Assessing Efficiency in Domain-Specific Transformer Models: Comparing, Pretraining, and Finetuning Small-Scale Transformer Models within Hardware Limitations for Financial NLP

Lukas Bierling

January 2024

This thesis embarks on a critical examination of transformer-based models optimized for the financial sector under computational constraints. The study systematically explores the adaptability and performance of various transformer architectures, including BERT, Reformer, and a custom-designed reversible dilated BERT, in handling domain-specific financial texts. A significant portion of this research is dedicated to the process of pretraining these models on specialized datasets to ascertain their effectiveness in financial sentiment analysis and topic classification tasks.

The findings indicate a nuanced landscape where the adjusted architectures, such as the Reformer and the reversible dilated BERT, show limited benefits in environments constrained by resources, particularly with smaller models and shorter sequence lengths of 128 tokens. This observation suggests that the potential advantages of these architectural modifications become more pronounced with longer sequence lengths, which were not the focus of this study due to the imposed hardware limitations.

Conversely, the Electra pretraining method, applied to the BERT model, demonstrated a promising pathway towards achieving high efficiency and robust finetuning outcomes within the specified constraints. This approach underscores the feasibility of deploying sophisticated NLP models in the financial sector, even when computational resources are limited, by leveraging domain-specific pretraining strategies.

Through a detailed analysis and comparison of transformer model architectures and their pretraining and finetuning performance, this thesis contributes valuable insights into the development of efficient NLP solutions tailored to the financial industry. It highlights the critical balance between model complexity, computational resource availability, and the specific requirements of financial NLP tasks, offering guidance for future research in this vital intersection of technology and finance.

Contents

List of Tables 6

List of Figures 7

1 Introduction 11

2 Foundation 13

2.1 Tokenization in NLP . . . 13
2.2 Word Embeddings . . . 14
2.3 Recurrent Neural Networks and its problems . . . 14
2.3.1 Introduction to Recurrent Neural Networks . . . 14
2.3.2 Training of RNNs . . . 16
2.3.3 Vanishing and Exploding Gradient Problem in Recurrent Neural Networks . . . 16
2.4 Attention in Recurrent Neural Networks . . . 17
2.4.1 Encoder-Decoder RNNs . . . 17
2.4.2 Attention based Encoder-Decoder RNNs . . . 19
2.5 Attention is all you need . . . 22
2.5.1 Introduction . . . 22
2.5.2 Model architecture . . . 22
2.5.3 Positional encoding . . . 23
2.5.4 Self-attention in transformers . . . 24
2.5.5 Multi Head Attention . . . 25
2.5.6 Attention Block . . . 27
2.5.7 Comparison to attention in RNNs . . . 28
2.5.8 Revolutionary impact . . . 29
2.6 Language models . . . 29
2.7 Transfer Learning . . . 31
2.8 Emerged Models . . . 32
2.8.1 Encoder models . . . 32
2.8.2 Decoder models . . . 32
2.8.3 Encoder-Decoder models . . . 33
2.9 Detailled overview of BERT-type encoder models . . . 33
2.9.1 BERT architecture . . . 33
2.9.2 BERT pretraining . . . 35
2.9.3 BERT fine-tuning . . . 35

2.9.4 BERT variants . . . 36
2.10 Summary . . . 37

3 Limits of Attention 38
3.1 Quadratic complexity . . . 38
3.2 Consequences of the quadratic complexity of self-attention . . . 39

4 Addressing Quadratic Complexity Constraints in Attention Mechanism 40
4.1 Introduction . . . 40
4.2 Sparsifying the attention matrix . . . 41
4.2.1 Sliding window Attention . . . 41
4.2.2 Dilated sliding window attention . . . 42
4.2.3 Global attention connection . . . 43
4.2.4 Locality Sensitive Hashing . . . 45
4.3 Reversible residual layers . . . 47
4.3.1 Backpropagation summary . . . 47
4.3.2 Residual networks summary . . . 48
4.3.3 Reversible layer . . . 48
4.3.4 Backward pass for reversible layers . . . 49
4.3.5 Backpropagation in reversible layers . . . 52
4.4 Electra Pretraining . . . 53
4.4.1 Review of current Pretraining Techniques for Masked LMs . . . 53
4.4.2 Generative and Discriminative models . . . 53
4.4.3 Discriminative Pretraining with Electra . . . 54
4.4.4 Training algorithm for Electra . . . 56
4.5 Summary . . . 57

5 Pretraining the model on financial text data 58
5.1 Introduction . . . 58
5.2 Tokenization and Embeddings . . . 58
5.3 Proposed Models . . . 58
5.3.1 Reversible and Non-Reversible Dilated BERT . . . 58
5.3.2 Reformer . . . 59
5.3.3 Plain BERT . . . 60
5.4 Domain specific data . . . 60
5.5 Computational ressources . . . 61
5.6 Training setup . . . 61
5.6.1 General Configuration . . . 61
5.7 Pretraining results . . . 62
5.7.1 Relative comparison of ressources . . . 62
5.7.2 Relative comparison of pretraining performance . . . 63

6 Finetuning on financial data 67
6.1 Financial sentiment analysis . . . 67

6.2 Financial news topic classification 68
6.3 Finetuning conlusion 69

7 Conclusion **70**

Bibliography **72**

A Repository of the implementation **76**

List of Tables

1 Comparison of model ressources during training 62
2 Comparison of model performance during pretraining after 70,000 steps . 66

List of Figures

1 Schematic visualization of a RNN cell. 15

2 Unrolled RNN for three time steps by creating a copy for each time step (Pascanu, Mikolov, and Bengio (2012b)). The hidden output of time step $t-1$ is input for the hidden state at t and so on. This is how the temporal information gets passed through the network. 16

3 Visualization of an Encoder-Decoder RNN. The representation vector c gets passed through every hidden state h_t of the Decoder. Note that only three hidden states were used here for simplification, but in real scenarios these can be extended to much larger sequences. 19

4 Simplified visualization of an attention backed Encoder-Decoder RNN. One can see that c_i is influenced by all encoder hidden states over α_{ti}. Input length T_x and output length T_y may differ. To maintain a clear overview, the connections from the encoder hidden states to each context vector are left out. In this step the weighted sum of the encoder hidden states would be computed. 21

5 Attention scores from english-french translation. Each output word (french) attends to all the input words (english) to a specific degree. The brighter the cell α_{ti}, the higher the attention score between word t and i. (Vaswani et al. (2017a) 21

6 The transformer architecture used in the original paper (Vaswani et al. (2017a) 23

7 Visualization of the attention mechanism. Assuming that the input sequence is of length n. The key, query, and value vector dimensions are d_k, d_k and d_v. The shape of each matrix was placed next to each node. The corresponding equation number was added inside the node. 25

8 Visualization of the multihead-attention mechanism. Assuming that the input sequence is of length n. The key, query, and value vector dimensions are of d_{model}. The shape of each matrix was placed next to each node. The following usually applies $hd_v = d_{model}$. Each q_i, k_i, v_i is the result of the matrix multiplication with the corresponding weight matrix with Q, K, and V and input to the corresponding $head_i$. 27

9 Attention matrix for a unidirectional language model with five tokens. . . 31

10 Bidirectional attention matrix with randomly masked tokens m. The masked tokens should be predicted by the model by taking all the non-mask tokens into account. This is also why this approach is thought of bidirectional, since all the tokens before and after the masked token are considered. 31
11 Visualization of BERT's input representation, which combines token embeddings, segmentation embeddings, and position embeddings.. (Devlin et al. (2018). 34
12 BERT's training involves two main stages: pretraining and fine-tuning. The core model architecture remains consistent across both stages, with the exception of the output layers. The model parameters obtained from pretraining are used as starting points for various downstream tasks. In the fine-tuning stage, all of these parameters are adjusted to better suit the specific task at hand. Special tokens, such as [CLS] at the beginning of inputs and [SEP] to separate segments like questions and answers, are used in the process (Devlin et al. (2018)). 36
13 Sliding window attention. Each tokenx_i only attends to its local neighborhood. In this case $w = 6$ (Beltagy, Peters, and Cohan (2020)). 42
14 Dilated sliding window attention with $w = 6$ and $d = 2$. There is a gap of $d - 1 = 1$ between each token in the local neighborhood (Beltagy, Peters, and Cohan (2020)). 43
15 Global attention combined with sliding-window attention for $w = 6$, $L = \{0, 1, 5, 15\}$ and $d = 1$. The first row is the corresponding [CLS]-token which uses global attention to capture the information of the whole input sequence (Beltagy, Peters, and Cohan (2020)) 45
16 LSH attention algorithm: The sequence is hashed into buckets, then sorted according to their bucket number and position in the bucket. Following this the sequence is chunked into parts of length m. As shown queries outside a chunk can only attend backward. 46
17 The forward pass of a reversible block (Gomez et al. (2017)). Based on equation 31. 49
18 The recomputation of the input activations a reversible block (Gomez et al. (2017)). Based on equation 32. 49
19 The computational graph of a reversible block. 50
20 Distinguishing between generative and discrimnative models. As we can see discriminative models from a decision boundary and learn the mapping between input and labels directly, while generative models learn the distribution on which a classification can be done afterwards (Google Developers (2022)). 54
21 Replaced token detection in Electra. Masked tokens are replaced with generator predictions. The discriminator learns to recognize generated tokens. After pretraing only the discriminator is used (Clark et al. (2020)). 55

22 Overall loss, train discriminator and train generator loss 63
23 Overall loss, train discriminator and train generator loss 64
24 Overall loss, train discriminator and train generator loss 65

25 Financial sentiment finetuning results 67
26 Financial topic finetuning results 69

Nomenclature

$\mathbf{W}$	Bold capitalized letters. Matrices, Tensors or Layer of a neural network. They are bold when they consist of learnable parameters.
A	Cap letters. Input or Output of functions, transformations, projections or layers. They do not consist of learnable parameters.

1 Introduction

The advent of deep learning has precipitated unparalleled advances across various domains of artificial intelligence, notably in natural language processing (NLP). This surge in innovation has been significantly propelled by the development of transformer models, which have set new benchmarks for a plethora of language-related tasks. However, as these models grow in complexity and size, they increasingly demand substantial computational resources, raising critical challenges for their application within environments constrained by hardware limitations. This thesis, titled "Assessing Efficiency in Domain-Specific Transformer Models: Comparing, Pretraining, and Finetuning Small-Scale Transformer Models within Hardware Limitations for Financial NLP", seeks to navigate these challenges, focusing on NLP in the financial sector.

Financial NLP, a subfield that applies NLP techniques to interpret and analyze financial texts, demands high accuracy and rapid processing speeds due to the time-sensitive and complex nature of financial data. The precision of models in this domain can significantly impact decision-making processes in finance, underscoring the need for efficient and effective model architectures. This work embarks on a comparative study of various transformer model architectures, including the Reformer, Plain BERT, and a novel reversible dilated BERT, examining their pretraining and finetuning efficacy within the constraints of limited computational resources.

At the heart of this investigation is the exploration of domain-specific pretraining on financial texts, an approach that tailors the model's learning to the intricacies and nuances of financial language. This specificity is crucial for enhancing model performance in domain-specific tasks, where general-purpose models may not capture the unique characteristics of financial discourse. The thesis meticulously evaluates the computational efficiency and model performance trade-offs, aiming to identify strategies that maximize accuracy without compromising on resource demands.

Additionally, the research delves into the Electra pretraining method, applied to the BERT model, to examine its potential for optimizing finetuning processes in a resource-constrained setting. By focusing on the discriminator component of the Electra model, this study offers insights into leveraging pretraining techniques that require fewer computational resources while maintaining, or even enhancing, model performance.

Through a detailed analysis of foundational concepts such as tokenization, embeddings, the challenges of recurrent neural networks (RNNs), and the transformative impact of attention mechanisms, this thesis sets the stage for a comprehensive exploration of transformer models in financial NLP. It aims to contribute to the ongoing discourse on optimizing NLP models for specialized domains, addressing the critical balance between computational efficiency and model effectiveness.

