Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 6
Prof. Dr.-Ing. Hermann Ney

Selected Topics in Human Language Technology and Machine Learning im WS2023

Unsupervised Training of Language Representations

Ahmed Ali

Matrikelnummer 414360

10. Januar 2023

Supervisor: David Thulke

Inhaltsverzeichnis

1 Introduction

1.1 What are Language models

Language modeling is the task of assigning a probability distribution over sequences of words that matches the distribution of a language. Given a sentence, the model should be able to predict the missing word or maybe the next sentence (Next sentence prediction). Language Models (LMs) can be used in many machine learning tasks including but not limited to Classification, Question Answering, Sentiment analysis, and many more tasks. Also, LMs able to learn the context or the meaning of the sentence so it could precisely predict the right word. For example, imagine a sentence like "The fridge did not pass through the door because it was very small", as humans, I know that the door was small, however, for a machine, it could be somehow complex to learn whether the word ßmallïs describing the fridge or the door. That way, this method forces the model to learn how to use information from the entire sentence in deducing what words are missing.

What could I do with models that understand language? I heavily use language models daily in our life. For instance, the predictions that come after each word I write in our chat is coming from a language model, Moreover, when I search in google and then click the page of our interesting, google highlights the related paragraph that could answer our question (Question Answering).

1.2 Our Goals and Objectives

In this work, I will focus on the aspect of training language models. I will start with a quick overview of how language models became so powerful starting from a bag of words (BOW) until Chat-GPT. However, our main focus will be more directed to models like BERT, ELECTRA, and DeBERTa. I will focus on question-answering tasks and how each of the mentioned models performs on them. Then I will move to a language model for Non-English languages, I will focus on the Arabic language model discussed here [3].

2 Preliminaries

2.1 Language Models history

In this section, I will discuss the state-of-the-art techniques for language modeling over the past decade. Of course, this section does not reflect all the research and techniques covered in the past few years, however, it will give you a solid background about each technique and how each model improved the performance of its predecessors. I will discuss each technique in a nutshell so I focus more on the main task of this work.

Bag of Words (BoW) We convert the words to numbers by analyzing the presence of the word in a particular sentence. A number is denoted as an encoded value against the word. This is the number of times that word has been represented in the sentence. If only the presence is to be considered, then the game is denoted in form 1's and 0's. When the word is present in the sentence, it is denoted as 1 else 0. This is called a binary bag of words. It is worth mentioning that BoW would be useful in tasks like topic modeling because it looks only if the word is present or not, however, in tasks like sentiment analysis BoW will fail to predict the sentiment Ï do not recommendäs a negative one because BoW

does not have the ability to see the relationship between words.

ELMo (Embeddings from Language Models) is a Bi-directional LSTM language model the representations of ElMo are deep, in the sense that they are a function of all of the internal layers of the bidirectional [17] . According to the authors, ELMo can be easily added to existing models and it is very easy to fine-tune for tasks like question-answering, sentiment analysis, and textual entailment. Since ELMo uses LSTMs, it suffered from capturing dependencies for very long sentences.

GPT is then introduced by Open AI, GPT-3 is a decoder-only model. Unlike ELMo, GPT-3 is a unidirectional model, however GPT-3 also has some limitations. On text synthesis, GPT-3 samples still sometimes repeat themselves semantically at the document level, start to lose coherence over sufficiently long passages, contradict themselves, and occasionally contain non-sequitur sentences or paragraphs [7].

There are a lot of other language models like BERT (the main focus of this document) and T5 (Transfer Learning with a Unified Text-to-Text Transformer) [19] and recently GPTchat. I will discuss BERT in-depth and compare it to some other models in terms of accuracy for some downstream tasks.

2.2 Transformers

Transformers are state-of-the-art models that use the attention mechanism to boost the training speed as well as parallelize it. Transformers have a lot of application including but not limited to, Auto text memorization, Named Entity Recognition, machine translation and sentiment analysis. The transformer first proposed by the Google brain team in the Attention is all you need paper [22]. The implementation of the transformers is available as a public resource for anyone to invest and see how they optimized the parallelization and improved the training time. In this section I will walk through the transformer architecture and I will see the math behind it.

2.2.1 Challenges

Previously in RNNs, LSTMs, and GRUs when I wanted to translate a sentence let's say A in English into sentence B in German, I had to go through every word in our inputs one by one. The translation is done in a sequential way. For that reason, there is not much room for parallel computations here. The bigger sequences I have, the more time it will take to process that sequence, with large sequences, the information tends to get lost within the network, I will start seeing some problems like vanishing gradient arise related to the length of our input sequence. LSTMs, and GRUs helped somehow with the vanishing gradient problem but still, those architectures do not perform well on very long sequences. Transformers solved both those problems (parallelization as well as vanishing gradients) by introducing self-attention layers that work in parallel.

2.2.2 Attention Models Overview

Unlike the sequence-to-sequence models, the attention decoder now will get all the inputs generated by the encoder and at every time step, the decoder will focus on the appropriate vector. Each output vector from the encoder will have a weight that indicates how much

this vector is associated with this word (Abbildung 1).

How does the decoder know which step to focus on? that process is learned during the

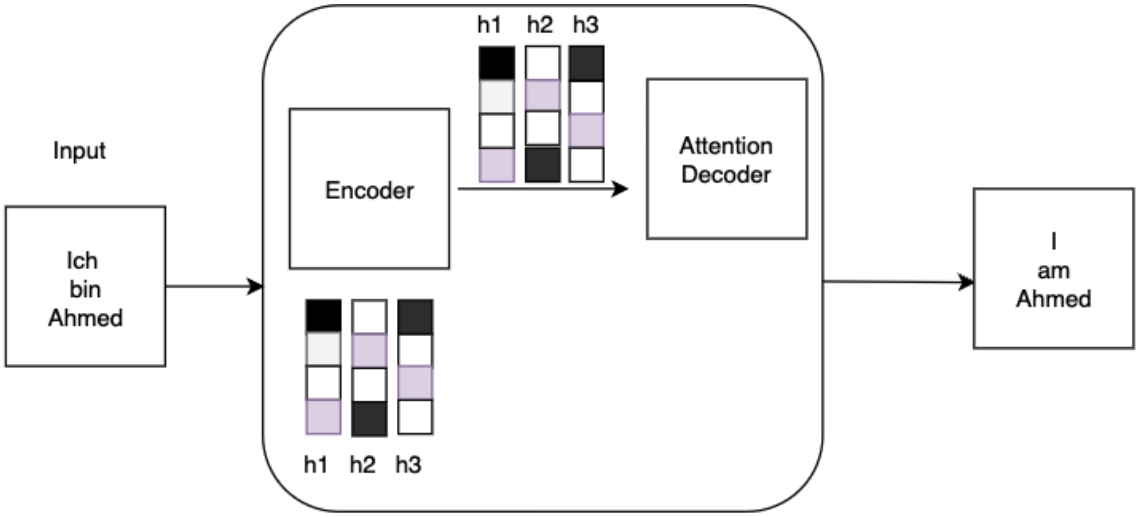


Abbildung 1: Overview of the attention models

learning phase, it doesn't go through them one by one, it learns how to skip the vectors and focus on the one related to the translated word. This is really helpful, especially in the machine translation tasks because not all languages have the order of words when you translate one sentence from one language into another. How does the attention decoder work? Now, since the decoder has all the hidden vectors of the encoder, it will score all the hidden states, If I feed all those scores to a softmax function, all these values will be positive and between zero and one, the score of each one of them indicates which vector the decoder should focus on while decoding the current output.

It is worth mentioning that there are different types of attention introduced in different papers like Additive Attention (Bahdanau) [5], Multiplicative Attention (Luong Attention) [15], Dot product attention, and multi-headed attention [22]. In the next section - when I will be discussing the architecture of the transformer - I will talk more about the dot product attention as well as the multi-head attention.

