.

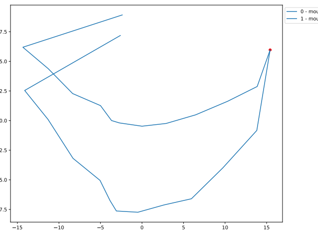


(a) Last embedding not included.

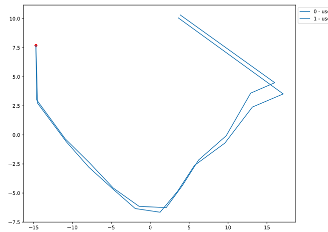


(b) Last embedding included.

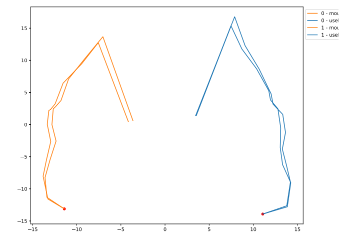
Figure 20: PCA projection of the evolution of token-level embeddings of two disjoint sentences; "The mouse was gone. The cat sat useless." and "The mouse was gone. The computer sat useless.". Red dot marks the starting embedding.



(a) The 'mouse' token.



(b) The 'useless' token.



(c) Both the 'mouse' and 'useless' tokens.

Figure 21: PCA projection of the evolution of only one or two tokens.

Another approach to get a better picture on how the evolution paths look is to check for the consecutive distances, cosine similarities and norms, see Figure 22. What we see here is that these embeddings tend to take "steps" of roughly the same length, change the "direction" very little[9] one at almost each step and move somewhat linearly away from the origin. Note that the radial movement can be the reason for only a small amount of each step, as the Euclidean distance 'hop' seems to be around 6, but the norms increase by only about 2 per layer.

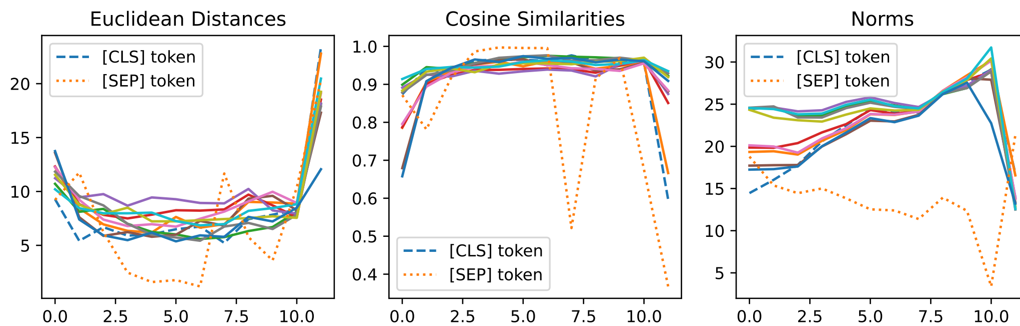


Figure 22: The Euclidean distances and cosine similarities of consecutive embeddings of the tokens in "The mouse was gone. The cat sat useless.", and the norms of the embeddings at each step.

Our second research question asked if we can observe the changes in the embedding vectors as they 'evolve' in the encoder stack. Thus with this section we have answered this research question in the positive.

4.4 Semantic content and structure of the embeddings after the encoder

We start by emphasizing that in this section we will be working solely with the final output of embeddings from the encoder. In particular, these are the embedding vectors that most down-stream tasks use as their input.

9. This is a bit surprising; any two random directions in a high dimensional Euclidean space are almost orthogonal with an overwhelmingly high probability. We conjecture that this effect arises from the skip connections that "stabilize" the evolution of the embedding vectors.

The claim we want to study in this section relates to the question about how well we should expect the semantic content of embeddings to be related to the structure of $\mathbb{R}^n$. As discussed in the beginning of this chapter, it is customary to use cosine similarity or Euclidean distance to measure embeddings, see again e.g. Rogers, Kovaleva, and Rumshisky 2020, but it is quite a strong implicit claim that this works, especially in relation to semantic content. Indeed, it is not at all obvious if the embeddings should have any relation to the structure we instictually affiliate with Euclidean spaces, or if they are just collections of numbers.

To emphasize the issue, consider another case where the vectors we study do not come from a language model but from some database. One of the coordinates is the last three numbers of a license plate, another a postal code, the third a serial number of a mechanical part and another mass in kilograms. Such vectors do consist of numbers, but there is no reason to assume that e.g. an *angle* would be a suitable measure for the similarity of two vectors. If the serial number and licence plate numbers are even pseudorandomly assigned, even Euclidean distance is probably a useless concept[10]. In this kind of setting the problem is related to the fact that some numbers like weight represent continuous quantities while other like a serial number are just indexes. They might not have even a reasonable ordering related to them.

For the embedding vectors we do sort of expect the structure of $\mathbb{R}^n$ to be somehow involved, though we should remember that when being trained, the model is just trying to minimize the loss function over training data. The loss function doesn't (in any case we're aware of) contain any component that would reward or penalize the model based on how well the semantic structure of the embedding space correlates with linear algebra.[11] So the fact that we do find some relation is a noteworthy and a non-trivial observation.