This introduction chapter outlines the motivation, objectives, and structure of the thesis, setting the foundation for a deep dive into the complexities and innovations at the intersection of transformer models, financial NLP, and hardware constraints. It frames the research within the broader context of AI advancements, highlighting the significance of efficient model design and domain-specific adaptation in pushing the boundaries of what is possible in NLP.

2 Foundation

2.1 Tokenization in NLP

NLP is fundamentally concerned with enabling computers to understand and process human languages. To achieve this, one of the initial steps involves converting the raw text into a format that can be easily interpreted by algorithms. This process is known as tokenization. Tokenization is the task of chopping up text into pieces, called tokens, which are semantically useful units such as words, phrases, or symbols (Mielke et al. (2021)). The choice of tokens is a critical step in NLP as it directly affects the representation of input data and the subsequent layers of processing.
In the early stages of text processing, tokenization was often as simple as splitting text by whitespace and punctuation marks. However, such a naive approach can be insufficient for understanding the complexities of language, where the meaning can be significantly altered by morphology and syntax. For instance, ”dogs” should be related to ”dog”, and ”talked” to ”talk”, but simple whitespace-based tokenization would treat each as a distinct token without any relationship to one another. Another issue was how to deal with punctuation. Looking at the word ’don‘t’, we can split it in three distinct parts (’do’, ’‘’, ’t’) or not split it at all. It was argued that the best way to split this word was into two pieces ’do’ and ’n‘t’ (Mielke et al. (2021)). This led to the upcoming of various tokenization techniques, each using its own approach to split words into character-sequences. Most methods can be split into three groups:

- Whitespace-delimited pre-tokens
- Learned subwords
- Byte based encoding

The granularity of the tokens increases with each new technique, meaning the byte-based encodings have the highest granularity.[1] Each approach is better suited for different tasks. Some tasks need very granular tokens while others emphasize the interpretability and efficiency of larger tokens (Mielke et al. (2021)). Each token is assigned a unique ID such that same tokens receive the same ID. Once tokenization is complete, the next challenge is to represent these tokens numerically so that they can be processed by algorithms. Important

[1]The higher the granularity the fewer number of characters per token.

2.2 Word Embeddings

One common approach to represent the tokens is to use one-hot-encoding vectors which are of size G where $G =$ number of tokens. The vector consists of all zeros but a one at the index $i \in 1, 2, \ldots G$ of the current token. This method, while straightforward, results in extremely high-dimensional and sparse vectors that are inefficient in terms of both storage and computation. More critically, one-hot encoded vectors fail to capture any semantic information; every word is equidistant from every other word in this space, making it impossible for models to learn the semantic relationships and similarities between words. This semantic gap is where word embeddings come into play, offering a dense and semantically rich alternative.

Word embeddings are a class of techniques where individual words are represented as fixed-length vectors in a predefined vector space. Each word is mapped to one vector. The embeddings capture the semantic properties of words, such that words that are semantically similar are located in close proximity to one another in the embedding space. An important feature of these embeddings is that the 'raw word vectors are first projected onto a so-called embedding layer before being fed into other layers of the network' (Almeida and Xexéo (2019a)). There are multiple ways to obtain these representations, famous ones are continuous bag-of-words, skip-gram, or Word2Vec (Almeida and Xexéo (2019b)).When building a new model, embedding layers and word representations are either learned from scratch by training the whole model or pretrained embeddings are used.

2.3 Recurrent Neural Networks and its problems

2.3.1 Introduction to Recurrent Neural Networks

Neural networks, since their inception, have been instrumental in advancing the field of machine learning. However, traditional feedforward neural networks have a significant limitation: they assume that all inputs and outputs are independent of each other. In many real-world scenarios, this is not the case. This can be especially stated for language. Recurrent neural networks (RNNs) are neural networks for modeling dependent data. The network structure is similar to a classic feedforward neural network, but we allow connections between hidden units associated with a time delay. Using these, the model can receive information about the past and learn temporal relations between inputs that are possibly far away from each other in the data.(Pascanu, Mikolov, and Bengio (2012a)) For a given input x_t (which is usually not the word itself but rather an embedding), the hidden state at time t denoted as h_t, is updated using the previous hidden state h_{t-1}, the current input x_t, and a bias term $\mathbf{b}$. The matrices $\mathbf{W_h}$ and $\mathbf{W_x}$ represent the weights applied to the hidden state and the input, respectively, and σ denotes the activation function, typically a *sigmoid* or *tanh* function:

$$h_t = \mathbf{W}_h \sigma(h_{t-1}) + \mathbf{W}_x x_t + \mathbf{b} \tag{1}$$

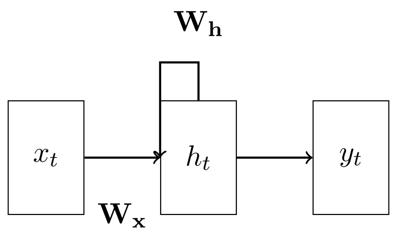


Figure 1: Schematic visualization of a RNN cell.

Important to note is that the input at $t = 0$, i.e. x_0, is provided by the user, set to zero, or learned. The parameters are shared across time (Pascanu, Mikolov, and Bengio (2012a)).

To understand the usage of the hidden states, an unrolled version of the recurrent neural network for a given timestep t, its predecessor $t - 1$, and its successor $t + 1$ is shown in Figure 2.

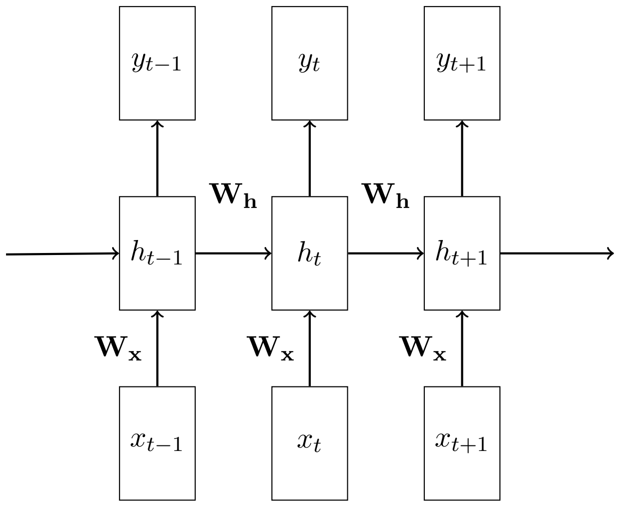


Figure 2: Unrolled RNN for three time steps by creating a copy for each time step (Pascanu, Mikolov, and Bengio (2012b)). The hidden output of time step $t-1$ is input for the hidden state at t and so on. This is how the temporal information gets passed through the network.

RNNs use these recurrent blocks and stack multiples of them combined with feedforward layers to create the final model. The model can be used in sequence-to-sequence tasks by using the hidden state outputs at each time-step t, or sequence-to-class, by using only the final output of the model. Usually, the input data is encoded into word-embeddings before passing it to the RNN.

2.3.2 Training of RNNs

To train the RNN and compute the gradients, a technique known as Backpropagation through time (BPTT) is used. The unrolled visualization of the RNN, as we saw in Figure 2 is solely done for simplification. However, it provides valuable insights into how BPTT works. In BBTT the recurrent model is represented as a multi-layer feedforward model by unrolling it as shown in Figure 2. Backpropagation is then applied on the unrolled model (Pascanu, Mikolov, and Bengio (2012b)).

2.3.3 Vanishing and Exploding Gradient Problem in Recurrent Neural Networks

RNNs are recognized as potent tools for processing sequential data, owing to their intrinsic ability to remember past inputs. However, like all models, they have their challenges. A notable difficulty they encounter is learning long-term dependencies. Specifically, if pertinent information within a sequence is spaced far apart, an RNN may find it challenging to link and capitalize on this information. This issue can be traced back to the vanishing gradient problem.

Deep neural networks often grapple with the vanishing gradient problem, where the gradients associated with long-range relations diminish exponentially fast, approaching zero in norm. The exploding gradient problem describes the opposite behavior. It occurs when the gradient norm of long-term components increases exponentially more than the short-term ones (Pascanu, Mikolov, and Bengio (2012b)). These challenges intensify in the context of RNNs. The reason is that gradients in RNNs are influenced not only by the "depth" of the network but also by the sequence's "time" dimension (Vennerød, Kjærran, and Bugge (2021)). Technically speaking, the hidden output z_i of the model at layer i is a function of all the outputs from previous layers z_j for $j \in \{i-1,\ i-2,\ \ldots,\ 0\}$ where i is the model's layer index, establishing the dependency. Furthermore, the RNN also depends on prior hidden states h_t for $t \in \{t-1,\ t-2,\ \ldots,\ 0\}$, with t signifying the time-step of the recurrent unit. This interdependence is more complex than that observed in standard neural networks. Consequently, the early layers or time steps receive negligible gradient updates, rendering the network unable to learn and adjust its weights effectively for distant past inputs.

To overcome the challenges of capturing long-term dependencies in RNNs, researchers developed several innovative solutions that have since gained widespread acceptance in the field. Notably, the Long-Short-Term-Memory (LSTM) and the Gated-Recurrent-Unit (GRU) stand out, both categorized under the broader umbrella of Gated Neural Networks (GNNs). At the heart of GNNs lies the principle of using gating mechanisms to control the information flow across layers or sequences (Hu et al. (2018)). These gates enable the network to more effectively identify, retain, and utilize pertinent information over extended sequences. It's worth noting that both GRU and LSTM architectures maintain compatibility with the input-output conventions of traditional RNNs. This compatibility allows for seamless substitution of RNN blocks within a given model without necessitating structural modifications. However, this work focuses on attention mechanisms. In the ensuing section, we will explore how RNNs, when augmented with attention, present a promising avenue to address the longstanding issues of vanishing and exploding gradients.

2.4 Attention in Recurrent Neural Networks

2.4.1 Encoder-Decoder RNNs

RNNs are well-suited for sequence-to-sequence tasks, with machine translation being a prime example of their application. Traditional RNNs, including their variants like GRUs and LSTMs, have been pivotal in these domains. Despite their capabilities, these networks often struggled with complex sequences, leading to suboptimal performance in certain contexts.

To address these limitations, the Encoder-Decoder RNN architecture was introduced. This innovative approach involves two coupled RNNs: the encoder and the decoder. The encoder processes the input sequence and compresses the information into a context vector, a fixed-length representation capturing the essence of the input data. The decoder then takes this vector and reconstructs a variable-length target sequence from it. The underlying strength of this model lies in its ability to maintain and manipulate the temporal dependencies within the data. The original authors of the Encoder-Decoder RNN framework succinctly describe the process: "[t]he encoder maps a variable-length source sequence to a fixed-length vector, and the decoder maps the vector representation back to a variable-length target sequence. The two networks are trained jointly to maximize the conditional probability of the target sequence given a source sequence"(Cho et al. (2014a)). So the goal of this architecture is to create a hidden representation c of the input data, which summarizes the information and its temporal dependency. The decoder accordingly, is trained to generate the output sequence by predicting the next symbol y_t given h_t^{dec}. Note that h_t^{dec} also depends on the previous hidden state h_{t-1}^{dec} , the previous output y_t and the summarized representation of the input c (Cho et al. (2014b)). Technically speaking, h_t^{dec} and y_t can be described by the functions f and g as follows:

$$h_t^{\text{dec}} = f(h_{t-1}^{\text{dec}}, y_{t_1}, c) \tag{2}$$

$$y_t = g(h_t^{\text{dec}}, y_{t_1}, c) \tag{3}$$

For a more intuitive grasp of this architecture, refer to Figure 3, which visualizes the Encoder-Decoder RNN framework.