2.2.3 Transformer Architecture

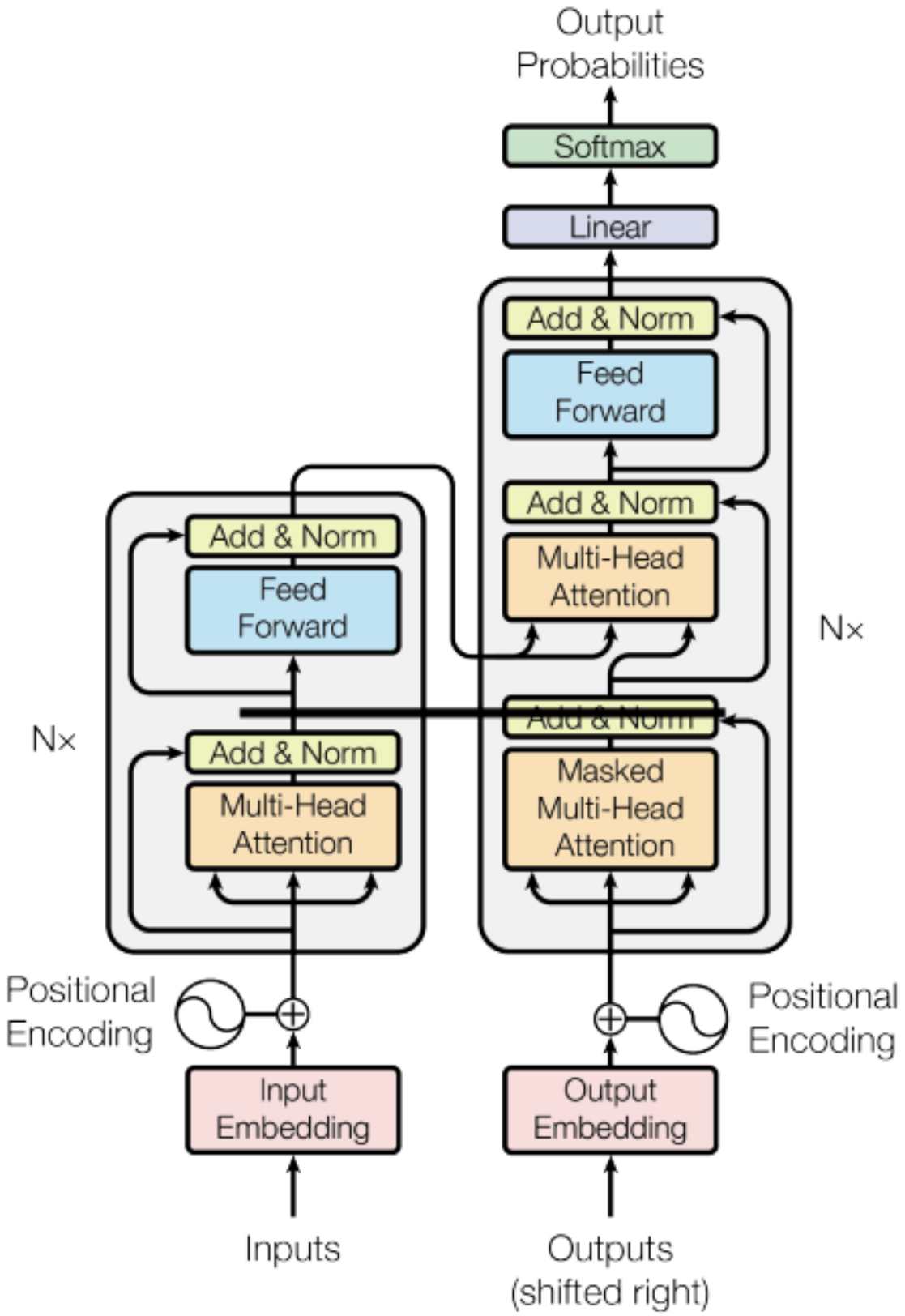


Abbildung 2: Transformer Architecture from [22]

.

Abbildung 2 shows the full architecture for the transformer introduced by Google Brain's team in Attention is all you need [22]. Since BERT's main component is the encoder part from that paper, I will focus more on the Encoder part here too and I will derive all the maths behind it.

The encoder consists of two main components, a self-attention layer and a feed-forward

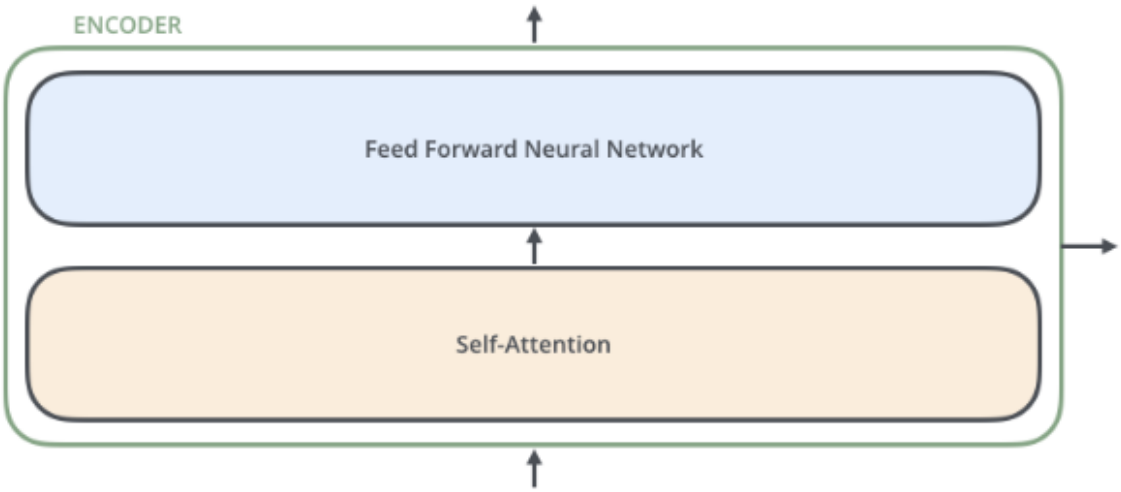


Abbildung 3: Transformer Architecture from [2]

.

neural network as shown in Abbildung 3. In the paper, they stacked 6 identical Encoders on top of each other. They also mentioned that 6 is not a magic number they got it from experimenting with different numbers, the same as what happens in hyper-parameters tuning. It is worth mentioning that the decoder also has the same number of decoders stacked together. All six encoders are identical in architecture but they do not share the same parameters. The input to the encoder first passes through the self-attention layer which helps the encoder to look at different words in the same input to relate the relationship between the current word and any other words (Like extracting the gender), then the output of the self-attention layer goes to the feed-forward network. Not all the encoders have an embedding layer, only the bottom one takes the input from the embedding layer and then the other encoders' input is just the output of the previous one.

The first step in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that I trained during the training process.

How do I calculate the self-attention output using vectors? The first in calculating the self-attention output as mentioned here [22] is to create three vectors from each of the encoder's input which in the first case will be our word embedding. These vectors are created by multiplying the embedding of each word by three different matrices that they trained during the training process.

We will denote the embedding each word as $x_1, x_2, ..., x_n$, and X will be the embedding matrix for the sentence I currently dealing with. Our three trained matrices are the queries matrix W^Q, the values matrix W^V, and the keys matrix W^K.