On the other extreme, we do not expect that the embedding vectors would place any specific weight to values like 3.14 or 42, as these are at least somewhat cultural concepts. So where exactly will we land here? How much of the familiar structure of the Euclidean space is present? Before diving into the literature, we'll do a few preliminary tests. We first look at the *topology* of the space with the question if similar concepts will have similar embeddings

10. And the problem can't be solved by "just normalizing the data".

11. Though, this could be argued against by the fact that cross entropy loss is at least continuous, so the continuity might get rewarded by some meta-mechanism.

in the sense of Euclidean distance. This is our weak approach to something like continuity. We then move to more algebraic questions by studying the difference vectors of embeddings and see if those have anything reasonable to say for use.[12]

4.4.1 Topology of embeddings

Here we want to see if sentences that have similar semantic content have also similar embedding vectors. To this end we have generated 4 sets of sentences that have roughly the same semantic content within the group but written in slightly different ways.

- 27 "Hello!" -sentences where the highlighted three words vary through synonyms.

  $$\left\{\begin{matrix}\text{Hi}\\ \text{Howdy}\\ \text{Hello}\end{matrix}\right\} \text{ there! I am } \left\{\begin{matrix}\text{glad}\\ \text{happy}\\ \text{felicitous}\end{matrix}\right\} \text{ to meet such a nice } \left\{\begin{matrix}\text{person}\\ \text{individual}\\ \text{somebody}\end{matrix}\right\} \text{ here.}$$

- 27 "New car" -sentences where the highlighted three words vary through synonyms.

  $$\text{We're going to buy the } \left\{\begin{matrix}\text{car}\\ \text{auto}\\ \text{automobile}\end{matrix}\right\} \text{ today. We}$$

  $$\text{want something that's } \left\{\begin{matrix}\text{fast}\\ \text{quick}\\ \text{speedy}\end{matrix}\right\} \text{ and } \left\{\begin{matrix}\text{cheap}\\ \text{inexpensive}\\ \text{affordable}\end{matrix}\right\}.$$

- 27 "Happy boss" -sentences where the highlighted three words vary through synonyms.

  $$\text{The } \left\{\begin{matrix}\text{boss}\\ \text{manager}\\ \text{supervisor}\end{matrix}\right\} \text{ was } \left\{\begin{matrix}\text{glad}\\ \text{happy}\\ \text{content}\end{matrix}\right\} \text{ with the } \left\{\begin{matrix}\text{employee}\\ \text{worker}\\ \text{staff}\end{matrix}\right\} \text{ for the new sales record.}$$

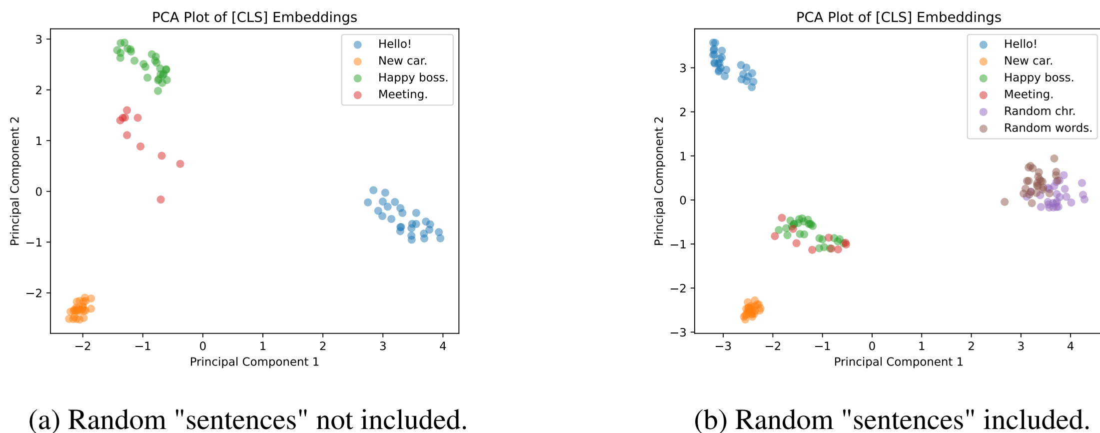
- 10 "Meeting" sentences, generated with the help of chatGPT. We've listed three examples here, the full list can be found from https://github.com/ramiluisto/NLP_toybox/blob/main/data/sentence_pairs_extended.json.

  - "The meeting is set for Monday at 10 AM, and breakfast will be provided by someone.",
  - "Someone will provide breakfast at the meeting, which is scheduled for 10 AM on Monday.",

12. Our separation of the concepts of $\mathbb{R}^n$ to topological vs algebraic is sort illusory as the standard topology of $\mathbb{R}^n$ is completely determined by the linear algebraic structure. And the so called exotic topological structures of $\mathbb{R}^n$ really are exotic and finding one of those here would be nothing short of a miracle. So this is a conceptual study and separation, not a mathematical one.

– "We've arranged the meeting for 10 AM on Monday, with breakfast being brought by a participant.",

There are of course slight differences in the content of these sentences within each group, since e.g. "Howdy" vs. "Hello" carry different cultural connotations. Regardless, the sentences should be more similar *within* each group than *between* different groups. We also added two classes of random sentences; one class that sampled random characters and another that sampled random English words. We've taken the [CLS]-token embedding vectors of these sentences and created a 2D PCA projection of the different classes, shown in Figure 23.



(a) Random "sentences" not included.

(b) Random "sentences" included.

Figure 23: The PCA projection of the [CLS] embeddings of various synonymous sentences.

As we've hoped, we see clearly different components for these different sentence types, and that within a group the sentences are similar. This is evidence towards the "semantic continuity" that we would like to see here, and answers our research question 3a which was asking for just this type of connection between the semantic content of text and the topology of the embedding vectors arising from it. Furthermore note that the two types of random sentences are quite similar, but there seems to be a level of differentiation. So the model can detect nonsense from non-nonsense, and even between types of nonsense.