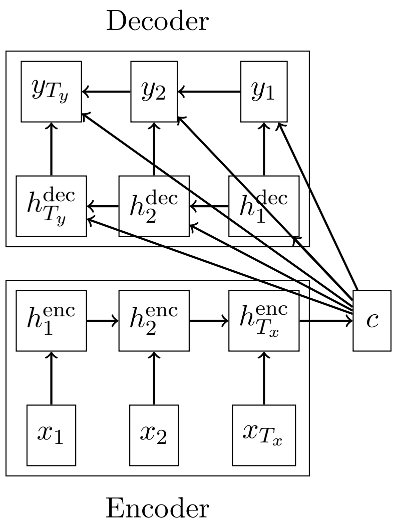


Figure 3: Visualization of an Encoder-Decoder RNN. The representation vector c gets passed through every hidden state h_t of the Decoder. Note that only three hidden states were used here for simplification, but in real scenarios these can be extended to much larger sequences.

The model is trained using a any gradient based algorithm since all parameters are differentiable.

2.4.2 Attention based Encoder-Decoder RNNs

The Encoder-Decoder RNN framework, while powerful, has its limitations, particularly when dealing with long input sequences. The model's performance tends to degrade because the fixed-length context vector c becomes an information bottleneck. As sequences grow longer, it becomes increasingly challenging for the encoder to compress all the necessary information into a single vector (Bahdanau, Cho, and Bengio (2016a)). It was demonstrated that the efficacy of a standard encoder-decoder model declines sharply with the extension of the input sentence length (Bahdanau, Cho, and Bengio (2016a)).

To address this limitation, Bahdanau et al. introduced an innovative architecture that redefines the context vector's role. Rather than relying on a single, static representation vector c, their model calculates a distinct context vector c_t for each decoding step. This approach allows each hidden state h_t^{dec} in the decoder to be dynamically influenced by a function f of the preceding step h^{dec}_{t-1}, the evolving context vector c_t and previous predicted output y_{t-1} (Bahdanau, Cho, and Bengio (2016b)). Consequently the updated functions for the output y_t and hidden stated h_t are expressed as:

$$h^{\text{dec}}_t = f(h^{\text{dec}}_{t-1}, y_{t_1}, c_t) \tag{4}$$

$$y_t = g(h_t^{dec}, y_{t_1}, c_t) \tag{5}$$

The introduction of the variable context vector c_t marks a departure from the earlier equations (2) and (3), where a single, unchanging context vector was used. The context vector c_t is computed as:

$$c_t = \sum_{i=1}^{T_x} \alpha_{ti} h_i^{enc} \tag{6}$$

and

$$\alpha_{ti} = \frac{\exp e_{ti}}{\sum_{k=1}^{T_x} \exp e_{tk}} \tag{7}$$

where

$$e_{ti} = \mathbf{A}^{RNN}(h_i^{enc}, h_{t-1}^{dec}\ ) \tag{8}$$

Looking at equation 6 one can see that the context vector c_t is a weighted sum of the encoder hidden states h_i^{enc}. The score e_{ti} is computed by the alignment model $\mathbf{A}^{RNN}$, which evaluates how well each encoder hidden state h_i^{enc}, at each time-step i in the input sequence, aligns with the previous decoder hidden state h_{t-1}^{dec} . The authors implement a as a fully connected feedforward layer. The alignment score helps to determine the focus of the decoder on different parts of the input sequence when generating the $t-th$ word of the output sequence. The attention weights α_{ti} are then derived by applying a softmax function to the scores e_{ti} , which creates a normalized probability distribution representing the relevance of each input sequence word to the current output (Bahdanau, Cho, and Bengio (2016b)).

To sum up, the proposed approach computes a dynamic context vector at each time-step, representing a weighted sum of all encoder hidden states. This enables the model to adaptively learn which input data points to attend to for predicting each element of the output sequence. Such an approach not only addresses the challenge of long-term dependencies—which are particularly problematic for traditional RNNs—but also mitigates the issues of vanishing and exploding gradients to a certain extent by providing shortcut connections through the attention mechanism. Moreover, it moves beyond the limitations of a single fixed representation of the input sequence, introducing a flexible context that varies according to the position in the output sequence. This innovation allows for more nuanced and effective handling of sequences, particularly as they increase in length. From now on this mechanism will simply be termed 'attention.' The intricacies of this process are illustrated in Figure 4. Furthermore, attention enables the interpretation of the relationship between input and predicted data. By examining which inputs the model focuses on—or the attention scores α_{ti} in equation 7, i.e. the weights to determine c_t—one can glean insights into language structures and the model's learning patterns. For an example refer to Figure 5.

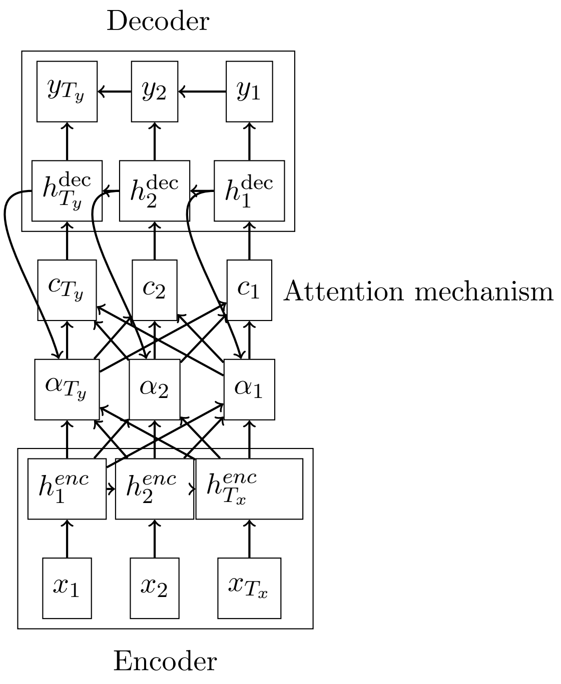


Figure 4: Simplified visualization of an attention backed Encoder-Decoder RNN. One can see that c_i is influenced by all encoder hidden states over α_{ti}. Input length T_x and output length T_y may differ. To maintain a clear overview, the connections from the encoder hidden states to each context vector are left out. In this step the weighted sum of the encoder hidden states would be computed.

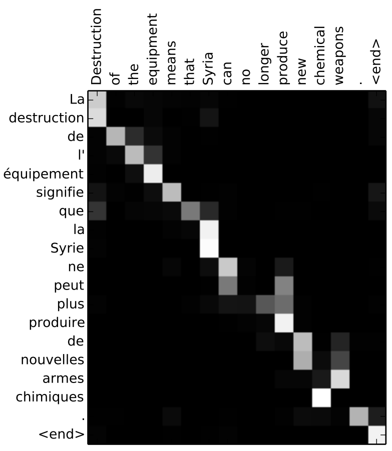


Figure 5: Attention scores from english-french translation. Each output word (french) attends to all the input words (english) to a specific degree. The brighter the cell α_{ti}, the higher the attention score between word t and i. (Vaswani et al. (2017a)

2.5 Attention is all you need

2.5.1 Introduction

In the preceding chapter, we explored the emergence of attention mechanisms as a powerful tool to augment RNNs. By enabling models to dynamically focus on different parts of the input sequence for each output, attention-backed RNNs, such as those using LSTM or GRU architectures, significantly improved the ability of neural networks to capture long-range dependencies and contextual nuances in sequence data.

Building upon the limitations of RNNs, which are constrained by their sequential computation thus hindering parallelization, the next evolutionary step comes in the form of the transformer model. Introduced by Vaswani et al. in the seminal paper ”Attention is All You Need”, this model signifies a departure from traditional sequence processing methods entirely. They are mentioning the rising role of attention mechanisms in the field of sequence-to-sequence modelling, but are emphasizing that these are almost always used in conjunction with recurrent networks. Consequently, they are proposing a model solely based on attention mechanisms to draw global dependencies between input and output (Vaswani et al. (2017b)). By relying only on attention mechanisms — without any recurrence — transformers marked the beginning of a new era in sequence modeling.

2.5.2 Model architecture

The transformer follows the structure of Encoder-Decoder RNNs. The encoder consists of six identical layers N stacked on top of each other. Each layer is divided into two distinct parts: the first part is a multi-head self-attention mechanism, and the second part is a straightforward feed-forward neural network that connects positions. To improve the flow of information through these layers, each one incorporates a residual connection [2], which is then normalized by a layer normalization process. [3]. This means the output from each part of the layer is the normalized sum of the input and the processed input. Additionally, to support these residual connections, every sub-layer, including the embedding layers, is designed to output data in a 512-dimensional space. (Vaswani et al. (2017a)).

The decoder in the transformer architecture mirrors the encoder's structure with six identical layers. Each decoder layer includes three sub-layers: two are similar to the encoder layers, and a third sub-layer is added for multi-head attention over the encoder's

[2]Residual connections add the input of the layer to the output to ease information flow and prevent vanishing gradients. Let x be the input and f the output mapping. Then a residual connection can be described as $z = f(x) + x$. For further information see arxiv.org/abs/1512.03385

[3]Layer normalization computes normalization statistics for each individual layer across features, unlike batch normalization which computes across the batch dimension. For further information see arxiv.org/abs/1607.06450

output. Residual connections, followed by layer normalization, are also a part of the decoder design. The decoder uses the encoder output as input. That is how the input information gets passed to the encoder. The decoder generates one output at a time. It utilizes the prior outputs and the encoder output. To ensure that the decoder respects the correct order of sequence generation, a masking technique is applied in the self-attention sub-layer. This technique ensures that when determining the output at a certain position, the model can only consider the previous positions, not the subsequent ones. This is further enforced by shifting the output embeddings by one position, safeguarding the model's autoregressive property, where each step's prediction is contingent upon the preceding steps only(Vaswani et al. (2017a)). Similar to Encoder-Decoder RNNs the encoder creates a representation of the input data by encoding all the information and global dependencies. Figure 5 displays the original architecture used in the paper.

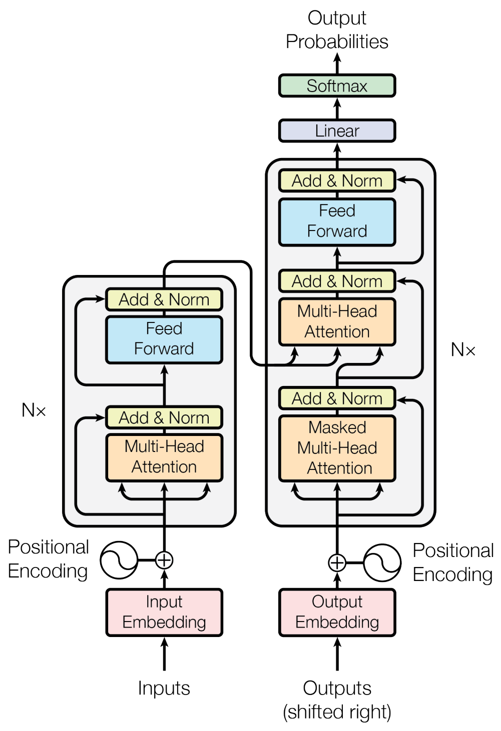


Figure 6: The transformer architecture used in the original paper (Vaswani et al. (2017a)

2.5.3 Positional encoding

The transformer has no spatial information about the input data as in RNNs (in which the input data is passed sequentially). Consequently, the authors suggest adding positional encoding to the input embeddings. There exist several techniques such as absolute encodings, relative encodings, or LSTM-based encodings (Rosendahl et al. (2019)). For the transformer model, absolute encodings based on sine and cosine functions of difference frequencies are used. The encodings have the same dimension as the input embedding layer, such that the two can be added. For a given position in the input sequence *pos* and dimension index *i* the encodings are computed as:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{embedding}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{embedding}})$$

For even dimension indices the *sin* is used and for uneven dimension the *cos* is used. The term $10000^{2i/d_{embedding}}$ controls the frequency of both functions. This ensures that each dimension of the positional encoding corresponds to a sinusoid with a different wavelength, ranging from very short to very long lengths. The authors hypothesized that these encodings would allow the model to learn the relative positions of the input data which is highly important for computing attention scores (Vaswani et al. (2017c)), which turned out to be true. In Figure 6 the summation of input and output embeddings and positional encodings can be seen inside the model architecture of the transformer before the attention block.