First, I will stack all our embedding inputs (for each word) to form the embedding matrix X Then, I will then multiply each of the queries, keys, and values matrices by X to get the new matrices Q, K, V which I will use to calculate the output of the self-attention layer.

$$Q = X \times W^Q$$
$$K = X \times W^K$$
$$V = X \times W^V$$

Now, since I have all the Q, K, and V matrices ready, I can start scoring each word against all the other words in the same sentence, as mentioned above, this will help the model to focus on how much each other word is important when encoding this one. The scoring matrix is calculated by taking the dot product of the queries matrix and the keys matrix.

$$\textbf{Scoring} = Q \cdot K$$
$$\textbf{Scoring} = Q \times K^T$$

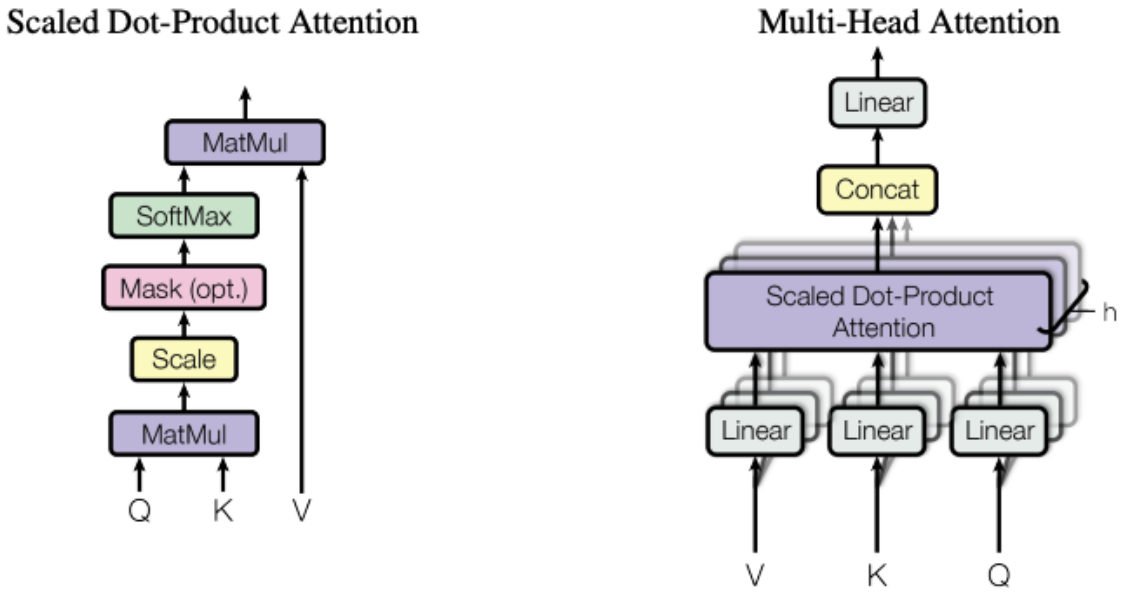


Abbildung 4: Scaled dot product attention, as well as Multi-Head Attention, explained in Attention is all you need [22]

.

In the paper, they introduced a new type of attention called Scaled dot product attention [22]. This type is similar to Dot-product attention [15], except for the scaling factor of $\frac{1}{\sqrt{d_k}}$ where d_k is the dimension of the queries Q and the keys K matrices. The authors of the paper mentioned that [22], dot-product attention is much faster and more space-efficient

in practice since it can be implemented using highly optimized matrix multiplication code.

Why the scaling factors d_k? With very large dimensions for the Q and K matrices, the magnitude of the dot product might be also very large which could cause some instability in the Softmax gradients. So, the main reason behind the scaling was to obtain much more stable gradients. Now, Since I have an overview of all the attention parameters, let's get back to the equations and drive the final attention output.

$$\textbf{Softmax}(\frac{Q \times K^T}{\sqrt{d_k}}).$$

Next, The output of the Softmax is multiplied by the values matrix V to give us the final equation for the self-attention layer which will be the input to our feed-forward network. I will denote the output of the attention layer as Z.

$$\textbf{Attention}(Q, K, V) = Z = \textbf{Softmax}(\frac{Q \times K^T}{\sqrt{d_k}})\ V.$$

Actually, the paper did not stop here, that was just an introduction to the Multi-Head attention mechanism they used in the paper. Instead of performing a single attention function with d_{model}-dimensional keys, values and queries, they found it beneficial to linearly project the queries, keys, and values h times with different, learned linear projections to d_k, d_q and d_v dimensions, respectively [22]. That helped improve the performance of the attention layers in many ways. It helped the model to focus on different positions to solve the issue I mentioned earlier in this document (The fridge did not pass because it was large), the model now knows that it references the fridge and not the door. Also, it gave the attention layer multiple representation subspaces because now I have multiple encoder/decoder layers stacked together.

Now, with the multi-head attention things got a little bit complex. Why? Because our feed-forward model expects just one attention output Z however now I have 8 (As mentioned here [22]) of them. They proposed a way to handle this issue by concatenating all the matrices that came out from each attention layer (Z_1, Z_2, ..., Z_9) into one big matrix Z. However, before passing this matrix to the Feed-Forward network, they multiply it with a matrix W^O following the same way they did for the Queries (W^Q), keys (W^K), and values W^V.

Before jumping in and discussing BERT, I need to explain two more concepts in the architecture of the transformer:

- Currently, I don't have any time steps like RNNs or LSTMs so, How do transformers encode the position of the sentences?
- What are the Add and Norm layers added after each self-attention layer?

To address the first question, They added the positional encoder to their architecture. A vector is added to each embedding, it helped the model to determine the position of each word and the distance between them when I project them on the queries, keys, and

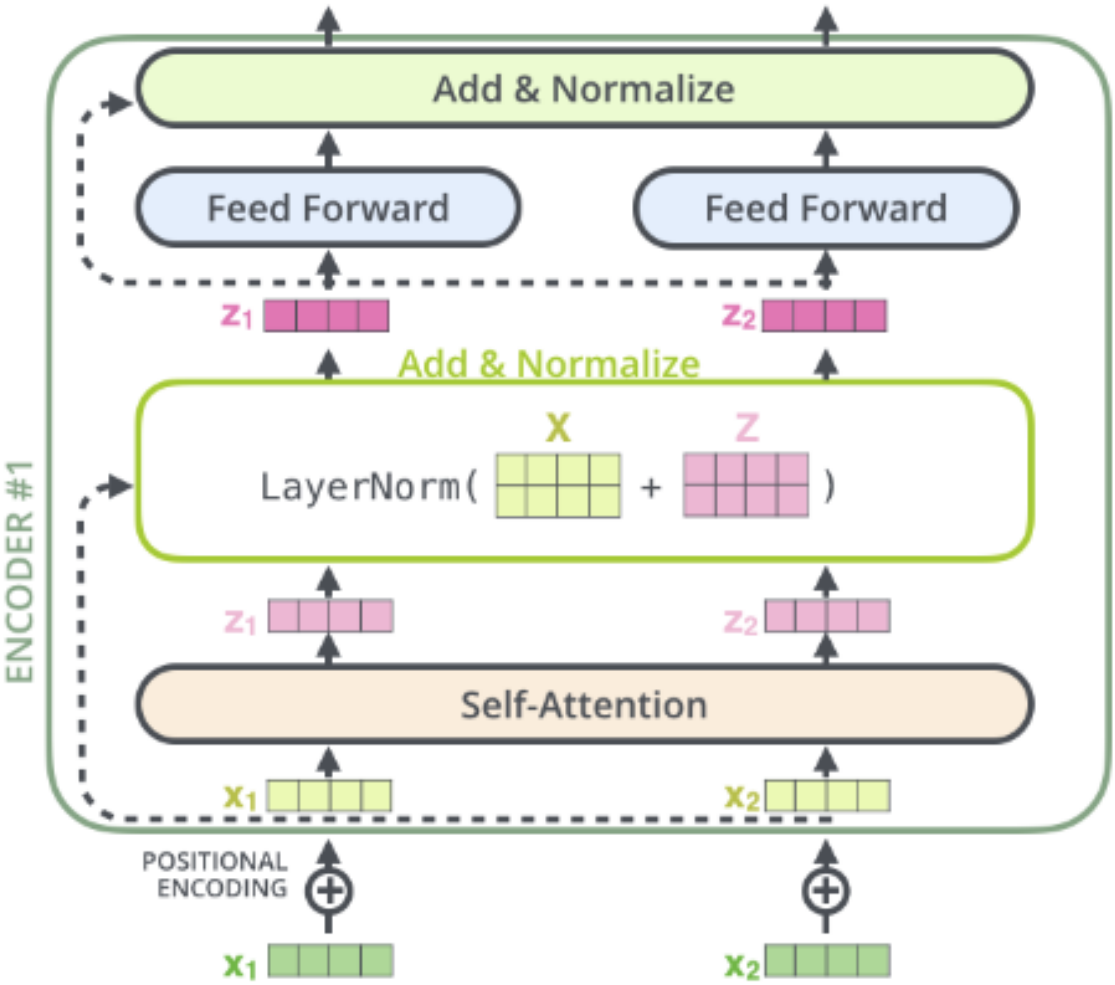


Abbildung 5: In-depth look for the Encoder from [2]