4.4.2 Algebra of embeddings

We now turn to look at the "algebraic" structure of embeddings. In particular, an important part of the embedding space is that it is not just a topological space, but that the concepts that it encodes have some relation with the linear algebra structure. In particular, the vector differences in embedding vectors convey some meaning. To illustrate what we mean, we've created sentence pairs in a few different categories, about 10 from each category. We've listed an example pair from each category in Table 6, the full set of pairs is available at https://github.com/ramiluisto/NLP_toybox/blob/main/data/sentence_pairs_extended.json.

Difference	**Sentences**
Bright-Dim	"The bright light illuminated the room."
	"The dim light illuminated the room."
Doctor-Nurse	"The doctor was walking down the street."
	"The nurse was walking down the street."
Female-Male	"She went home."
	"He went home."
Finnish-Czech	"The Finnish scientist discovered a new element."
	"The Czech scientist discovered a new element."
Funny-Sad	"The movie was so funny, we couldn't believe it."
	"The movie was so sad, we couldn't believe it."
Laugh-Cry	"The baby was laughing."
	"The baby was crying."

Table 6: Sentence pairs for difference analysis.

For each sentence pair (S_1, S_2) we've then created the embedding vectors (E_1, E_2) of the [CLS] token of these sentences with the standard BERT, and taken the vector difference $E_2 - E_1$ of the embeddings. These differences are PCA-visualized in Figure 24 with the average vector of each set of 10 differences marked with an arrow.

What we note here is that many of the differences are clearly correlated within a type and

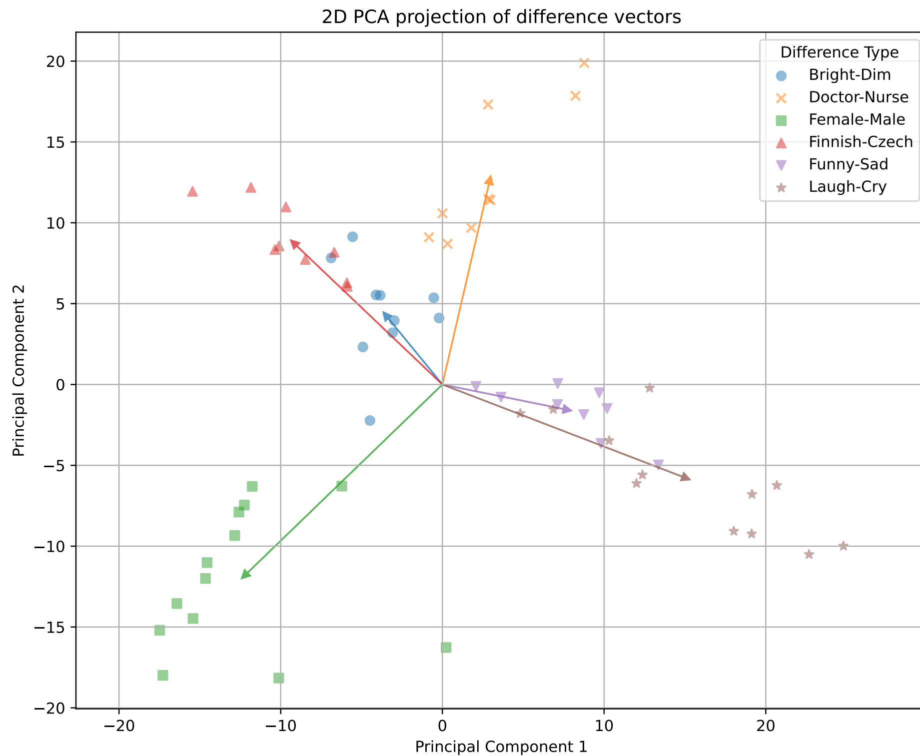


Figure 24: PCA projection of the differences in the embedding vectors. The points represent individual differences and the arrows the average difference vector of each type.

separated from each other! So the algebraic action of taking a difference does bear some connection to the semantic content. In particular, we could now consider that the green vector, which is the average value of the green differences, is a sort of internal representation of the idea of "more sad than happy". The idea now is that with this interpretation we can try to see if these concepts are correlated in the internal model of the, well, model.

The way we check the correlation is by looking at the cosine similarity of the average vectors within each difference class – we have listed these in Table 7. We've highlighted some of the extreme values: the smallest similarity is between the concepts of "Doctor vs Nurse" and "Bright vs Dim" - the model seems to find very little similarity between light level and these two professions. On the other hand, the cosine similarity between "Funny vs Sad" and "Laugh vs Cry" is quite high – this sounds reasonable as these differences do seem to be strongly related. We also note some higher correlation between brightness and mood, and amusingly in Finnish people being more likely to cry then the Czech.

A very crucial point here is, however, the similarity between "Female vs Male" and "Doctor vs Nurse". With cosine similarity of -0.644 the model seems to have an internal model of the world where these two differences have a strong anticorrelation. In plain English, BERT seems to think that being a doctor correlates strongly with being a man and being a nurse correlates strongly with being a woman. This is not an original observation, but a classical example and a good starting point in the analysis of bias in language models. The issue here is that this quite misogynistic bias is something that has been present in the training material of the language model, and thus BERT has learned to model itself accordingly. So we have better optimized at minimizing the loss function at training time by fitting to this bias, but when building AIs that have any sort of agency in the real world, this will be a bad thing.