2.5.4 Self-attention in transformers

As highlighted in the preceding discussions, the key innovation of the transformer was to rely on attention mechanisms only. Hence, they proposed a new way of computing attention scores. They describe their process as mapping a key and a set of key-value pairs to an output. The keys, queries, values, and outputs are all vectors, which are concatenated n times to a matrix, where n is the length of the input sequence. They are typically derived by applying a feedforward layer to the input embeddings with an individual weight matrix for each vector. The output is computed as a weighted sum of the values. The weights are computed by a similarity function, like the dot-product, of the query and the corresponding key (Vaswani et al. (2017d)). The authors name their attention mechanisms 'Scaled Dot-Product Attention'(Vaswani et al. (2017d)). The keys K and queries Q are of dimension d_k and values V are of dimension d_v. Beginning with a dot product of queries and keys and dividing the result by $\sqrt{d_k}$, applying a softmax (equation 9) to compute the attention scores $\mathbf{S}$ and multiplying with the value vectors (equation 10) to receive the final weighted matrix $\mathbf{A}$, the process can be formally written as:

$$\mathbf{S}(Q,K) = softmax(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}}) \tag{9}$$

$$\mathbf{A}(Q,K,V) = \mathbf{S}(Q,K)V \tag{10}$$

(Vaswani et al. (2017d)). Refer to Figure 7 for a visualization of the process. It is worth highlighting that this attention computation, computes the attention of input data to itself. Precisely, if the input sequence has length n the attention score matrix $\mathbf{S}$ is of shape $n \times n$. By looking at the attention score matrix of the so-called self-attention mechanism, one can derive how each input data point attends to all the other input data points. There does also exist "encoder-decoder-attention" when queries come from the previous decoder layer and keys and values come from the output of the encoder(Vaswani et al. (2017d)). However, the focus of this work lies on encoder models, for which self-attention is the only attention used. The encoder-decoder attention is well suited for sequence-to-sequence tasks, where the decoder computes attention scores for the output data and relates it to the encoder attention output.

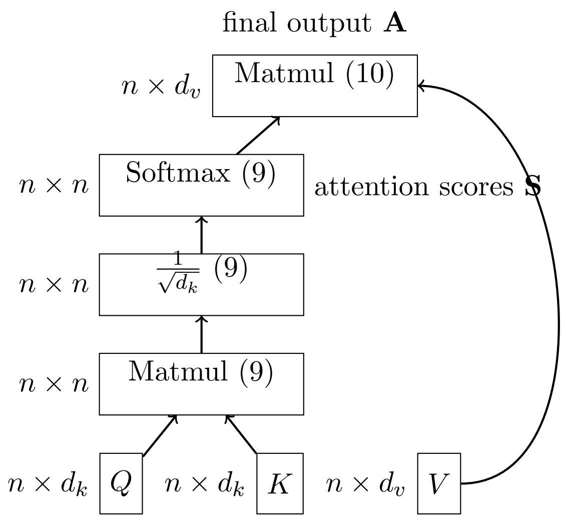


Figure 7: Visualization of the attention mechanism. Assuming that the input sequence is of length n. The key, query, and value vector dimensions are d_k, d_k and d_v. The shape of each matrix was placed next to each node. The corresponding equation number was added inside the node.

In the case of stacked attention blocks as in the transformer architecture presented the input does not come from the input or output embeddings but rather from the previous attention block output.

2.5.5 Multi Head Attention

The presented self-attention computes one attention score matrix $\mathbf{S}$ for d_{model} - dimensional keys, queries and values (in this case $d_v = d_k$, and we term it d_{model}).

The authors however also proposed a technique called 'multi-head attention", in which keys, queries, and values are projected h-times with different, learned linear projections to d_k, d_k and d_v. The attention is then computed on each of these projections allowing for parallel computation for each d_v-dimensional attention-'head'. The heads are concatenated and projected again, leading to the final value.

The key objective is that the model can jointly attend to information from different representation subspaces (by linearly projecting the 'whole' key, queries, and values to lower dimensional spaces). The authors of the paper suggest $h = 8$ different attention heads, for which they use $d_k = d_v = d_{model}/h = 64$ Vaswani et al. (2017d).

This derivation is based on their chosen model dimension d_{model} of 512. For given query $Q \in \mathbb{R}^{\mathrm{n \times d_{model}}}$, key $K \in \mathbb{R}^{\mathrm{n \times d_{model}}}$, value $V \in \mathbb{R}^{\mathrm{n \times d_{model}}}$ and weight matrices $\mathbf{W_i}^{\mathbf{Q}} \in \mathbb{R}^{\mathrm{d_{model} \times d_k}}$, $\mathbf{W_i}^{\mathbf{K}} \in \mathbb{R}^{\mathrm{d_{model} \times d_k}}$ and $\mathbf{W_i}^{\mathbf{V}} \in \mathbb{R}^{\mathrm{d_{model} \times d_v}}$ and $\mathbf{W}^{\mathbf{O}} \in \mathbb{R}^{\mathrm{hd_v \times d_{model}}}$, the process can be described formally as:

$$head_i = \mathbf{H}(Q, K, V, i) = \mathbf{A}(Q\mathbf{W_i}^{\mathrm{Q}}, K\mathbf{W_i}^{\mathrm{K}}, V\mathbf{W_i}^{\mathrm{V}}) \tag{11}$$

$$\mathbf{B}(Q, K, V) = Concat_{col}(\mathbf{H}(Q, K, V, 1), \mathbf{H}(Q, K, V, 2), \ldots, \mathbf{H}(Q, K, V, h)) \tag{12}$$

and thus

$$\mathbf{B}(Q, K, V) = Concat_{col}(head_1, head_2, \ldots, head_h)\mathbf{W}^{\mathbf{O}}$$

where $Concat_{col}$ is defined as the column-wise concatenation of k matrices $M \in \mathbb{R}^{\mathrm{a \times b}}$

$$Concat_{col} : \underbrace{\mathbb{R}^{\mathrm{a \times b}} \times \mathbb{R}^{\mathrm{a \times b}} \times \ldots \mathbb{R}^{\mathrm{a \times b}}}_{k\text{-times}} \to \mathbb{R}^{\mathrm{a \times kb}}$$

and as it will be used later $Concat_{row}$ is the row-wise concatenation of k-matrices $M \in \mathbb{R}^{\mathrm{a \times b}}$

$$Concat_{row} : \underbrace{\mathbb{R}^{\mathrm{a \times b}} \times \mathbb{R}^{\mathrm{a \times b}} \times \ldots \mathbb{R}^{\mathrm{a \times b}}}_{k\text{-times}} \to \mathbb{R}^{\mathrm{ka \times b}}$$

However, it is worth mentioning that the concatenation of the heads might be executed slightly different in practice. Since modern GPUs are optimized for tensor operations, especially tensor multiplication, $\mathbf{B}$ can be computed as a three-dimensional tensor product (neglecting batch dimension) instead of explicitly concatenating each head. So the adjusted Q, K and V are of shape $n \times h \times \frac{d_{model}}{h}$ instead of shape $n \times d_{model}$. As we can see, the model dimension is split into two dimension, leading to the three dimensional tensor.

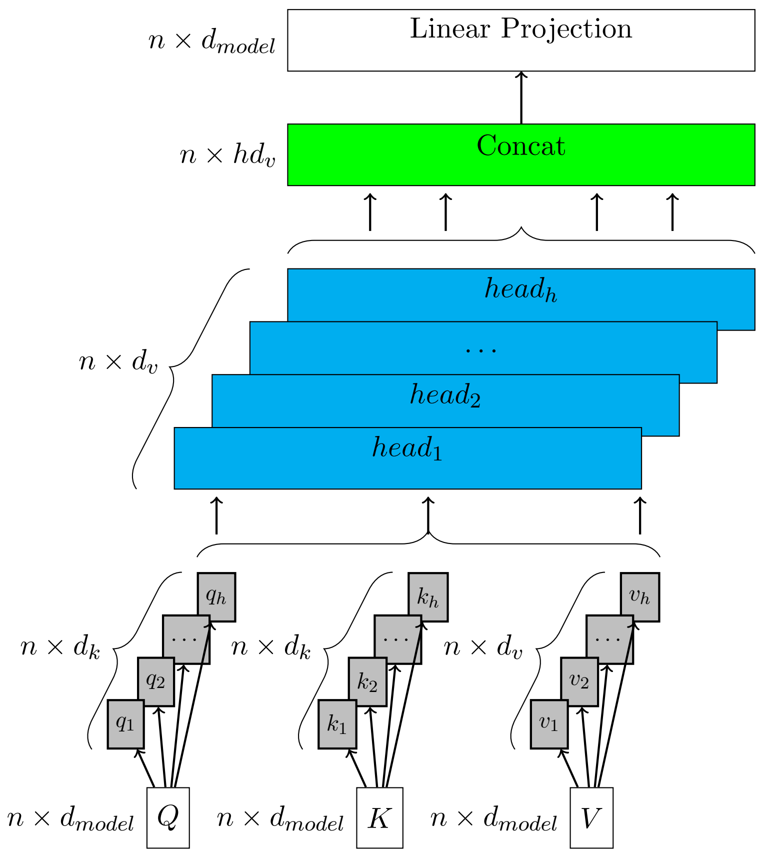


Figure 8: Visualization of the multihead-attention mechanism. Assuming that the input sequence is of length n. The key, query, and value vector dimensions are of d_{model}. The shape of each matrix was placed next to each node. The following usually applies $hd_v = d_{model}$. Each q_i, k_i, v_i is the result of the matrix multiplication with the corresponding weight matrix with Q, K, and V and input to the corresponding $head_i$.

One of the key features of multihead-attention next to allowing the model to attend to different information representations is the ability to fully parallelize the multihead-attention computation over all heads. This is inherently different from modeling in RNN-like models, where is no possibility for parallelizing because the data is processed sequentially. By making use of this feature, very complex models can be trained much faster and more efficiently than any other sequential model. Refer to Figure 8 for a visualization of the process.

2.5.6 Attention Block

the transformer architecture is based on several attention blocks which all use scaled-dotproduct self-attention to understand sequential data. Even though already touched on in the introductory section, the following section will elaborate on the inner workings of the encoder attention block and how the attention mechanism is used in conjunction with already existing structures to form the final block. Please note that for the following description the process is described for a batch size $b = 1$. Hence, all the shapes are squeezed ignoring the batch dimension.