.

values. The last piece I will discuss in the encoder is the add and Normalized layer as shown in Abbildung 5. Thus the output of each sub-layer is:

$$\textbf{LayerNorm}(X + \textbf{Sublayer}(X)).$$

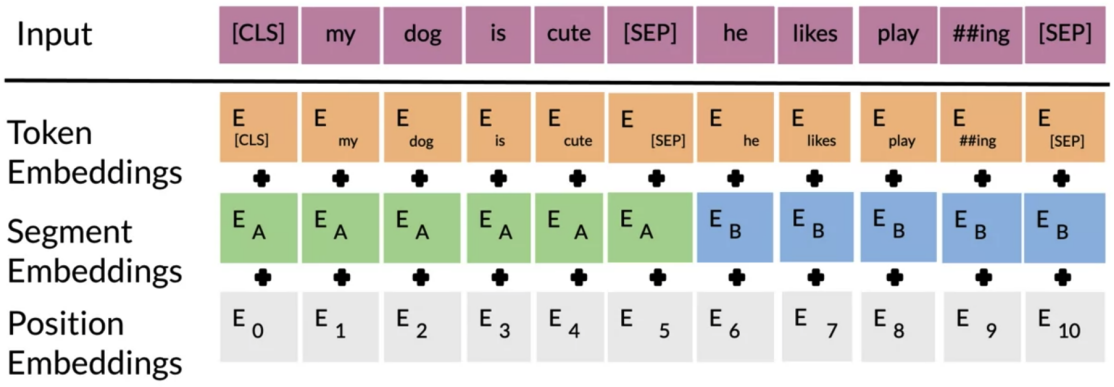
3 BERT, DeBERTa, ELECTRA, and DeBERTaV3

BERT [9] and DeBERTa [11] are both language models based on transformers architecture. Both models achieved state-of-the-art results in many downstream tasks such as machine translation, question-answering, and many more NLP tasks. I will start with BERT, and quickly go through the architecture but not in-depth because BERT encoders are the same as the encoders I described above in section 2. Here, I will focus more on pre-training and fine-tuning. Also, I will discuss DeBERTa and how it improved the performance by introducing a new type of attention called Disentangled attention, Here I will discuss the new math introduced in this paper [11]. Finally, I will compare the experiments of both models and which one outperformed the other.

3.1 BERT

Bidirectional Encoder Representations from Transformers [9] or BERT in short is a language model introduced by Google AI in 2018. BERT architecture is the same as the transformer explained in Attention is all you need and that's why they omitted the architecture from their paper. They built two different versions of the model, $BERT_{BASE}$ which was designed to be the same size as Open AI GPT, and the other version is $BERT_{LARGE}$. It is worth mentioning that despite $BERT_{BASE}$ and Open AI GPT having the same model size, they are both different in how each word relates itself to the surrounding words. BERT is a bidirectional model where each word can look forward and backward to have more context about the sentence, while the GPT Transformer uses constrained self-attention where every token can only attend to context to its left [9]. The architecture for $BERT_{BASE}$ is (L=12, H=768, A=12, Total parameters=110M) and for $BERT_{LARGE}$ is (L=24, H=1024, A=16, Total parameters=340M) as mentioned here [9] where L is the number of transformer blocks, H is the hidden size, and A is the number of self-attention heads.

Abbildung 6: BERT input in-detials [9]



.

Abbildung 6 shows BERT's input representation. The first component is the positional embedding which allows indicating the position of the word in a sentence then, I have the segment embedding which allows the model to know which sentence this word belongs to (Sentence A or sentence B). I need the segment embedding because one of BERT's fine-tuning tasks is the next sentence prediction, I will get to that again in the next subsection. Finally, the token embedding layer contains some special tokens like [CLS] to mark the start of the sentence as well as [SEP] to separate two sentences from each others. Simply, I need just to sum up all those three embeddings to get our input. I haven't talked yet about how to prepare the data that I will pass to the embedding layers, I will also discuss that later in the training set subsection.

3.1.1 BERT Pre-training

The pre-training tasks used the BookCorpus (800M words) [23] and English Wikipedia (2500M words), they mentioned that for Wikipedia they extract only the text passages

and ignore lists, tables, and header [9]. The pre-training process is split into two different tasks:

- Masked language model. Training a bidirectional model could be challenging because each word can look into its left and its right and simply can see itself which will make the predictions really trivial. The authors of BERT introduced a way to overcome that by masking some words (Masked language models). They masked 15% of the words and they even split that percentage into three different categories. 80% of those 15% masked words get the real mask token, 1.2% are replaced with a random token, and 1.2% are not touched but flagged for prediction so that during prediction the model does not know whether this token is replaced or is unchanged.
- Next sentence prediction. For the next sentence prediction, I have two sentences, sentence A and sentence B, they picked random sentences, and 50% of the time they put the right sentence after it and labeled it with isNext = True, the other 50% they just put a random sentence and labeled the is isNext label = false. They mentioned at the end that the sentence doesn't have to be a complete sentence, for example, sentence, A could be something like (I ran into my mom today and I invited.) i,e noncomplete sentences are also possible. The rule of thumb was that the length of the combined sentences should sum up to 512.

3.1.2 BERT Fine-tuning

For the hyper-parameters fine-tuning, they just kept everything the same as training and play around only with the batch size, learning rate, and number of epochs. for the batch size they tried [16, 32], [5e-5, 3e-5, 2e-5] for the learning rate and, [2, 3, 4] for the number of epochs. Also, BERT can be fine-tuned to work with a lot of tasks including but not limited to sentiment analysis, question-answering, and classification tasks.

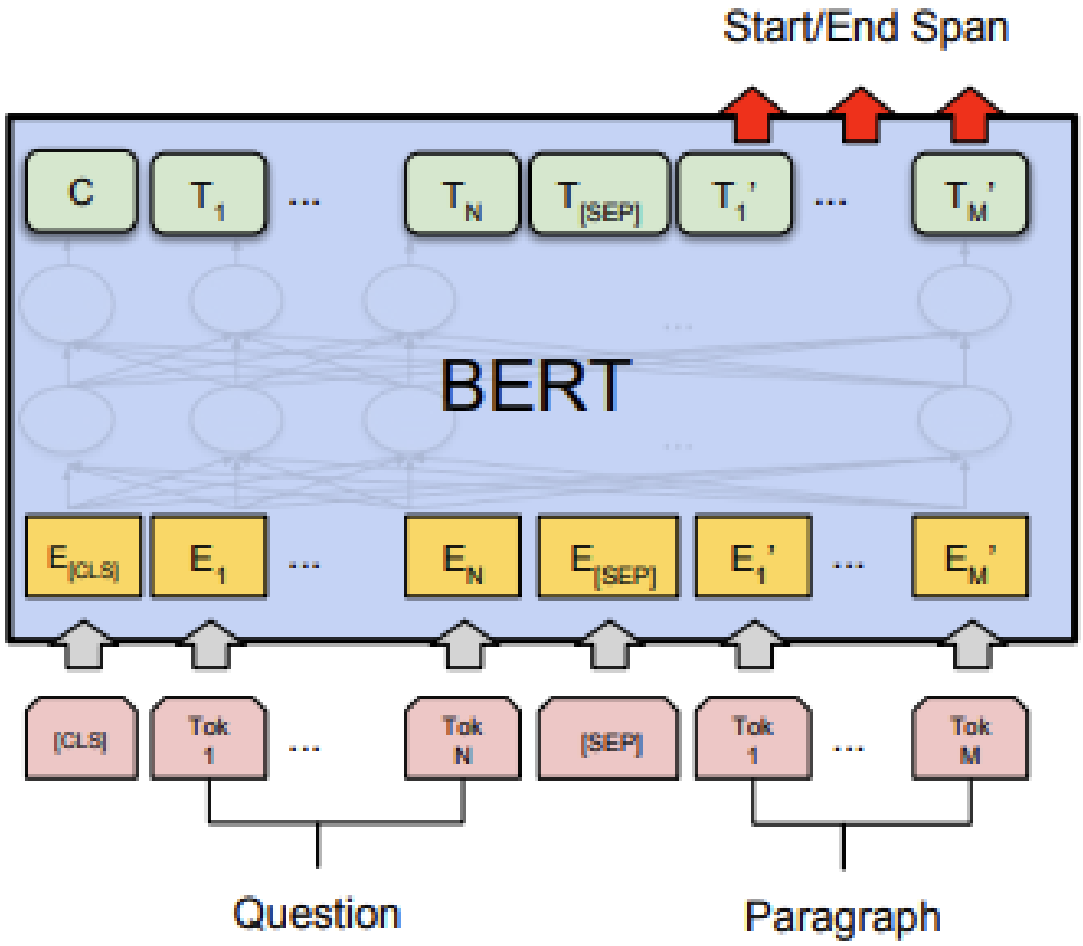
We will focus more on the question-answering task here. Also, I will show some results from the experiments ran on SQuAD v1.1 (The Stanford Question Answering Dataset) [13] later when we compare multiple models together like (DeBERTa - ELECTRA - DeBERTaV3). The question-answering here is not the same task that I can use for example with Amazon Alexa and Apple Siri to ask questions and give answers, it's more like giving BERT a question and a passage contains the answer, BERT will try to find the answer inside the given passage.