Quantity 1	Quantity 2	Cosine similarity
Bright-Dim	Doctor-Nurse	**0.002**
Bright-Dim	Female-Male	0.056
Bright-Dim	Finnish-Czech	0.097
Bright-Dim	Funny-Sad	0.133
Bright-Dim	Laugh-Cry	0.145
Doctor-Nurse	Female-Male	**-0.644**
Doctor-Nurse	Finnish-Czech	-0.046
Doctor-Nurse	Funny-Sad	0.018
Doctor-Nurse	Laugh-Cry	0.150
Female-Male	Finnish-Czech	0.018
Female-Male	Funny-Sad	-0.073
Female-Male	Laugh-Cry	-0.064
Finnish-Czech	Funny-Sad	-0.022
Finnish-Czech	Laugh-Cry	-0.173
Funny-Sad	Laugh-Cry	**0.465**

Table 7: Table showing the cosine similarities of the average difference vectors. We've highlighted some of the more extreme values.

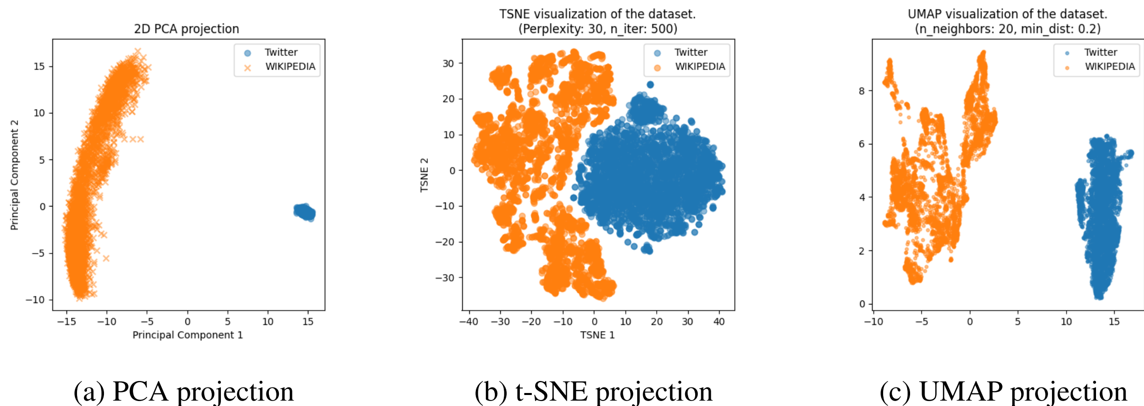
These results answer our research question 3b, which was asking if there is an observable

connection between the linear algebraic structure of the embedding vectors and the semantic content of the text from which they arise.

4.4.3 Geometry of embeddings

For the geometry of embeddings we base our approach to the idea that a single point or a pair of points has no geometry - geometry will be a property of a large collection of points.[13]

Trying to understand the geometry of BERT is not a novel idea, see e.g. Reif et al. 2019. A full geometric analysis is beyond the scope of this thesis, but we do want to observe the nontriviality of the task. To this end we've generated a random sample of 3860 Wikipedia texts via the Wikipedia API[14] and taken a collection of 3257 Twitter texts from a Kaggle dataset[15]. In Figure 25 we show the basic projections of the [CLS] token embeddings of these texts.



(a) PCA projection (b) t-SNE projection (c) UMAP projection

Figure 25: Three projections of Wikipedia and Twitter data.

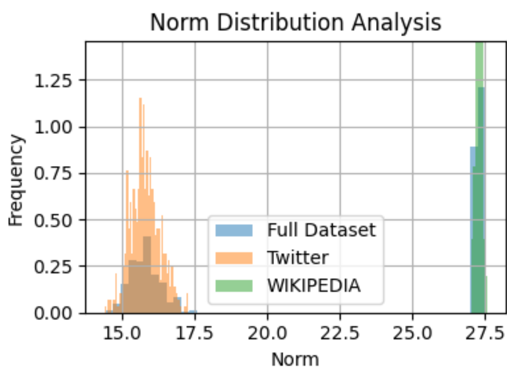
What is obvious from the very surface is, that the different data sources seem to be quite dissimilar in the embedding space, and that they contain nontrivial geometrical structures – especially for the Wikipedia embeddings.[16] This difference in structures is also supported by the distribution of our various metrics; see Figure 26.

13. Three points already form a triangle, and we can argue that geometry starts. For the current purposes, however, three is still a small number.

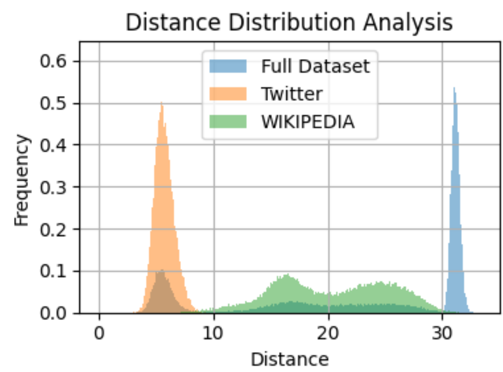
14. See https://github.com/ramiluisto/NLP_toybox/blob/main/src/wikipedia_datafetch.py.