The block gets fed an input embedding $E \in \mathbb{R}^{\mathrm{n} \times \mathrm{d}_{\mathrm{model}}}$, where n is the sequence length. The embedding is linearly projected onto three subspaces to result in the well-known query $Q \in \mathbb{R}^{\mathrm{n} \times \mathrm{d}_{\mathrm{k}}}$, key $K \in \mathbb{R}^{\mathrm{n} \times \mathrm{d}_{\mathrm{k}}}$, and value $V \in \mathbb{R}^{\mathrm{n} \times \mathrm{d}_{\mathrm{v}}}$ matrices by multiplying E with the corresponding projection matrices $\mathbf{W}_Q^{\mathrm{proj}} \in \mathbb{R}^{\mathrm{d}_{\mathrm{model}} \times \mathrm{d}_{\mathrm{k}}}$, $\mathbf{W}_K^{\mathrm{proj}} \in \mathbb{R}^{\mathrm{d}_{\mathrm{model}} \times \mathrm{d}_{\mathrm{k}}}$ and $\mathbf{W}_V^{\mathrm{proj}} \in \mathbb{R}^{\mathrm{d}_{\mathrm{model}} \times \mathrm{d}_{\mathrm{v}}}$. Q, K, and V are used as input for the multihead-attention layer with h heads (Equation 13). A residual connection is applied around the multihead-attention output $A_{multihead} \in \mathbb{R}^{\mathrm{n} \times \mathrm{d}_{\mathrm{model}}}$ followed by a layer normalization $\mathbf{L}$ (Equation 14). The resulting normalized $A_{multihead}^{\mathrm{norm}}$ is passed to a feedforward layer $\mathbf{F}$ to result in a matrix $Z \in \mathbb{R}^{\mathrm{n} \times \mathrm{d}_{\mathrm{model}}}$ (Equation 15). Again, a residual connection is used followed by a layer normalization. This leads to the final output $Y \in \mathbb{R}^{\mathrm{n} \times \mathrm{d}_{\mathrm{model}}}$ of the attention block (Equation 16). The process can be depicted formally as:

$$A_{multihead} = \mathbf{B}(\mathbf{W}_Q^{\mathrm{proj}} E, \mathbf{W}_K^{\mathrm{proj}} E, \mathbf{W}_V^{\mathrm{proj}} E) \tag{13}$$

$$A_{multihead}^{\mathrm{norm}} = \mathbf{L}(A_{multihead} + E) \tag{14}$$

$$Z = \mathbf{F}(A_{multihead}^{\mathrm{norm}}) \tag{15}$$

$$Y = \mathbf{L}(Z + A_{multihead}^{\mathrm{norm}}) \tag{16}$$

Refer to Figure 6 to get a visual understanding of the encoder attention block (the encoder block is located in the left subnetwork of the transformer in the figure). The authors of the original paper stack six of these attention blocks (Vaswani et al. (2017d)), where the output of one block is the input for the next block, resulting in an encoded representation of the input sequence. This can then be either used for the decoder of the transformer or for several tasks, e.g. classification.

2.5.7 Comparison to attention in RNNs

This mechanism works similarly to the attention found in Encoder-Decoder RNNs. In both methods attention is computed and used as weights for the following weighted sum of input information. Even though attention is calculated in different ways, the core objective of attention in both models is to enable the model to focus on important parts of encoded information. While the transformer uses encoder-self-attention and also encoder-decoder-attention where output words attend to input words, attention in RNNs is only encoder-decoder-attention. Consequently, the transformer is leveraging its ability to learn relations from input words to themselves enabling a profound understanding of language.

2.5.8 Revolutionary impact

The advent of the Transformer architecture has not only revolutionized the field of NLP but has also had a profound impact on the broader domain of AI. By decoupling the processing of sequence data from sequential computation, the Transformer has enabled models to learn and predict with unprecedented efficiency and accuracy. This shift has facilitated the development of models that can process language in a way that is more akin to human understanding, capturing nuances and relationships that were previously out of reach for machine learning algorithms.

In NLP, the Transformer's influence is evident in the surge of state-of-the-art models that have followed its design principles. These models have set new benchmarks in a variety of tasks, from machine translation to question answering, fundamentally changing the landscape of what machines can achieve with language. Beyond NLP, the principles of the Transformer are being adapted to tackle challenges in image recognition(Dosovitskiy et al. (2020)), audio processing(Verma and Berger (2021)), and even in areas such as genomics(Anwar Khan and Lee (2021)), where the ability to model complex, long-range dependencies is crucial.

The Transformer's self-attention mechanism has proven to be a versatile and powerful tool, enabling models to focus on different parts of the input data as needed, without the constraints imposed by the sequential nature of previous architectures. This has opened up new possibilities for parallel computation, allowing for more efficient training and inference, and paving the way for the development of even larger and more complex models.

The next sections will delve into the various attention based models that have emerged. These models build upon the Transformer's foundation, each bringing its own innovations. However, we start by introducing the concept of Language Models and transfer learning, which is s key concept in how these models differ.

2.6 Language models

Language modeling involves training a model to understand the likelihood of sequences of tokens, which are selected from a predetermined list of vocabulary. Given a sequence of tokens [4] (x1, x2, ..., xn), the goal is to establish a probability distribution expressed as P(x1, x2, ..., xn). Typically, the joint probability distribution is calculated by applying the chain rule:

$$P(x) = \prod_t P(x_t|x_{<t})$$

[4]Words are usually encoded in some kind of token. Both will be often used interchangeably.

In this formula, denotes all the tokens that precede the token currently being considered. In easier terms, the Language Model (LM) is trained to predict a token at position t given its preceding tokens at positions $t-1, t-2, ...1$ .While this method is widely adopted, the field is rich with a variety of other techniques documented in scholarly research (Alyafeai, AlShaibani, and Ahmad (2020)).

1. Unidirectional LMs
   Unidirectional LMs take into account only the tokens that precede or follow the current token within a context. For example, if we have the sequence (x1, x2, x3, x4), and we aim to predict x3, we would consider (x1, x2) as the preceding context and x4 as the subsequent context. In self-attention mechanisms, this is often implemented using triangular matrices that ensure the attention weights are zero for the current features, a technique typically referred to as autoregressive encoding (Alyafeai, AlShaibani, and Ahmad (2020)). Refer to Figure 9 to get an understanding of this approach. In the figure, one can see that the upper triangle of this matrix is indeed set to zero.[5]

2. Bidirectional LMs
   Bidirectional Language Models allow each token to consider all other to- kens in its context. So, for predicting x3 in the sequence (x1, x2, x3, x4), x3 would be influenced by (x1, x2, x4). However, this makes the task of predicting the next word somewhat redundant since any token can look ahead to the next one. To circumvent this issue, the literature often describes the use of masked language models (Alyafeai, AlShaibani, and Ahmad (2020)).

3. Masked LMs
   Masked Language Models are commonly employed in bidirectional LMs and involve concealing some tokens within the context. The model's objective is then to infer these hidden tokens. These are indicated with a [MASK] label. For instance, the sequence (x1, x2, [MASK], x4) would prompt the model to deduce that the masked token is x3. This approach is often termed denoising autoencoding.(Alyafeai, AlShaibani, and Ahmad (2020)). Refer to Figure 10 for a visualization of this approach.

The main goal of the latest language model methods is to get around the need for lots of labeled data. Before, training big models meant having people label a huge amount of data for things like figuring out feelings in text or translating languages. This work was expensive and also limited how well the models could understand language, because they only learned from a limited set of examples.

The new language models change this by teaching themselves. They don't need people to label the data anymore. Unidirectional models look at the words that come before

[5]This is usually done by setting the upper triangle tokens of the dot-product of QK^{T} to $-\infty$ before applying the $softmax$ function of $\mathbf{S}$ in Equation 9.

and try to guess the next word. Bidirectional, i.e. Masked models, look at words before and after a certain spot to guess a hidden word in that spot.

By learning this way, these models can use much more information to get better at understanding language. They don't have the same limits as before because they don't rely on human-labeled data. This means they can learn from just about any piece of writing out there, which helps in generalizing and understanding human language.

$$\begin{array}{cc} & \begin{array}{ccccc} \text{T1} & \text{T2} & \text{T3} & \text{T4} & \text{T5} \end{array} \\ \begin{array}{c} \text{T1} \\ \text{T2} \\ \text{T3} \\ \text{T4} \\ \text{T5} \end{array} & \begin{pmatrix} a_{11} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} & 0 \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix} \end{array}$$

Figure 9: Attention matrix for a unidirectional language model with five tokens.

$$\begin{array}{cc} & \begin{array}{ccccc} \text{T1} & \text{T2} & \text{T3} & \text{T4} & \text{T5} \end{array} \\ \begin{array}{c} \text{T1} \\ \text{T2} \\ \text{T3} \\ \text{T4} \\ \text{T5} \end{array} & \begin{pmatrix} a_{11} & a_{12} & m & a_{14} & a_{15} \\ a_{21} & a_{22} & m & a_{24} & a_{25} \\ a_{31} & m & a_{33} & a_{34} & m \\ a_{41} & a_{42} & m & m & a_{45} \\ a_{51} & a_{52} & a_{531} & m & a_{55} \end{pmatrix} \end{array}$$

Figure 10: Bidirectional attention matrix with randomly masked tokens m. The masked tokens should be predicted by the model by taking all the non-mask tokens into account. This is also why this approach is thought of bidirectional, since all the tokens before and after the masked token are considered.

2.7 Transfer Learning

Historically, NLP models began with randomly set parameters, but the advent of Transfer Learning has revolutionized this approach. By pretraining on a general task before fine-tuning on a specific one, models now learn more quickly and require less data for fine-tuning. While Transfer Learning was once predominantly linked with adapting ImageNet-trained neural networks [6] to new vision tasks, recent progress in NLP has

[6]ImageNet is a large database of labelled images.

made it feasible to apply this technique to language-based applications as well (Malte and Ratadiya (2019a)).
This general task allows the model to grasp a broad understanding of human language. An example pipeline would look like this: Start by pretraining the model on a generic task like predicting the next token given the previous tokens using the WikiText[7] Dataset Malte and Ratadiya (2019a) and then finetuning the model on a classfication task. There multiple other approaches, each using their speficic model-architecture and pretraining technique. To summarize them and generate an overview the next section introduces several models that have emerged after the original Transformer came out.

2.8 Emerged Models

Since the inception of the original transformer model, a large number of varieties of the original transformer have emerged. Researchers have found that the components of the Transformer's architecture all serve different purposes and hence can be isolated. Therefore, three types of models came into view.

2.8.1 Encoder models

Firstly, Encoder models focus on understanding input sequences as a whole and creating encoded representations, which contain the relevant information and relations. They can be assigned to the group of Bidirectional LMs. Referring to Figure 6, encoder models make use of the left subnetwork of the Transformer only. One example is the BERT model (Devlin et al. (2018)). As this work focuses on encoder models, the BERT model and its variants will be introduced detailed in the following sections.

2.8.2 Decoder models

Secondly, decoder models are used for generative tasks of predicting the next word in a sentence. In contrast to encoder models, decoder models do not have access to all of the input data at once. When given an input sequence all words, or tokens, after the current word are masked. In easy terms, the model should not 'cheat' in predicting the next word by already seeing the correct word in the sequence that comes after the current word. The decoder model belongs to the group of unidirectional LMs. Examples are the GPT-family (which stands for Generative Pretrained Transformer) of OpenAI (Brown et al. (2020)) or the Conditional Transformer Language Model (Keskar et al. (2019)).

[7]A large general-purpose dataset that consists of 28,595 preprocessed articles and 103 million words.

2.8.3 Encoder-Decoder models

Lastly, Encoder-Decoder models are typical sequence-to-sequence models, where one sequence needs to be transformed into another sequence. The Transformer is the classic example of this model. The Encoder creates a context representation of the input sequence and the decoder uses the context and the output sequence to generate the prediction. The Decoder uses self-attention and encoder-decoder-attention to establish relations between input and output sequences. Self-attention blocks in the decoder in this composite model of encoder and decoder are also unidirectional. Machine translation is a classic use-case for these models. Examples are the T5 (Raffel et al. (2019)) or Google's Neural Machine Translation System (Wu et al. (2016)).

2.9 Detailled overview of BERT-type encoder models

BERT, or Bidirectional Encoder Representations from Transformers, represents a significant advancement in the field of natural language processing. Introduced by Devlin et al. (2018), BERT addresses the challenge of integrating bidirectional context in Transformer-based models. Traditional approaches to bidirectionality were limited, as they allowed words to implicitly 'see themselves' in a sequence, which is not conducive to language modeling. Typically, models were constrained to either left-to-right or right-to-left contexts, but a truly bidirectional model could provide a more comprehensive understanding of the sequence context.

2.9.1 BERT architecture

BERT makes use of the Transformer's architecture while ignoring the decoder part of the model. To make BERT variable for a variety of down-stream tasks, the model is able to represent both a single and a pair of sentences[8] in one input token sequence[9]. Distinct sentences are packed together by separating them with a separation token ([SEP]) and by adding a learned embedding $S \in \mathbb{R}^{(\mathrm{n}+1)\times \mathrm{d}_{\mathrm{model}}}$ [10] to every token signaling the model whether the token belongs to sentence A or sentence B.