Generally the fine-tuning is pretty straightforward since the self-attention mechanism in the Transformer allows BERT to model any downstream tasks whether they involve single text or text pairs by swapping out the appropriate inputs and outputs [9]. For the Question-Answering task specifically, We have to pre-process our dataset into three different parts. First, The Questions we will feed our transformed, the context or the passage that not just contains the answer but also a big chunk of different topics Finally, The answer inside each of our contexts. As shown in Abbildung 7 each sentence must have a starting token. Also, the questions and the paragraphs are separated by a [SEP] token.

3.2 DeBERTa

DeBERTa [11] is the next iteration of BERT. it's also a language model that achieved the state-of-the-art results in many NLP tasks, DeBERTa outperformed BERT and T5

Abbildung 7: BERT input in-detials [9]



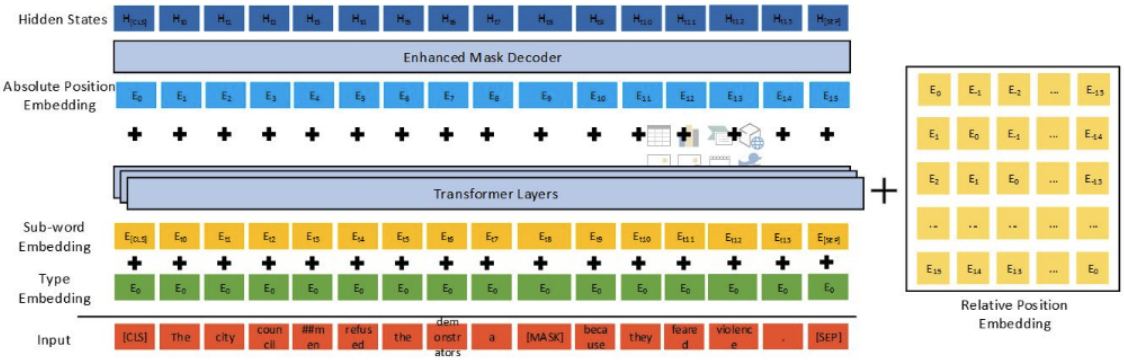
.

(Text-to-Text Transfer Transformer) [19] in many downstream tasks. DeBERTa has 1.5B parameters and compared to T5 which has 11B parameters, it's a massive success to outperform it with just 10% of the training parameters. DeBERTa was introduced by Microsoft in June 2019 and they approved two main parts from BERT.

First, they proposed a new technique called disentangled self-attention mechanism. BERT represented the input as the summation of its embedding and the positional embedding however, DeBERTa authors proved that using only the positional embedding is not enough and I need also to represent the relative position between words. So, instead of having just one vector, DeBERTa split it into two different vectors, the content and the position embedding.

The enhanced Mask Decoder (EMD) [19] was another method introduced by DeBERTa to improve BERT's performance. EMD references to predicting the word in the masked language model, instead of having the positional encoding before the transformers layers, DeBERTa just used the positional encoding right before the softmax layer i.e after the transformation layers. They mentioned that the early incorporation of absolute positions

Abbildung 8: DeBERTa architecture from [9]



.

used by BERT might undesirably hamper the model from learning sufficient information about relative positions [19].

3.2.1 DeBERTa Pre-training

DeBERTa followed the same procedures BERT used for pre-training. They also masked 15% of the words and split them into 80% to get the real token, 10% replaced with a random token, and 10% are not touched but also flagged for prediction purposes. For the training data, DeBERTa also used the BookCorpus and English Wikipedia however they added some extra training data from OPENWEBTEXT(public Reddit content [12] 38GB and STORIES10 a subset of CommonCrawl [21]. The total data size after data de-duplication [20, 19] is about 78GB. However, for $DeBERTa_{1.5B}$ they added some extra data, to sum up to 161 GB.

There are three different models for DeBERTa, $DeBERTa_{base}$, $DeBERTa_{Large}$ and $DeBERTa_{1.5B}$. To pre-train each one $DeBERTa_{base}$ took just 10 days, while $DeBERTa_{Large}$ and $DeBERTa_{1.5}$ took 20 and 30 days respectively[9].

3.2.2 DeBERTa disentangled attention

In this section, I will explain the math behind the disentangled attention for DeBERTa. Let's first start by recapping the equations I discussed before for BERT and Transformers. I will be using the same equations however, I will just introduce the new concepts of the disentangled attention using the same equations and notations. Previously I used the scoring function to address the Q and K matrices and our final scoring equation was:

$$\textbf{Scoring} = Q \cdot K$$

$$\textbf{Scoring} = Q \times K^T$$

Now, Q and V, instead of having just one matrix, are split into two different matrices. H represents the content of the current word, P represents the relative position of the token. Let's pick a simple sequence and explain our equations on it, Äm I good at playing football?". Now our scoring function will be between any two words I pick. let's pick the words Am at position 0 and playing at position 4, Now our scoring function for those two Tokens will be

$$\textbf{Scoring} = Q \times K$$

$$\textbf{Scoring} = \{H_0, P_{0|4}\} \times \{H_4, P_{4|0}\}$$

When I multiply the vectors above our equation will as follow:

$$\textbf{Scoring} = H_0 H_4^T + H_0 P_{4|0}^T + P_{0|4} H_4^T + P_{0|4} P_{4|0}^T$$

Let's dive deep and see what each of the four components above tells us.

- The first one $H_0\ H_4^T$ is a content-to-content component. this is the classical attention used in BERT, For example in our sequence the word am tries to know the nouns in the sentence, it's pretty obvious here that the noun will be I but imagine another verb like go, this component will help the token to decide whether the verb should go or goes or went according to the content of the other tokens.

- the second component $H_0\ P_{4|0}^T$ is the content-to-position. This component helps each token to be aware of what's around it. Here in our case, for example, the token Am knows that it starts a sentence which boosts the probability of being a question.

- The third component $P_{0|4}\ H_4^T$ is the position-to-content. This one helps each token to send proper info to the target token, It's kinda asking a question like Ökay, I'm the word Am at position zero, what kind of info do I need to send for word playing at position 4".

- The last one $P_{0|4}\ P_{4|0}^T$ is the position-to-position component. this component tries to help send information from a token at position 0 to a token at position 4 and since here I are just talking about relative positions, they decided to remove it from the equation [9].