15. https://huggingface.co/datasets/tweet_eval/tree/main/emotion

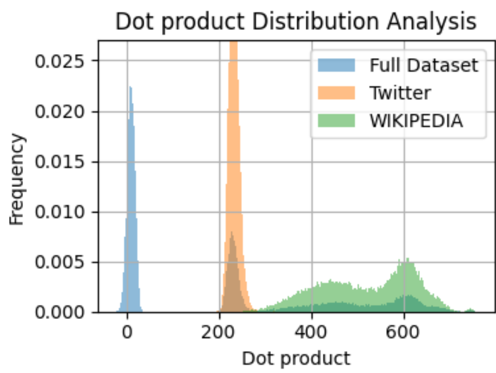
16. Maybe partially due to the longer length and thus higher possible variations?



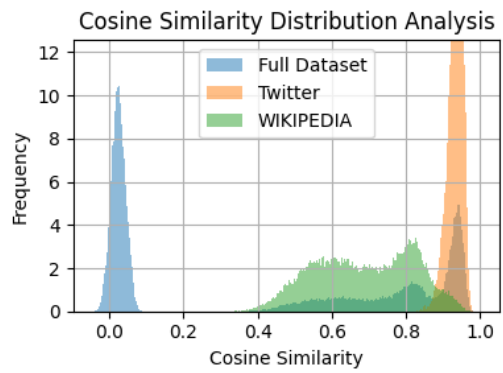
(a) Euclidean norm distribution



(b) Euclidean distance distribution



(c) Dot product distribution



(d) Cosine similarity distribution

Figure 26: Distribution visualizations of a sampling of random Wikipedia and Twitter texts.

One very strong phenomenon we observe is that within each class of texts the cosine similarities are all positive, and quite large even. As previously mentioned, this contrasts the fact that randomly sampled vectors in high-dimensional Euclidean spaces tend to be almost orthogonal. The observation of this discrepancy has been observed also in Ethayarajh 2019 where they note that randomly sampled word embeddings tend to *not* average around zero for BERT or GPT-2, though we are here only looking at the [CLS] token embeddings. Our conjecture here is that idea of two concepts being linguistically "independent" (i.e. cosine similarity near zero) or "opposites" (i.e. cosine similarity near -1) is very context dependent. For example, the sentences "The room was dark." and "The room was bright." do contain in some sense opposite claims, but they also both discuss the illumination status of rooms in a short grammatically correct English sentences. Even the singular words/tokens "dark" and "bright" are opposites only in a particular contexts; they are both adjectives in English and used in similar sentences and we simply ignore their similarities when thinking them as opposites.

With this we can also claim a positive answer to our research question 3c, asking for observable geometrical properties in the data clouds arising from the embedding vectors of text collections.

4.5 Some practical applications of the embeddings

This chapter has been filled with various probings to the behaviour and structure of the embedding spaces. The motivation has been to observe some inherent properties of embedding vectors and the spaces they form, but now we wish to close up this chapter by listing some of the practical uses of embeddings.

4.5.1 Vector databases for text search

Creating text embeddings for a large text corpus can be very useful in creating semantic search functionality to an otherwise unstructured collection of data. The idea here is that we split our collections to suitable lengths, e.g. to about 512 tokens, and create a database where we have the text, possibly some tags about their source and location, and the embeddings of

these texts. (In practice we might split the text with some overlap to avoid any critical words or sentences losing their context in a split boundary.) Then when you wish to search for e.g. "all texts that discuss bears walking on their hind legs", you simply create the embedding for this query and find the top N database entries sorted by the cosine similarities between the query text embedding and the embeddings in the database. As we've seen in this section, such an approach has promise since the embedding vectors seem to capture semantic ideas.

At this point of the thesis, the natural question of course is: "which embedding should you use?" For a given text of N tokens we get $13 \cdot (N+2)$ embeddings from the BERT model. A classical quick choice would be the [CLS] token embedding from the last layer, but we note that already in (Devlin et al. 2019) the authors recommend fine-tuning before using [CLS] to tasks outside sentence pair prediction. Other solutions might replace (or augment) the [CLS] token embedding with an average or coordinate-wise max of the other embeddings from the final layer. You might even want to include embeddings from lower levels in some cases. Then there is the question if you should use the basic BERT model, or fine-tune it on your corpus data in some fashion.

None of these questions have a universal correct answer, but understanding even the basics of embeddings is crucial to making an informed decision.

4.5.2 Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) is a technique for improving the context awareness of text-generating AIs like chatGPT. The idea here is that we have in the backend of the system a vector database of relevant data. When a user writes a query, before the query is passed to the chat-AI as a prompt, we perform a vector database search to find relevant context data, and pass that on with the prompt.

This can greatly improve the chat-AI performance, and also the reliability. You have less to fear about hallucination when the chat-AI can give you a link to e.g. internal documentation. A classical application example would be a chatbot that aids you to study a company's internal documentation. See e.g. the RoboCorp ReMark at https://chat.robocorp.com/ for an example.

4.5.3 Compute-effective NLP

Training 110M parameter language models takes compute resources, even if you only do fine-tuning. Running the models is orders of magnitudes faster and less resource-heavy. For this reason we can often use models like BERT to simply do a static mapping of texts to embedding vectors that we can use as features, and then use other less resource-intensive ML tools on those features.

Here it is even more important to grasp the differences of the different embedding types so that you can choose the right ones for the task.

4.5.4 Interpretability

In many practical applications of ML, it is not enough to report the accuracy or f1 score of the system and call it a day. Especially in systems that handle sensitive data or which are involved in delicate operations like medical care, you usually have to be able to *explain* at least partially what the system is doing, how and why. For large neural networks like BERT it is, at least currently, impossible to completely understand what is going on, but we can do better than calling it just a black box. The various methods used here can be included in a study to see what a full system is really made of, and get at least partial answers to questions about why something is happening.