The authors of the BERT paper make use of WordPiece embeddings. WordPiece is a word tokenization technique that helps by splitting up big or uncommon words into smaller, more common pieces. This makes it easier for the model to deal with words it

[8] A sentence is termed as an 'arbitrary span of contiguous text, rather than an actual linguistic sentence' (Devlin et al. (2018)).

[9] A sequence is termed as the input sequence which consists of token to BERT. It may be a single sentence or two sentences combined (Devlin et al. (2018).)

[10] The row dimension is $n + 1$ because the class token is added at the beginning.

doesn't see often and helps it understand and represent words in languages that have lots of word endings or stick words together (Song et al. (2020)).

In the BERT model, a special classification token ([CLS]) $C \in \mathbb{R}^{1 \times \mathrm{d_{model}}}$ gets added to the input token sequence. The final hidden state of this token is used as the encoded representation context vector of the whole input sequence for classification tasks.

Contrary to the original Transformer, BERT makes use of self-learned Position Embeddings $P \in \mathbb{R}^{(\mathrm{n}+1) \times \mathrm{d_{model}}}$. instead of predetermined, non-learned sinusoid positional encodings. This allows BERT to optimize positional representations during the training process, potentially capturing more nuanced positional relationships.

The final representation T_i for a given input token is constructed by summing the corresponding input token embedding E_i, segment embedding, and position embedding (Devlin et al. (2018)). This can then be expressed as:

$$T_i = E_i + S_i + P_i \tag{17}$$

and for the whole sequence

$$T = Concat_{col}(C, E) + S + P \tag{18}$$

where $Concat_{col}(E, C)$, S, $P \in \mathbb{R}^{(\mathrm{n}+1) \times \mathrm{d_{model}}}$ to allow for element-wise addition.

Refer to Figure 11 for a visual representation of BERT's input processing.

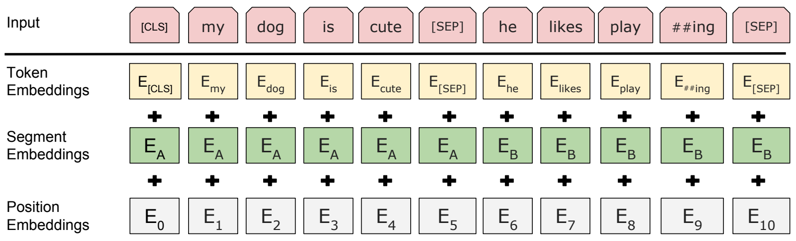


Figure 11: Visualization of BERT's input representation, which combines token embeddings, segmentation embeddings, and position embeddings.. (Devlin et al. (2018).

The authors of the BERT paper proposed two BERT models with different sizes for N number of attention blocks, d_{model} the dimension of the embeddings inside the model and A the number of attention heads for multihead-attention inside each attention-block:

- $BERT_{BASE}(N = 12, d_{model} = 768, A = 12, TotalParameters = 110M)$
- $BERT_{LARGE}(N242, d_{model} = 1024, A = 16, TotalParameters = 340M$

2.9.2 BERT pretraining

BERT innovates with its Masked Language Model (MLM) approach, where 15% of input tokens are randomly masked. The model is then trained to predict these masked tokens, enabling it to capture context from both directions of a sequence. The masking strategy is threefold:

- 80% of the time, tokens are replaced with a [MASK] token.
- 10% of the time, tokens are replaced with a random token.
- The remaining 10% of the time, the original token is retained.

In addition to MLM, BERT employs a Next Sentence Prediction (NSP) task to improve its understanding of sentence relationships. The model is trained to predict whether a sentence logically follows another, which is essential for downstream tasks such as question answering and natural language inference. The training set for NSP is composed of a balanced mix of 50% positive examples, where sentences are actual neighbors, and 50% negative examples, where the second sentence is randomly selected (Devlin et al. (2018)).

2.9.3 BERT fine-tuning

BERT's architecture allows it to be fine-tuned with smaller datasets and less computational power than required for pretraining. This fine-tuning has led to significant improvements over previous state-of-the-art results in a variety of NLP tasks, including natural language inference, question answering, and semantic similarity. The fine-tuning process adapts BERT to specific tasks, which can be categorized as follows:

- Single Sentence Classification Tasks: These involve adding layers to the [CLS] token's embedding and processing the input sequence with a preceding [CLS] token.
- Sentence Pair Classification Tasks: In these tasks, two sentences are separated by a [SEP] token, and classification layers are added to the [CLS] token.
- Question Answering Tasks: BERT is adapted to predict the answers to questions based on the context provided.
- Single Sentence Tagging Tasks: These tasks involve adding tagging information to the output for tasks like named entity recognition.

Despite the slower convergence due to the MLM task, BERT demonstrates immediate improvements in accuracy. For the NSP task, it achieves an accuracy rate between 97% and 98%, underscoring its capability to discern the relationship between two sentences effectively (Malte and Ratadiya (2019b)). We will focus on the context vector representation the [CLS] token generates in this work. The relation between fine-tuning and pretraining phase can be seen in Figure 12.

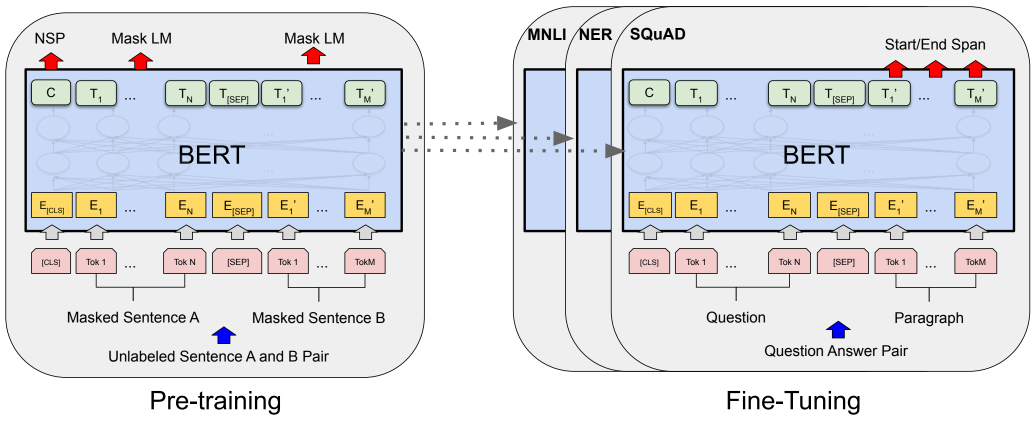


Figure 12: BERT's training involves two main stages: pretraining and fine-tuning. The core model architecture remains consistent across both stages, with the exception of the output layers. The model parameters obtained from pretraining are used as starting points for various downstream tasks. In the fine-tuning stage, all of these parameters are adjusted to better suit the specific task at hand. Special tokens, such as [CLS] at the beginning of inputs and [SEP] to separate segments like questions and answers, are used in the process (Devlin et al. (2018)).

2.9.4 BERT variants

The success of BERT has spawned a variety of adaptations and improvements, each aiming to enhance the model's performance or to tailor it to specific applications. These variants build upon the original BERT architecture, often modifying the pretraining tasks, model size, or input representations to achieve better results or more efficient training. Some notable BERT variants include:

- **RoBERTa (A Robustly Optimized BERT Pretraining Approach):** This variant modifies the BERT pretraining procedure, including training the model longer, with more data, and on longer sequences, which leads to improved performance on downstream tasks (Liu et al. (2019)).
- **DistilBERT (Distilled BERT):** DistilBERT is a smaller, faster, cheaper, and lighter version of BERT. It retains 97% of BERT's performance on downstream tasks while being 40% smaller and 60% faster (Sanh et al. (2019)).
- **ALBERT (A Lite BERT):** ALBERT restructures BERT's architecture to reduce the model's memory consumption and increase its training speed. It achieves this by factorizing the embedding layer and sharing parameters across layers (Lan et al. (2019)).

2.10 Summary

This section has provided a comprehensive exploration of the evolution of language modeling techniques. We began with the foundational RNNs, progressed through the advancements brought by Encoder-Decoder RNNs, and culminated with the game-changing development of Transformers and their subsequent variants. Particular emphasis was placed on the BERT model, which stands as a pivotal encoder-based language model that has significantly influenced the field.

Moving forward, we will shift our attention to the limitations inherent in traditional attention mechanisms and explore potential strategies to address these challenges. The remainder of this thesis will be dedicated exclusively to examining and enhancing encoder models.

3 Limits of Attention

3.1 Quadratic complexity

The computation of the attention weights **S** involves the scaled dot-product of the query matrix Q and the transpose of the key matrix K resulting in a matrix of scores with quadratic complexity with respect to the input sequence length n. Specifically, the operation QK^{T} requires n^2 time and memory, as it computes a score for each pair of tokens in the sequence (Tay et al. (2020)). As illustrated in Figures 10 and 11, this results in a full $n \times n$ matrix representing the attention scores before the $softmax$ normalization. The complexity of this operation is given by:

$$\text{Score Matrix} = \frac{\mathbf{QK}^{\mathrm{T}}}{\sqrt{d_k}} = \mathcal{O}(n^2) \tag{19}$$

After the score matrix is computed, the $softmax$ function is applied to obtain the attention weights **S**. The $softmax$ is an element-wise operation that scales with the size of the score matrix but does not increase the quadratic complexity:

$$\mathbf{S} = softmax\left(\frac{\mathbf{QK}^{\mathrm{T}}}{\sqrt{d_k}}\right) = \mathcal{O}(n^2) \tag{20}$$

The final attention output **A** is then obtained by multiplying the attention weights S with the value matrix V. This multiplication is also quadratic in complexity, but since it follows the computation of S, the overall complexity remains the same. Thus:

$$\mathbf{A} = \mathbf{SV} = \mathcal{O}(n^2) \tag{21}$$

The self-attention mechanism's quadratic cost affects both the training and inference speeds of Transformers. Generally, memory restrictions are more applicable to training (because of gradient updates and the need to store each activation of a forward pass in memory) and less to inference (no gradient updates). While it is a significant factor, it only accounts for part of the total computational expense. The feedforward layers within each attention block also require substantial computation, approximately half of the total compute time (Tay et al. (2020)). Although the complexity of feedforwards is linear in relation to the sequence length, it remains a considerable part of the overall computational cost. However, in this work, we will focus on the bottleneck of the attention mechanism solely.

3.2 Consequences of the quadratic complexity of self-attention

The introduction of the Transformer architecture marked a significant leap forward in the ability to develop complex models efficiently, thanks to its groundbreaking approach to fully parallelize attention computations. This innovation has paved the way for a new generation of models that are both more sophisticated and larger than their predecessors. The key to the Transformer's success lies in its ability to process inputs in parallel, a stark contrast to the sequential nature of older models like RNNs, which struggled with long sequences due to vanishing and exploding gradients.

Despite these advancements, the drive to enhance the Transformer's capabilities has encountered a critical bottleneck: the architecture's computational demands increase quadratically with the length of the input sequence. This quadratic complexity presents a significant challenge, not because of an inherent limitation in handling long sequences per se, but due to the exponential growth in required computational resources. As a result, the practical application of Transformers is heavily influenced by the constraints of available computational power.

In the case of models such as BERT, which typically limit input sequences to 512 tokens (Devlin et al. (2018)), the challenge of computational efficiency becomes acutely relevant. The self-attention mechanism, a core feature of the Transformer that enables its high efficiency in parallel processing, is also the source of its computational intensity. This creates a pivotal tension between the potential to process longer sequences and the realistic limitations imposed by computational resources.

The consequence of this bottleneck is significant, extending beyond the mere ability to process longer texts. It highlights a fundamental limitation in the scalability of current AI models under computational constraints, making them less adaptable to tasks that demand extensive data processing. This limitation also acts as a driving force for innovation, pushing researchers to explore alternative language models and techniques that can circumvent these computational challenges.