Now, after removing the last position-to-position component, I can write a general form of our equations as follow:

$$\textbf{Scoring} = H_i H_j^T + H_i P_{j|i}^T + P_{i|j} H_j^T$$

Same as BERT, each matrix of the Q, K, and V has its own projection matrices W_q, W_k and W_v but now I two different matrices for each of them (H and P), our equations will be as follow:

$$Q_c = HW_{q,c}, Q_r = PW_{q,r}, K_c = HW_{k,c}, K_r = PW_{k,r}, V_c = HW_{v,c}.$$

$$\textbf{Scoring} = Q^c K^{cT} + Q^c K^{rT} + K^c Q^{rT}$$

Each of the Q^r and K^r is accompanied by a distance measure δ which is defined as follow:

$$\delta(i,j) = \begin{cases} 0 & \textbf{for i - j} \leq \textbf{-k} \\ 2k-1 & \textbf{for i - j} \geq \textbf{k} \\ i-j+k & \textbf{others}. \end{cases}$$

where k is the sequence length. K is also dependent on the number of attention heads, They started with just 12 attention heads (Transformer blocks) in $DeBERTa_{base}$ and ended up with 24 attention heads in $DeBERTa_{1.5B}$ [9]. I can calculate the maximum distance using the following equation:

$$\textbf{max tokens} = 2(k-1)L$$

This means that in theory, $DeBERTa_{large}$ with a sequence length $k = 512$ and 24 attention heads (L) can handle up to 25,528 tokens.

3.2.3 DeBERTa Fine-tuning

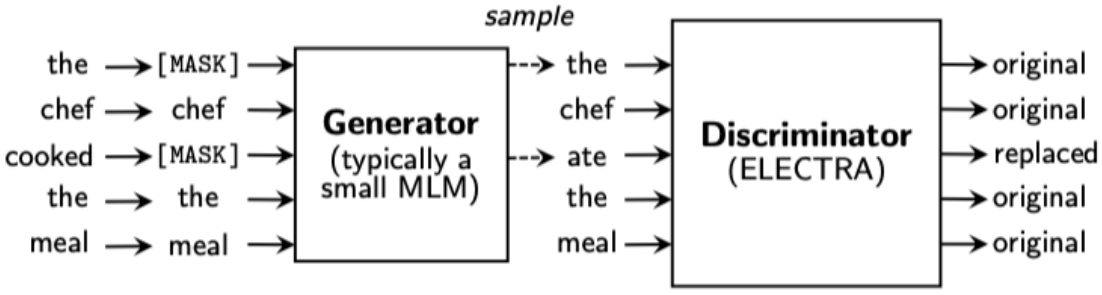
For the hyper-parameters fine-tuning, like BERT, they tried different batch sizes [16, 32, 48, 64] also different learning rates [5e-6, 8e-6, 9e-6, 1e-5] They also mentioned that the maximum number of epochs was 10 but they didn't mention the numbers they tried before that. Below (Tabelle 2) you can see all the hyper-parameters fine-tuning.

3.3 ELECTRA Overview

Efficiently Learning an Encoder that Classifies Token Re- placements Accurately or ELECTRA in short introduced by a team of 4 people working at Stanford as well as Google Brain [8]. The authors of ELECTRA introduced a new technique to pre-train language models using Replaced Token Detection (RTD). The main idea is to first some words probabilistically using a softmax and then pass this masked input to a generator. The Generator (G) will try to predict the masked words, sometimes it will correctly predict the other words and sometimes it will fail which results in a corrupted sequence.

Here comes the role of discrimination (D), the discriminator will take the corrupted sequence and try to predict whether or not the generated token is original, Abbildung 9. For every token in the corrupted input, it will predict if this is the original token or a corrupted token [8]. What makes this method special? In BERT, when I generated the tokens, they were all randomly generated using static masking (See section 3.5) and the goal was to predict those tokens. That means that some tokens will not be considered for the loss functions (All the tokens that have not been masked), which also means that BERT needs much more data and computational resources than ELECTRA.

Abbildung 9: ELECTRA overview from [8]



.

Each of the discriminatory and the generator consists of a transformer network and as I mentioned earlier, ELECTRA uses a probabilistic model to mask each token as follows:

$$P_G(x_r|x) = softmax(E(x_r)^T Z_G(x)_r).$$

where $E(x_r)$ is the token embedding of a word at position r and Z_G is the output of the generator (a Transformer network) which maps a sequence of input tokens into a sequence of contextualized vector representations [8]. They used maximum likelihood loss instead of the GANs adversarially to fool the discriminator, they claimed that the traditional GANs training process is not suitable for text, moreover, they tried the concept and it performed worse than maximum-likelihood training [8]. Also, it's worth mentioning that all the weights were shared between the discriminator and the generator.

The issue was that to share the weights between the discriminator and the generator, they both must be in the same size. However, the authors of ELECTRA found that it's more efficient to have a small generator. So, they decided to share only the embeddings between the generator and the discriminator [8]. The reason behind that is, Language models are good in learning those representations. I will discuss the weight sharing more in the next section when I compare different results from ELECTRA and DeBERTaV3.

The loss functions for both the generator and discriminator are as follow:

$$L_{MLM} = \mathbb{E}(-\sum_{i \in c} logP_{\theta_G}(\bar{x}_{i,G} = x_i|X_G)).$$

Where θ_G represents the generator parameters and x_i is the token at position i out of a complete sequence X_G. The loss function for the discriminator:

$$L_{RTD} = \mathbb{E}(-\sum_{i} logP_{\theta_D}(\mathbf{1}(\bar{x}_{i,D} = x_i)|X_{D,i})).$$

where $\mathbf{1}$ is the indicator function and θ_G represents the generator parameters [8]. Both loss functions are added together with a discriminator weight λ which was set to 50.

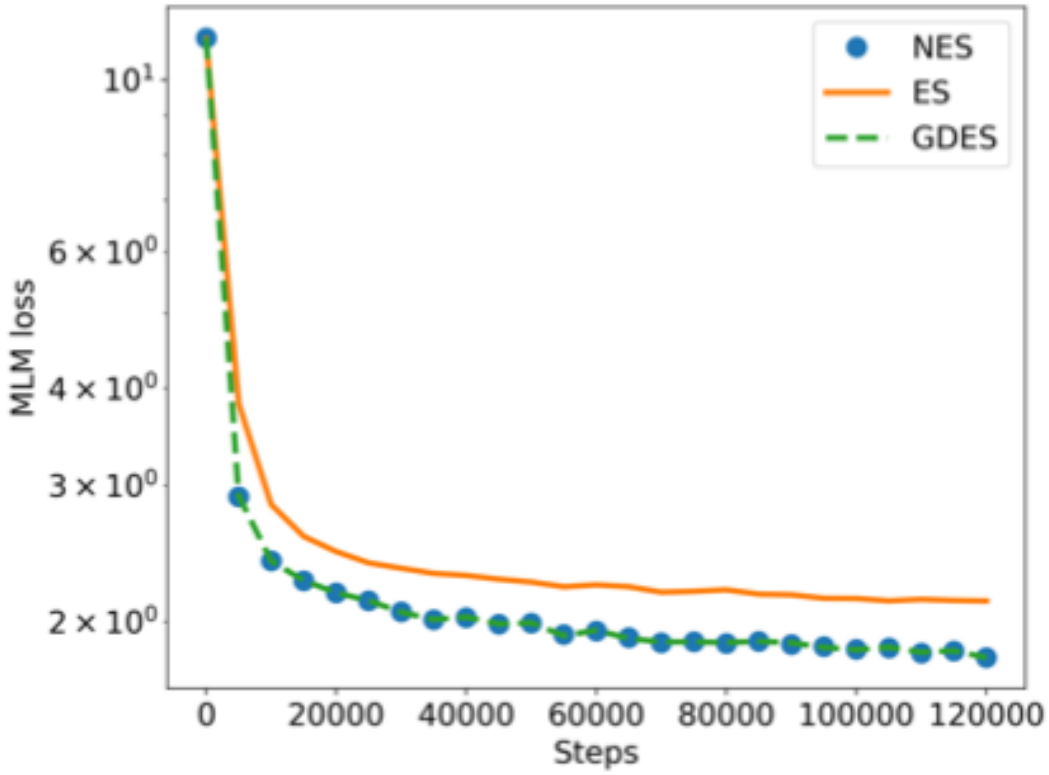
$$\textbf{Total Loss} = \frac{\partial L_{MLM}}{\partial E} + \lambda \frac{\partial L_{RTD}}{\partial E}.$$

3.4 DeBERTaV3 Overview

DeBERTaV3 [10] is an enhancement for the original DeBERTa using the techniques proposed in ELECTRA (RTD). DeBERTaV3 also improved another aspect of the weight sharing between the generator and the discriminator by introducing a new method called gradient-disentangled embedding sharing (GDES) [10]. Sharing the weights was not a good idea because each of them (the generator and the discriminator) has a different task, the generator's task is to predict the masked tokens out of a sequence and the discriminator is being used to detect whether this is a fake or real token. So, somehow the embeddings were going in two opposite directions creating the "tug-of-war" dynamics [10].