4.5.5 Bias detection

As we've noted, the BERT model has biases. To be able to counteract these we must first find them, and in both of these tasks we must turn our attention to the internals of the model. Embeddings are not the full story here for BERT and other transformer models as the attention mechanism is very crucial here as e.g. the creator of bias, but it is in the embedding vectors that we often find the biases from.

5 Conclusion(s)

We finish the thesis, naturally, by writing up our conclusions.

We stated three research questions in the introduction, and we've found a suitable answer to each:

1. The differences between positional encodings, context-free input embedding vectors and sentence type embeddings are very clear. Their differences are easily visible both in their intergroup statistical distributions and in their pairwise cosine similarities between the groups. The exception being some special tokens, with the [CLS] token and the index 0 positional encoding being so close to essentially be the same vector learned via two different paths. This is natural as these two vectors are always summed together and thus we cannot, under normal operations, observe them separately.
2. The evolution of the embedding vectors through the encoder stack was observed, though we did not probe into what exactly was learned in the progress. However, we could see how the meaning of homonyms (computer mouse vs. an animal mouse) drifted apart in the evolution.
3. In studying synthetic and real-world data we were able to observe the connections of the semantic content of text to the topology, algebra and geometry of $\mathbb{R}^{768}$. This also provides a very strong indicator on the usefulness of the BERT embeddings in various downstream tasks.

5.1 Some post facto reflections and a critical look at the research methodology

As discussed in Section 4.1, our research "methodology" was based on exploration via hypotheses and tests driven by curiosity and personal preferences. This naturally made the approach personally very fulfilling (especially since we at times succeeded) and fun, but now it is a natural point to reflect a bit – also on the shortcomings of the approach.

Since this thesis is a work of limited scope, we have naturally selected only the most inter-

esting observations we have made to be included in the final version. This is both how most of scientific publication works, and also very close to what we would call *p-hacking*.[1] By this we mean that there is a balance in how to report one's research results. On the one hand, no-one wants to overwhelm the reader (or reviewer) by describing all the intellectual dead ends that were traversed on the way to the result, and so it is best to focus on the interesting results that ended up working. On the other hand, it is counter productive to science as a whole to try to pretend that you set out to find a particular discovery when in fact you tried a thousand things and only showed the one that happened to stick to the wall. Especially since if you leave undocumented the dead ends you found, you only leave them unmapped for the next unwary researcher.

In this work we set out to understand how the geometry and structure of embeddings look like, and we've reported some of the more easily digestable and demonstrable findings. But at the source folder of this thesis, our `./img` folder has over 300 images, many of them being various plots generated with Python data analysis tools. Most of these pictures are not useless or wrong, but they tend to be either less striking or not so informative on the various exciting points of the geometry that we wanted to discuss. This does not mean that we are cheating, but we still need to keep in mind that the plots and data we see are the *crème de la crème* of the many things we could be showing.

Besides this "survivorship bias" of plots, there are also several research avenues which we approached but either did not bear fruit, or bore fruit but in such a laborious way that we could not think of a way to include the results in this thesis without severly overcrowding it. For example, we did a few deep dives to see how biases evolve through the encoder block, and a more detailed study on some of the subparts that seemed to be visible in the various 2D projections of our Wikipedia and Twitter datasets. Neither of these approaches did not yield anything exciting, at least on our first pass. We also did a somewhat deep mathematical analysis on the geometry of almost orthogonal vectors in $\mathbb{R}^{768}$ and, while interesting, it did not really support any of the main observations we have focused here. Thus all of these avenues have been left out, though we hope to report on them in some other future work.

It would have been nice if we had been able to draw a better 'map' of which avenues of study

1. See e.g. https://en.wikipedia.org/wiki/Data_dredging.

we explored and how they panned out. But in the end despite some outer appearances this is not a personal learning diary but a master's thesis and it is hard to find a scientific way to report that an approach was tried and abandoned due to reasons like "did not feel promising", "was not exciting enough" or "I kind of forgot about this".

With this disparagement of our methodology out of the way, we still find the approach to have been a success. With our exploratory approach we had motivation to familiarize ourselves with wide swathes of modern research and we ran across enough interesting phenomena that we could afford to be picky about what we include. Explaining our discoveries and trying to find matches in the literature that used very different terminology (and usually a more mature approach) helped us grasp the concept on a much deeper level than if we had at the start read existing literature and written down our own summaries.

5.2 Some possible further research questions

There are several things that we feel would merit further study. We list some of them here.

1. We noted that the cosine similarities between embeddings, context-free or context-aware, were not averaging around zero. (See again also Ethayarajh 2019.) We discussed some hypothesis on how the concept of "opposites" is very context dependent. Are there methods where we could tease out the idea of oppositeness in embedding vectors? E.g. by substracting from the vectors all "adjectiveness" and the like? Or by just doing orthogonal projections to each others?
2. In continuation of the previous question, could we train a GAN to find sentence pairs that BERT would consider truly opposites? Would we be able to find any that are even somewhat grammatical? This question bears resemblance to techniques used to extract information about what kinds of features maximize the activation of various computer vision model subcomponents.
3. What really is the 1-dimensional structure of the positional encodings? Is it a spiral or something more fractal-like? Measures of Hausdorff-dimension or persistent homology might be interesting. Especially if we try to find common themes across a few different model types.