In response to this, the subsequent section shifts focus towards examining solutions that aim to address the computational bottleneck of Transformers. The goal is to propose a model that integrates various strategies to alleviate the computational demands, thereby enhancing the efficiency of processing within the constraints of available resources. This exploration is essential for pushing the boundaries of what is possible with AI models, ensuring they can continue to evolve and tackle increasingly complex tasks without being hindered by computational limitations.

4 Addressing Quadratic Complexity Constraints in Attention Mechanism

4.1 Introduction

In the following section, we will explore innovative strategies designed to address the challenges posed by the quadratic complexity inherent in attention mechanisms. Our discussion will be structured around three primary paradigms, each targeting a distinct aspect of these limitations.

Firstly, we will delve into methods aimed at optimizing the attention matrix itself. These techniques primarily focus on reducing the size of the attention matrix, consequently diminishing the computational resources required. This is essentially achieved by 'sparsifying' the attention matrix. Among the notable approaches in this domain are:

- The LinFormer, which leverages the low-rank property of attention matrices to approximate them, thus reducing complexity to a linear scale (Wang et al. (2020)).
- BigBird, introducing a randomly sparsified attention matrix(Zaheer et al. (2020)).
- BlockBert, which sparsifies the attention matrix by confining attention scores to a direct local neighborhood, effectively reducing computational overhead (Qiu et al. (2019)).

The second area of focus will be on cutting-edge model architectures designed to facilitate memory-efficient computation while maintaining the overall depth and intricacy of the network. In this context, we will examine the use of 'reversible' blocks.

Lastly, we will introduce a novel training approach for Masked Language Models (LMs) termed 'Electra'. This method enables the accelerated training of large models, yielding comparable results in a shorter timeframe. Although Electra does not directly lessen memory requirements, it significantly reduces the costs associated with training by shortening the duration needed to train a Masked LM.

4.2 Sparsifying the attention matrix

To overcome the limitations imposed by the quadratic complexity of standard attention mechanisms, it is essential to sparsify the attention matrix. This entails to design a scheme whereby each token x_i for $i \in 0, 1, \ldots, n$ attends selectively to a subset of tokens indexed by $S_i \subseteq \{0, 1, \ldots, n\}$. Such an approach ensures that not every token is attended to by every other, thereby reducing the computational burden of each attention operation. This reduction introduces a critical trade-off: the smaller the size of S_i, the lower the computational cost, yet with the increasing risk of omitting interactions crucial for capturing the intended semantics of the input.
The selection of the subset S_i need not be static but can vary depending on the token at position i. Our objective is to find a set of positions S_i for each token such that the resulting sparsified attention matrix closely approximates the full attention matrix in the best possible way.

4.2.1 Sliding window Attention

One way to sparsify the attention matrix is to use sliding windows as fixed pattern around the current token x_i. For a fixed window size w each token attends to $\frac{w}{2}$ tokens on each side (Beltagy, Peters, and Cohan (2020)). The set S_i, representing the indices of tokens within the window of x_i, can be formalized as:

$$S_i = \{i - \frac{w}{2}, \ldots, i, \ldots, i + \frac{w}{2}\} \tag{22}$$

The authors of the paper which introduced the concept explained: "Using multiple stacked layers of such windowed attention results in a large receptive field, where top layers have access to all input locations and have the capacity to build representations that incorporate information across the entire input, similar to CNNs"(Beltagy, Peters, and Cohan (2020)). This can also be seen as a giving the model an inductive bias where human assumptions are coded into the model's structure and architecture. By doing this the model is forced to learn in a specific way, here by limiting its attention to a local neighborhood. The window size w may also vary across different layers and as such is a tunable hyperparameter.
We adjust the current computation of the attention matrix $\mathbf{S}$ with:

$$\mathbf{S}_i^{\mathrm{w}}(q_i, k_{S_i}) = softmax(\frac{q_i {k_{S_i}}^{\mathrm{T}}}{\sqrt{d_k}}) \tag{23}$$

for each token x_i. To compute the final attention output $\mathbf{A}^{\mathrm{w}}$ we need to consider the local context of the attention matrix $\mathbf{S_i}^{\mathrm{w}}$ and therefore start by calculating $\mathbf{A_i}^{\mathrm{w}}$ as:

$$\mathbf{A_i}^{\mathrm{w}}(q_i, k_{S_i}, v_{S_i}) = \mathbf{S_i}^{\mathrm{w}}(q_i, k_{S_i}) v_{S_i} \tag{24}$$

To finalize the attention output, each token-wise attention output $\mathbf{A_i}^{\mathrm{w}}$ is concatenated row-wise:

$$\mathbf{A}^{\mathrm{w}}(Q,K) = Concat_{row}(\mathbf{A}_0^{\mathrm{w}}(q_0, k_{S_0}, v_{S_0}), \ldots, \mathbf{A}_n^{\mathrm{w}}(q_n, k_{S_n}, v_{S_n})) \tag{25}$$

where $q_i \in \mathbb{R}^{1 \times \mathrm{d_{model}}}$ is the query vector for token x_i and $k_{S_i} \in \mathbb{R}^{\mathrm{w} \times \mathrm{d_{model}}}$ is a subset of the key matrix K that corresponds to the window S_i containing only the keys that token x_i is allowed to attend to and finally, $v_{S_i} \in \mathbb{R}^{\mathrm{w} \times \mathrm{d_{model}}}$ the subset of the value matrix V. The sliding-window attention matrix is of shape $n \times w$ instead of $n \times n$ and the final attention output is again of shape $n \times d_{model}$

Since each token attends to w other tokens the complexity of the sliding window attention mechanism can formally described as:

$$\mathbf{S}^{\mathrm{w}} = \mathcal{O}(nw) \tag{26}$$

and thus scales linearly with the input sequence length n. To ease the understanding in the following section, a set H is defined which contains all the additional hyperparameters that have to be considered using the novel approaches. Currently H can be described as:

$$H = \{W\}$$

where W contains the window sizes w_l for each layer l. Refer to Figure 13 to understand sliding window attention visually.

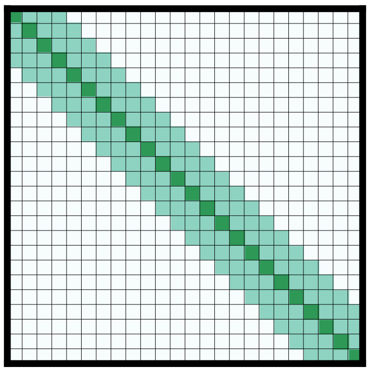


Figure 13: Sliding window attention. Each tokenx_i only attends to its local neighborhood. In this case $w = 6$ (Beltagy, Peters, and Cohan (2020)).

4.2.2 Dilated sliding window attention

Similar to dilated (or atrous) CNNs (Chen et al. (2016)) where some values are skipped with a fixed rate d when computing the convolution, the concept of dilation can also be applied to the attention mechanism. The sliding-window approach of the previous

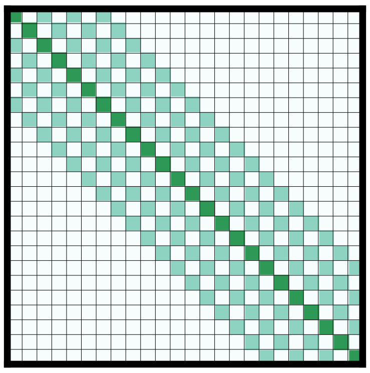


Figure 14: Dilated sliding window attention with $w = 6$ and $d = 2$. There is a gap of $d-1 = 1$ between each token in the local neighborhood (Beltagy, Peters, and Cohan (2020)).

section is constrained on short distances since the model only is able to attend to tokens inside the window size w making it impossible to capture long-range dependencies. To overcome this limitation dilated sliding window attention was introduces. Instead of using a sliding window of size w the window is extended to size dw but with gaps of size $d-1$ between tokens (Beltagy, Peters, and Cohan (2020)). This means each token x_i still attends to a total number of w tokens but the the Set S_i changes to

$$S_i = \{i - \frac{dw}{2}, i - \frac{dw}{2} + d, \ldots, i, \ldots i + \frac{dw}{2} - d, i + \frac{dw}{2}\} \tag{27}$$

The range to which tokens x_i increases from $\frac{w}{2}$ in both direction to $\frac{dw}{2}$ in both directions giving the model a larger context to work with and learn from. This making it suitable for very long input sequences without a significant increase in computational costs.

Since each token still only attends to w other tokens the complexity of the dilated sliding window attention mechanism can formally be described also as:

$$\mathbf{S}^{\mathrm{w}} = \mathcal{O}(nw) \tag{28}$$

The computation for $\mathbf{A}^{\mathrm{w}}$ and everything else stays the same. The only thing that is changed is the set S_i. Consequently, H is adjusted to:

$$H = \{W, D\}$$

where D contains the dilation rates d_l for each layer l. Refer to Figure 14 for a visualized explanation.

4.2.3 Global attention connection

For masked LMs the model uses the context around the masked token to predict it. In BERT-style models an additional [CLS]-token is added at the beginning which represents the whole input sequence for further tasks like classification or clustering. The goal is to encode the the information of the input sequence inside the [CLS]-token embedding of the last attention block. When using (dilated) sliding window attention, the attention computed is not flexible enough to learn 'good' task-specific representations. Therefore global attention is added. Global attention in this context refers to the concept of adding full attention computations on few pre-selected input locations. This is a symmetric operation: A token with global attention attends to all tokens across the sequence and all tokens in the sequence attend to it (Beltagy, Peters, and Cohan (2020)).

In our case, the windowed and dilated attention are not flexible enough to learn task-specific representations. Accordingly, we add "global attention" on few pre-selected input locations. Importantly, we make this attention operation symmetric: that is, a token with a global attention attends to all tokens across the sequence, and all tokens in the sequence attend to it (Beltagy, Peters, and Cohan (2020)). The set L contains all the positions of token with global attention. That is for all $i \in G$, the following is stated:

$$S_i = S$$

which means that the local attention subset equals the full attention set. Additionally, since this is a symmetric operation (Beltagy, Peters, and Cohan (2020)) each element of G has to be added to each S_i stating:

$$S_i = S_i \cup L$$

The size of L, i.e. the number of global attention tokens is small relative to the independent tokens n the complexity of the combined local and global attention computation is still $\mathcal{O}(nw)$ (Beltagy, Peters, and Cohan (2020)).
The hyperparameter set H is adjusted accordingly:

$$H = \{W, D, L\}$$

Figure 15 provides a visualized explanation of the global attention.

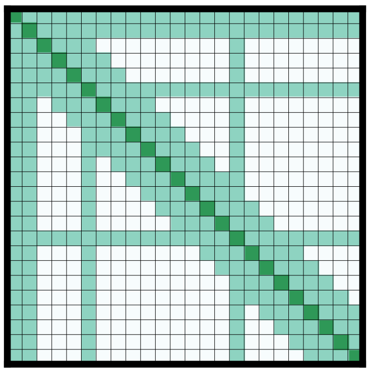


Figure 15: Global attention combined with sliding-window attention for $w = 6$, $L = \{0, 1, 5, 15\}$ and $d = 1$. The first row is the corresponding [CLS]-token which uses global attention to capture the information of the whole input sequence (Beltagy, Peters, and Cohan (2020))

.

4.2.4 Locality Sensitive Hashing

One of the downsides of restricting attention to a fixed size of tokens around the attention token is that important long range tokens which may capture more relevant information than the tokens in the direct neighborhood will be thrown away, hence leading to a significant loss of information. The Locality Sensitive Hashing (LSH) attention mechanism proposed by Kitaev et al. is another way to sparsify the attention matrix, however, instead of limiting attention to the direct neighborhood, it tries to compute attention only between somewhat "similar" tokens (Kitaev, Kaiser, and Levskaya (2020)).

Since tokens are represented by high-dimensional embedding vectors inside the model, the goal is to formulate an algorithm that finds the l nearest neighbors in high-dimensional spaces. This is achieved by using a hashing function h which assigns each vector x to a hash $h(x)$. This function is 'locality-sensitive,' meaning vectors that are close to each other are likely to have the same hash, while distant ones do not (Kitaev, Kaiser, and Levskaya (2020)).