First, they proved their claims by implementing another variant of ELECTRA using the No Embedding Sharing (NES) technique. NES improved the generator's efficiency and

Abbildung 10: MLM training loss of the generator with different word embedding sharing methods from [10]



.

reduced the training time as shown in Abbildung 10. However, when they tried to fine-tune the discriminator in some downstream tasks, it performed worse than the original ELECTRA which used only embedding sharing (ES).

As shown in Abbildung 11 part b, No Embedding sharing (NES) updates the generator and the discriminator separately. So, instead of updating all the parameters in just one backward pass like Embedding Sharing (ES), NES will update E_G and θ_G with regards to L_{MLM}, then E_D and θ_D will be updated via the backward pass with regards to L_{RTD} (for the discriminator). As I discussed earlier in ELECTRA, sharing the embedding would allow for some improvements in some downstream tasks like Question-Answering. So, DeBERTaV3 introduced a new technique called gradient-disentangled embedding sharing (GDES). Using GDES, I can still share the weights between the generator and the discriminator but with a limit. The gradients of the generator is calculated only based on L_{MLM}. Does that sound weird? Because I just mentioned that GDES uses embedding sharing between the generator and the discriminator. Well, They share the weights but at the end of the training process. So, L_{MLM} used to update both E_D and E_G however, L_{RTD} used to update only E_D.

$$E_D = sg(E_G) + E_\Delta.$$

Tabelle 1: Fine-tuning results on SQuAD v2.0 from [10]

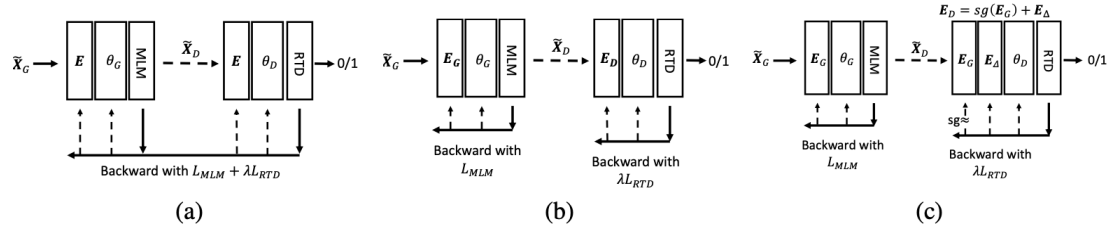
Model (DeBERTa + RTD_{base})	SQuAD v2.0 (F1/EM)
ES	86.3/83.5
NES	85.3/82.7
GDES	87.2/84.5

Tabelle 2: Hyper-parameters for fine-tuning DeBERTa on down-streaming tasks [9].

Hyper-parameter	$DeBERTa_{1.5B}$	$DeBERTa_{Large}$	$DeBERTa_{Base}$
Dropout of task layer	0,0.15,0.3	0,0.1,0.15	0,0.1,0.15
Warmup Steps	50,100,500,1000	50,100,500,1000	50,100,500,1000
Learning Rates	1e-6, 3e-6, 5e-6	5e-6, 8e-6, 9e-6, 1e-5	1.5e-5,2e-5, 3e-5, 4e-5
Batch Size	16,32,64	16,32,48,64	16,32,48,64
Weight Decay	0.01	0.01	-
Maximun Training Epochs	10	10	10
Learning Rate Decay	Linear	Linear	Linear
Adam β_1	0.9	0.9	0.9
Adam β_2	0.999	0.999	0.999
Gradient Clipping	1.0	1.0	1.0

What's *sg*? This is a stopping gradient operator which only allows gradients propagation through [10]. E_Δ is initialized as a zero matrix and for each sequence, the generator will first generate the output to the discriminator which is the predicted words that were initially masked using our probability function (discussed in ELECTRA). Then, I will not wait for the discriminator to run its forward passes instead, I will do a backward pass to calculate the gradients for E_G with respect to L_{MLM}. Next, I run the discriminator forward pass using the output of the generator and run a backward pass with respect to the RTD loss to update E_D by propagating gradients only through E_Δ [10]. After model training, E_Δ is added to E_G and the sum is saved as ED in the discriminator [10]

Abbildung 11: differences between different embedding sharing techniques from [10]



.

Tabelle 3: F1 accuracy for static and dynamic masking for BERT on SQuAD 2.0 [14].

Masking	SQuAD 2.0
static	78.3
dynamic	78.7

Tabelle 4: F1 accuracy for static and dynamic masking for BERT on SQuAD 2.0 [14].

Model	SQuAD 2.0 (F1/EM)	Number of params (M)
$BERT_{large}$	81.8/79	345
$ELECTRA_{large}$	-/88.1	335
$DeBERTalarge$	91.5/89.0	305
$DeBERTa1.5B$	92.2/89.7	1500

3.5 Experiments

In this section, I will discuss the differences between each model, and which parts got improved by which model, I will also compare results and see what made each model unique. When I compare some experiments, I will focus on the question-answering task.

Let's talk about the differences between DeBERTa and BERT. Besides the disentangled attention and EMD discussed earlier, BERT uses static masking. What does this mean? It means that BERT relies on randomly masking and predicting tokens using a single static mask, which was generated during data pre-processing [14]. To avoid using the same mask for each training instance in every epoch, training data was duplicated 10 times so that each sequence is masked in 10 different ways over the 40 epochs of training. Thus, each training sequence was seen with the same mask four times during training (mentioned here [14]).

DeBERTa uses dynamic masking which means, they generate the masking pattern not just once but every time they feed a sequence to the model. It helps them to train the model for an extended period of time and a large number of epochs. The authors of RoBERTa also showed how the dynamic masking improved BERT itself, see Tabelle 2.

ELECTRA then came and introduced a new concept (RTD) and that was the main difference between ELECTRA and DeBERTa. The problem was that ELECTRA tried to optimize multi-learning tasks (Predict the token - Classify whether or not this is the real token). Those tasks were pulling the tokens in two different directions leading to the tug-of-ward problem. DeBERTaV3 introduced a new technique (GDES) that made use of sharing the weights but with a limit. First as shown in Abbildung 10, they proved how efficient their algorithm is without sharing the embeddings between both tasks, then they ran some experiments on some downstream tasks and DeBERTaV3 outperformed all the mentioned models.

For the question-answering task, DeBERTa outperformed all the other models with the same segment (model params). SQuAD v2.0 is fine-tuned using the sequence tagging

tasks, where a token classification head is plugged on top of the hidden states of each token at the last layer, SQuAD v2.0, ReCoRD fine-tuned as sequence tagging tasks, where a token classification head is plugged on top of the hidden states of each token at the last layer [10]. Also, DeBERTaV3 outperformed some other models not just in question-answering, but also in tasks like Named Entity Recognition, RACE, SWAG, and MNLI, DeBERTaV3 is able to outperform all the other models on these tasks [10].