4. The context-free embedding vectors had "pac-man" shape and some other geometric properties in the PCA projection. What kinds of components could we distinguish with deeper study?
5. The process of compressing language models by dropping the bit size of floating points used in the weights is very popular. Does this create any noticable non-trivial effects on the geometry of the embeddings?

5.3 Epilogue

As first-year student I once happened upon a poster session of some physics students. The one poster that caught my eye was a study on how a boiled egg will spin differently from unboiled ones. They had done the science and concluded with a sentence that stuck to my mind for years:

> We were not able to replicate the result, but we were able to observe the phenomenon.

This could be a slogan for this thesis as well. There are hundreds of high-quality studies on the behaviour of BERT alone, see e.g. Rogers, Kovaleva, and Rumshisky 2020 and the references within, and most of the tests we tried we have then found in the literature with more extensive analyses and more solid statistical foundations. But even if we haven't found anything new here we've been able to observe many interesting phenomenon. And that has been very much worth the effort.

Bibliography

Alexander, Scott. 2024. "Janus Simulators." Astral Codex Ten. Accessed: 2024-04-24. Accessed April 24, 2024. https://www.astralcodexten.com/p/janus-simulators.

Barbieri, Francesco, Jose Camacho-Collados, Leonardo Neves, and Luis Espinosa-Anke. 2020. "Tweeteval: Unified benchmark and comparative evaluation for tweet classification." *arXiv preprint arXiv:2010.12421.*

Björklund, Anton, Lauri Seppäläinen, and Kai Puolamäki. 2024. "SLIPMAP: Fast and Robust Manifold Visualisation for Explainable AI." In *Advances in Intelligent Data Analysis XXII,* edited by Ioanna Miliou, Nico Piatkowski, and Panagiotis Papapetrou, 223–235. Cham: Springer Nature Switzerland. ISBN: 978-3-031-58553-1.

Britz, Denny, Anna Goldie, Minh-Thang Luong, and Quoc Le. 2017. "Massive exploration of neural machine translation architectures." *arXiv preprint arXiv:1703.03906.*

Clark, Kevin, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. 2019. "What Does BERT Look at? An Analysis of BERT's Attention." In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP,* 276–286. Florence, Italy: Association for Computational Linguistics. https://doi.org/10.18653/v1/W19-4828. https://www.aclweb.org/anthology/W19-4828.

Conneau, Alexis, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2019. "Unsupervised cross-lingual representation learning at scale." *arXiv preprint arXiv:1911.02116.*

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers),* 4171–4186. https://aclweb.org/anthology/papers/N/N19/N19-1423/.

Ethayarajh, Kawin. 2019. “How Contextual Are Contextualized Word Representations? Comparing the Geometry of BERT, ELMo, and GPT-2 Embeddings.” In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP),* 55–65. Hong Kong, China: Association for Computational Linguistics. https://www.aclweb.org/anthology/D19-1006.

Goldberg, Yoav. 2019. “Assessing BERT’s Syntactic Abilities.” *arXiv preprint arXiv:1901.05287,* http://arxiv.org/abs/1901.05287.

Gordon, Mitchell A, Kevin Duh, and Nicholas Andrews. 2020. “Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning.” *arXiv preprint arXiv:2002.08307,* https://arxiv.org/abs/2002.08307.

Guo, Fu-Ming, Sijia Liu, Finlay S. Mungall, Xue Lin, and Yanzhi Wang. 2019. “Reweighted Proximal Pruning for Large-Scale Language Representation.” *arXiv:1909.12486 [cs, stat]* (December). Accessed August 19, 2020. arXiv: 1909.12486 `[cs, stat]`. http://arxiv.org/abs/1909.12486.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. “Long short-term memory.” *Neural computation* 9 (8): 1735–1780.

Howard, Jeremy, and Sebastian Ruder. 2018. “Universal Language Model Fine-Tuning for Text Classification.” In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* 328–339. Melbourne, Australia: Association for Computational Linguistics, July. Accessed January 28, 2020. https://doi.org/10.18653/v1/P18-1031. https://www.aclweb.org/anthology/P18-1031.

Htut, Phu Mon, Jason Phang, Shikha Bordia, and Samuel R Bowman. 2019. “Do Attention Heads in BERT Track Syntactic Dependencies?” *arXiv preprint arXiv:1911.12246,* http://arxiv.org/abs/1911.12246.

Jolliffe, Ian T. 2002. *Principal component analysis for special types of data.* Springer.

Karpathy, Andrej. 2015. *The Unreasonable Effectiveness of Recurrent Neural Networks.* https://karpathy.github.io/2015/05/21/rnn-effectiveness/. Accessed: YYYY-MM-DD.

Kovaleva, Olga, Alexey Romanov, Anna Rogers, and Anna Rumshisky. 2019. "Revealing the Dark Secrets of BERT." In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP),* 4356–4365. Hong Kong, China: Association for Computational Linguistics. https://doi.org/10.18653/v1/D19-1445. https://www.aclweb.org/anthology/D19-1445.

Li, Zhuohan, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E Gonzalez. 2020. "Train Large, Then Compress: Rethinking Model Size for Efficient Training and Inference of Transformers." *arXiv preprint arXiv:2002.11794,* https://arxiv.org/abs/2002.11794.

Liu, Nelson F., Matt Gardner, Yonatan Belinkov, Matthew E. Peters, and Noah A. Smith. 2019. "Linguistic Knowledge and Transferability of Contextual Representations." In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers),* 1073–1094. Minneapolis, Minnesota: Association for Computational Linguistics. https://www.aclweb.org/anthology/N19-1112/.