4 AraBERT (Non-English LM)

So far we have discussed a lot of language models and how each one achieved the state-of-the-art in most of the NLP challenges however, all these models are just for one language (English). In this section, we will discuss another language model variation similar to BERT which is AraBERT [4]. Fining a dataset for BERT was not challenging at all because there are massive resources available in English the challenge in BERT was to find suitable data. However, in Arabic, there are not many available texts and corpus compared to that for English. Moreover, the Arabic language is known for its lexical sparsity which is due to the complex concatenative system of Arabic, words can have different forms and share the same meaning [4]. We will discuss each of these challenges in the upcoming sections we will also compare AraBERT to mBERT [18] which is a multilingual version of BERT that supports multiple languages.

AraBERT uses the same configurations used by $BERT_{Base}$ which are, 12 attention heads, 512 maximum sequence length, and a total of 110M params [4]. Also, some extra preprocessing steps had to be taken because of the complexity of the Arabic language, let's start by discussing the pre-training process for AraBERT.

4.1 AraBERT pre-training

The pre-training dataset used for AraBERT is around 24GB of text and they came from different resources. They had to manually scrap a lot of articles and news from different websites to build their dataset. In addition to that, they used two popular Arabic Corpus El-Khair which contains 1.5 billion words and OSIAN corpus [6] which has almost 1 billion words from different Arab countries. The final size of the dataset was 70 million sentences but it still, needs to be processed more to fit the Arabic language representation.

As we mentioned earlier, there are a lot of words in Arabic that would give you the same meaning, for example, a word like "اللغة - Alloga"could be written also as "لغة - loga"because the definite article in Arabic (which is equivalent to the in English) is not an intrinsic part of the word [4]. So, they segmented the words using Farasa [1] into stems and trained a SentencePiece in uni-gram mode. To evaluate how this tokenization works, they introduced another version of AraBERTv0.1 which contains only the words without any segmentation, and compared it to the original AraBERTV.1. AraBERT followed BERT in the pre-training objective task (Masked Language models), it also followed the same criteria for masking 15% of the tokens, 80% of the masked tokens get the [MASK], 10% replaced with random token and the last 10% left untouched.

Tabelle 5: Performance of AraBERT on Arabic Question-Answering task compared to mBERT from [6].

Task	metric	mBERT	AraBERTv0.1/v1
QA(ARCD)	Exact-Match	**34.2**	30.1/30.6
	macro-F1	61.3	61.2/**62.7**
	Semi-Match	90.0	**93.0**/92.0

4.2 AraBERT fine-tuning

AraBERT fine-tuned for three different NLP tasks, Sequence Classification, Named Entity recognition, and Question-Answering. I will focus here on Question-Answering in both the fine-tuning and the experiments sections.

How AraBERT works with Question-Answering? The basics are the same, we provide the model with a question and a paragraph containing the answer, AraBERT will try to predict the start and end tokens for the answer under the condition that the start token will always come before the end token. During the training process, the embedding of each token is passed into two classifiers. The dot product between the embedding and the classification output is then fed into a softmax layer which decides, which token should be the start token according to a probability distribution. Similarly, the end token goes through the same process under the same condition we mentioned earlier (End-token comes after Start-token) [6].

AraBERT was able to outperform mBERT in some NLP tasks including question-answering. But, before we jump into the results, it's worth comparing the resources used by each of them. For instance, mBERT didn't use tokens as much as AraBERT used. AraBERT used around 24GB of data in comparison with 4.3GB used for mBERT. Also, the vocab size used by AraBERT resulted in 64k unique tokens however, mBERT trained only on 2K tokens [6]. Not to mention, AraBERT trained only for the Arabic language, and mBERT trained for 104 different languages. Finally, the preprocessing done for AraBERT took into consideration the complexity of the Arabic language.

4.3 AraBERT and mBERT Experiments

Taking into consideration all the differences between AraBERT and mBERT in the previous section, AraBERT obviously outperformed mBERT in the downstream tasks AraBERT trained on. AraBERT fine-tuned only on Question-Answering, Named Entity Recognition, and Sequence Classification. Tabelle 5 shows the results from the AraBERT paper [6].

AraBERT has been evaluated on the Arabic reading Comprehension Dataset (ARCD) [16]. This dataset has 1400 questions on Wikipedia articles, moreover, it has almost 3000 questions machine-translated from SQuAD (Arabic-SQuAD). AraBERT trained on the whole Arabic-SQuAD and just 50% of the ARCD dataset.

AraBERT outperformed BERT in the Sentence-Match (by roughly 2%) and F1 metrics

but not the exact match. However, the difference was only one or two words that do not have significant performance on the overall answer.

5 Conclusion

This work explained the important aspects of some recent language models. Of course, We didn't cover all the other models like T5, GPT-3, ChatGPT, and many more. We compared the performance of all the models we mentioned in this work, which parts were used by most of them, and which parts got improved by each newly released model. Hence, We can confidently tell how the transformers opened the door to massive language models, The Transformers proposed by Google Brain in the Attention is all you need was the same component used by all the language models. Transformers solved the puzzle of the slow predecessor time-series models like LSTMs, RNNs, and GRUs, It also improved the computational performance and introduced a new way to parallelize sequences on different resources (GPUs).

Recently language models are able to outperform humans in some tasks like question-answering and machine translation. ChatGPT which just came out was is to solve some complex problems from LeetCode (a problem-solving website for software engineers) in just seconds, these problems marked as HARD and might take a couple of hours to be solved by humans. Also, ChatGPT is able to give you detailed answers about how to deploy your machine-learning models on Amazon Web Services (AWS) or Google Cloud Platform (GCP). Moreover, the models were able to solve the assignments for some students and solve some online assessments for some interviewees. On the other hand, ChatGPT sometimes was not able to solve some really trivial math problems like Ï was 6 and my Sister was half of my age, when I will be 70, How old will she be?". That's actually a good and bad sign, First, We are on the right track to achieving better results in machine learning and language models but, those models will never replace humans, maybe in some traditional jobs but not the science ones.

In the experiments section, We have seen how each model is getting smaller and achieving better results which means, We are on our way to not caring much about how many resources we have instead, We should focus more on how to prove our current work. Lastly, I would like to mention that, Language models are being used by us daily, whether or not we admit that we do use language models in our google searches, google translations,s and many more daily tasks which shed the light on how important the language models are to everyone's life.

Literatur

[1] Ahmed Abdelali, Kareem Darwish, Nadir Durrani, and Hamdy Mubarak. Farasa: A fast and furious segmenter for Arabic. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 11–16, San Diego, California, June 2016. Association for Computational Linguistics.

[2] Jay Alammar. The illustrated transformer, 2018.

[3] Wissam Antoun, Fady Baly, and Hazem Hajj. Arabert: Transformer-based model for arabic language understanding. 05 2020.

[4] Wissam Antoun, Fady Baly, and Hazem Hajj. Arabert: Transformer-based model for arabic language understanding, 2020.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[6] Ramy Baly, Alaa Khaddaj, Hazem Hajj, Wassim El-Hajj, and Khaled Bashir Shaban. Arsentd-lev: A multi-topic corpus for target-based sentiment analysis in arabic levantine tweets, 2019.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[8] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. Electra: Pre-training text encoders as discriminators rather than generators, 2020.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[10] Pengcheng He, Jianfeng Gao, and Weizhu Chen. Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing, 2021.

[11] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention, 2020.

[12] Shota Kato, Kazuki Kanegami, and Manabu Kano. Processbert: A pre-trained language model for judging equivalence of variable definitions in process models*. *IFAC-PapersOnLine*, 55(7):957–962, 2022. 13th IFAC Symposium on Dynamics and Control of Process Systems, including Biosystems DYCOPS 2022.

[13] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents, 2014.

[14] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[15] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[16] Hussein Mozannar, Karl El Hajal, Elie Maamary, and Hazem Hajj. Neural arabic question answering, 2019.

[17] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018.

[18] Telmo Pires, Eva Schlinger, and Dan Garrette. How multilingual is multilingual bert?, 2019.

[19] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019.

[20] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2019.

[21] Trieu H. Trinh and Quoc V. Le. A simple method for commonsense reasoning, 2018.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[23] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 19–27, Dec 2015.