Liu, Yinhan, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. "RoBERTa: A Robustly Optimized BERT Pretraining Approach." *arXiv:1907.11692 [cs]* (July). Accessed August 14, 2019. arXiv: 1907.11692 [cs]. http://arxiv.org/abs/1907.11692.

MacKay, David JC. 2003. *Information theory, inference and learning algorithms.* Cambridge university press.

Martin, Eric, and Chris Cundy. 2017. "Parallelizing linear recurrent neural nets over sequence length." *arXiv preprint arXiv:1709.04057.*

McCarley, JS. 2019. "Pruning a BERT-based Question Answering Model." *arXiv preprint arXiv:1910.06360,* https://arxiv.org/abs/1910.06360.

McInnes, Leland, John Healy, and James Melville. 2018. "Umap: Uniform manifold approximation and projection for dimension reduction." *arXiv preprint arXiv:1802.03426.*

Michel, Paul, Omer Levy, and Graham Neubig. 2019. "Are Sixteen Heads Really Better than One?" *Advances in Neural Information Processing Systems 32 (NIPS 2019),* http://papers.nips.cc/paper/9551-are-sixteen-heads-really-better-than-one.

Miller, George A. 1995. "WordNet: A Lexical Database for English." *Communications of the ACM* 38 (11): 39–41. https://doi.org/10.1145/219717.219748.

Olah, Chris, Nelson Elhage, Neel Nanda, Nicholas Schiefer, Anthony Bau, Boris Power, David R. C. MacLean, et al. 2022. *Toy Models of Superposition.* https://transformer-circuits.pub/2022/toy_model/index.html. Accessed: 2024-08-07.

Prince, Simon J.D. 2023. *Understanding Deep Learning.* The MIT Press. http://udlbook.com.

Radford, Alec, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. "Improving Language Understanding by Generative Pre-Training." https://www.cs.ubc.ca/~amuham01/LING530/papers/radford2018improving.pdf.

Raffel, Colin, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." *arXiv:1910.10683 [cs, stat],* http://arxiv.org/abs/1910.10683.

Reif, Emily, Ann Yuan, Martin Wattenberg, Fernanda B Viegas, Andy Coenen, Adam Pearce, and Been Kim. 2019. "Visualizing and Measuring the Geometry of BERT." In *Advances in Neural Information Processing Systems,* 8592–8600.

Rogers, Anna, Olga Kovaleva, and Anna Rumshisky. 2020. "A Primer in BERTology: What We Know About How BERT Works." *Transactions of the Association for Computational Linguistics,* https://doi.org/10.1162/tacl_a_00349.

Tenney, Ian, Patrick Xia, Berlin Chen, Alex Wang, Adam Poliak, R. Thomas McCoy, Najoung Kim, et al. 2019. "What Do You Learn from Context? Probing for Sentence Structure in Contextualized Word Representations." In *International Conference on Learning Representations.* Accessed January 21, 2020. https://openreview.net/forum?id=SJzSgnRcKX.

Tunstall, Lewis, Leandro Von Werra, and Thomas Wolf. 2022. *Natural language processing with transformers.* " O'Reilly Media, Inc."

Van der Maaten, Laurens, and Geoffrey Hinton. 2008. "Visualizing data using t-SNE." *Journal of machine learning research* 9 (11).

van Schijndel, Marten, Aaron Mueller, and Tal Linzen. 2019. "Quantity Doesn't Buy Quality Syntax with Neural Language Models." In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP),* 5831–5837. Hong Kong, China: Association for Computational Linguistics. https://doi.org/10.18653/v1/D19-1592. https://www.aclweb.org/anthology/D19-1592.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. "Attention is All you Need." In *Advances in neural information processing systems,* 5998–6008. http://papers.nips.cc/paper/7181-attention-is-all-you-need.

Vig, Jesse. 2019. "A Multiscale Visualization of Attention in the Transformer Model." In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations,* 37–42. Florence, Italy: Association for Computational Linguistics, July. https://doi.org/10.18653/v1/P19-3007. https://www.aclweb.org/anthology/P19-3007.

Virtanen, Antti, Jenna Kanerva, Rami Ilo, Jouni Luoma, Juhani Luotolahti, Tapio Salakoski, Filip Ginter, and Sampo Pyysalo. 2019. "Multilingual is not enough: BERT for Finnish." *arXiv: Computation and Language.*

Wang, Benyou, Lifeng Shang, Christina Lioma, Xin Jiang, Hao Yang, Qun Liu, and Jakob Grue Simonsen. 2020. "On position embeddings in BERT." In *International Conference on Learning Representations.*

Warstadt, Alex, Yu Cao, Ioana Grosu, Wei Peng, Hagen Blix, Yining Nie, Anna Alsop, Shikha Bordia, Haokun Liu, Alicia Parrish, et al. 2019. "Investigating BERT's Knowledge of Language: Five Analysis Methods with NPIs." In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP),* 2870–2880. https://www.aclweb.org/anthology/D19-1286/.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation." *arXiv preprint arXiv:1609.08144,* https://arxiv.org/abs/1609.08144.

Wu, Zhiyong, Yun Chen, Ben Kao, and Qun Liu. 2020. "Perturbed Masking: Parameter-Free Probing for Analyzing and Interpreting BERT." In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics,* 4166–4176. Online: Association for Computational Linguistics, July. Accessed August 14, 2020. https://doi.org/10.18653/v1/2020.acl-main.383. https://www.aclweb.org/anthology/2020.acl-main.383.