Let $R \in \mathbb{R}^{d_{model} \times \frac{b}{2}}$ be a random matrix and where b is the number of hashes. The hashing function h can then be described as:

$$h(x) = argmax(Concat_{row}(xR, -xR)) \tag{29}$$

Adoni et al. demonstrated that this is a practical and valid LSH schema, which is not only easy to implement but also effective for individual vectors and batches of vectors (Andoni et al. (2015)).

We can refer to equation 23 to understand how attention is computed in a sparsified attention matrix. The only difference is that our set S_i changes. Instead of containing the neighboring position token indices, it now consists of the tokens within the hashed bucket. However, the authors also omit for LSH-Attention the scaling factor $\sqrt{d_{model}}$ and set $Q = K$. Equation 23 then changes for LSH-Attention to:

$$\mathbf{S}_i^{\mathrm{w}}(q_i, k_{S_i}) = softmax(q_i {k_{S_i}}^{\mathrm{T}}) \tag{30}$$

for each token x_i while the rest of the equations 23-25 remain consistent. The set S_i can now be depicted as:

$$S_i = \{j : h(q_i) = h(k_j)\} \tag{31}$$

Meaning the set S_i consists of all positions j such that the hash value $h(q_i)$ equals the hash value of $h(k_j)$.

The process involves organizing queries by grouping them into buckets based on their hash values and arranging them in order within each bucket. To address the issue where a bucket might have a disproportionate number of queries compared to keys, the authors recommend normalizing each query vector q_j by its magnitude, setting $k_j = \frac{q_j}{||q_j||}$. This adjustment ensures that the hash values for the queries q_j and the keys k_j match, leading to an equal distribution of queries and keys in each bucket.

Once sorted, these query-key pairs are divided into segments, each containing $m$pairs. In cases where a bucket has more than m pairs, a unique attending rule applies: a query in one segment can refer to all keys in the same bucket from the previous segment, but it cannot refer to keys outside its current segment. This rule, however, doesn't apply to other queries in the same bucket; they are not allowed to refer to a key that's outside their current segment. This method is illustrated in Figure 16.

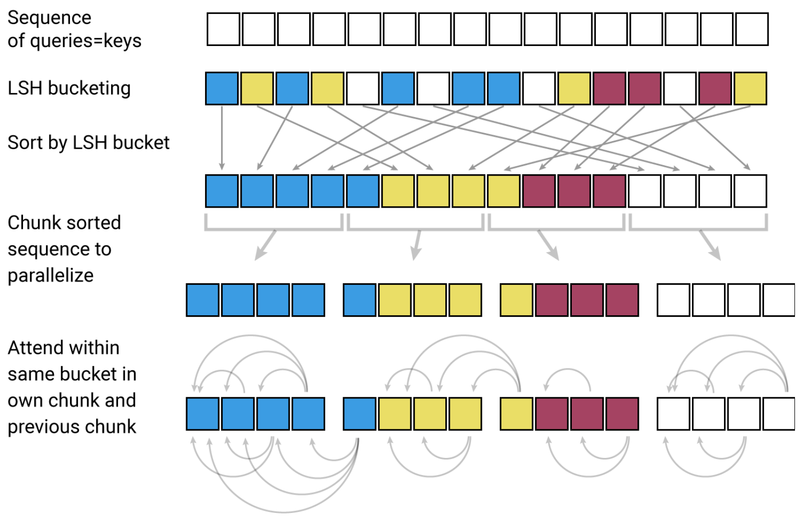

Sequence of queries=keys
LSH bucketing
Sort by LSH bucket
Chunk sorted sequence to parallelize
Attend within same bucket in own chunk and previous chunk

Figure 16: LSH attention algorithm: The sequence is hashed into buckets, then sorted according to their bucket number and position in the bucket. Following this the sequence is chunked into parts of length m. As shown queries outside a chunk can only attend backward.

The authors state that their attention mechanism has a complexity of

$$\mathbf{S}^{\mathrm{w}} = \mathcal{O}(n \log n) \tag{32}$$

which can be more efficient than the local attention of the previous section depending on the sequence length n and the chosen window size w.

Finally, our hyperparameter set H can then be stated as:

$$H = \{b, m\}$$

Our set consists of the number of hashes b where in general more hashes mean the probability that similar tokens are grouped while the computational complexity increases and the chunk length m which stands for the size of the parts the sequence is chunked into after sorting it according to their buckets and positions.

4.3 Reversible residual layers

4.3.1 Backpropagation summary

The hardware limits because of quadratic complex are especially relevant when training the model. That is, during training every output of each layer, also called activation, needs to be stored in memory. This is a direct consequence of how backpropagation and the chain-rule works. During the forward-pass each computation is stored as a node in a computational graph G. Let $v_0, v_1, v_2, \ldots, v_K$ be the nodes of G where v_k is the cost function C. Each node v_i can be described as a function f_i of its parent nodes. Looking at equations 13 to 16 this process can be seen quite well. Each computation is a function of the previous computation. In backpropagation the total derivative denoted with a bar symbol

$$\bar{v_i} = \frac{\delta C}{\delta v_i}$$

is computed. To determine $\bar{v_i}$ the backprogation algorithm iterates over the nodes in reverse order and applies the following rule:

$$\bar{v_i} = \sum_{j \in Child(i)} (\frac{\delta f_i}{\delta v_i}) \bar{v_j} \tag{33}$$

which is a recursive procedure in $\bar{v_j}$ and $\frac{\delta f_i}{\delta v_i}$ is the Jacobian matrix[1]. $Child(i)$ denotes the set of nodes that are directly influenced by v_i. That is they are children of the node v_i in terms of the computational graph. The summation of all children accounts for the contributions of all child nodes to the gradient of v_i. Each term in the summation

[1]Jacobian matrix is a matrix of first order derivatives of a vector-valued function.

represents one path from v_i to the output. It captures how v_i changes v_j and how v_j changes C. This is the total derivative of C with respect to v_i. Summing up, being able to compute $\bar{v}_i$ requires all outputs (or activations) of the children nodes to be saved in memory (i.e. as nodes in the computational graph) such that they can be accessed when applying equation 29. The larger the computational graph, the larger the memory requirements.

4.3.2 Residual networks summary

Residual networks were already established in an earlier section. However, as they are the underlying principle of the reversible residual network which is introduced in this section, they will be explained more rigorously now. At the core of a residual network is the idea of a residual block. Each block, be it an attention block (introduced in section 2.5.6), a convolutional block, or a multi-layer feedforward block, implements a specific transformation. Let $\mathbf{F}$ be a residual block that maps an input x to an output y. A residual connection can then be described as:

$$Z = \mathbf{F}(x) + x \tag{34}$$

where Z is the output of the residual block. Residual networks stack multiple residual blocks where each block follows its own methodology by mapping an input x to an output y. Residual connections help stabilize gradients and allow for better information flow. This improves the training of very deep and complex models (He et al. (2015)).

4.3.3 Reversible layer

Residual blocks can contain multiple layers making them possibly very large. Large models increase the size of the computational graph and therefore also the need for more memory. When considering attention block this adds even more memory requirements due to the quadratic complexity discussed earlier. To overcome this problem, reversible residual layers were proposed. The core idea is to build a model that does not need to store the activation of the layers apart from the last one in memory. To achieve this each residual connection is adjusted to be invertible. Then each activation can be recomputed when needed in equation 29. Such invertible residual connections are called reversible blocks (Gomez et al. (2017)).
A reversible block expects two inputs x_1, x_2 and produces two outputs y_1, y_2 with the following transformation:

$$\begin{aligned} y_1 &= x_1 + \mathbf{F}(x_2) \\ y_2 &= x_2 + \mathbf{G}(y_1) \end{aligned} \tag{35}$$

where **F** and **G** are residual blocks. The activations of the layers can be reconstructed from the succeeding layer's activations with the following (Gomez et al. (2017)):

$$\begin{aligned} x_2 &= y_2 - \mathbf{G}(y_1) \\ x_1 &= y_1 - \mathbf{F}(x_2) \end{aligned} \tag{36}$$

By knowing the outputs y_1 and y_2 each input activation of the model can be easily restored.
For this work attention and feedforward blocks are used as residual blocks. The transformer attention block consists of two residual connections: The first for the attention output and the second for the feedforward layer.
As the reversible blocks expect two inputs x_1, x_2 and the block input is x, x has to be partitioned channelwise, that is dimensionwise. Considering attention blocks, we can say that $x \in \mathbb{R}^{\mathrm{n} \times \mathrm{d_{model}}}$ is split in $x_1 \in \mathbb{R}^{\mathrm{n} \times \frac{\mathrm{d_{model}}}{2}}$ and $x_2 \in \mathbb{R}^{\mathrm{n} \times \frac{\mathrm{d_{model}}}{2}}$ by halving the dimension size.
To make full usage of reversible blocks, multiple of them are stacked and only the activation of the last block is stored. In summary, reversible blocks allow to reconstruct previous activations when needed while only saving the last activation in memory.

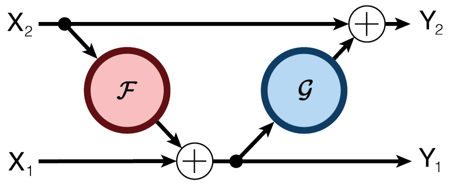


Figure 17: The forward pass of a reversible block (Gomez et al. (2017)). Based on equation 31.

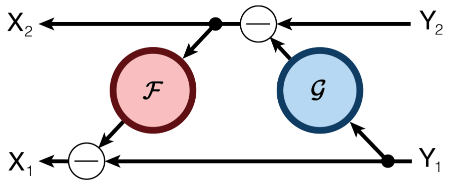


Figure 18: The recomputation of the input activations a reversible block (Gomez et al. (2017)). Based on equation 32.

4.3.4 Backward pass for reversible layers

To get an understanding of the backpropagation procedure for reversible layers it is helpful to rewrite forward:

$$z_1 = x_1 + \mathbf{F}(x_2) \tag{37}$$

$$y_2 = x_2 + \mathbf{G}(z_1) \tag{38}$$

$$y_1 = z_1 \tag{39}$$

and backward, i.e. reverse equations (Gomez et al. (2017)):

$$z_1 = y_1 \tag{40}$$

$$x_2 = y_2 - \mathbf{G}(z_1) \tag{41}$$

$$x_1 = z_1 - \mathbf{F}(x_2) \tag{42}$$

Note that y_1 and z_1 represent different nodes of the computational graph. z_1 is an intermediate state which simplifies the gradient computation inside the block. For a better understanding the computational graph for a reversible block is displayed in Figure 18.

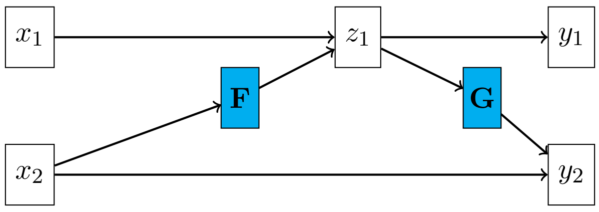


Figure 19: The computational graph of a reversible block.

Using the computational graph to determine the total derivative $\bar{x_1}$ for example we have to follow every path from x_1 to the leaves of the graphs. By doing this we can determine which outputs are influenced by x_1 and hence for which a partial derivative has to be computed. By adding all partial derivatives of the paths, the total derivative $\bar{x_1}$ is calculated. In summary the computation of the total derivative using a graph structure to track each calculation follows two main rules:

R1 Multiply the derivatives along one path

R2 Add the derivatives of different paths

Keeping this in mind, one can derive an the gradient of the reversible block.

$$\bar{z_1} = \bar{y_1}\frac{\partial y_1}{\partial z_1} + (\frac{\partial \mathbf{G}}{\partial z_1})^T \bar{y_2} = \bar{y_1} + (\frac{\partial \mathbf{G}}{\partial z_1})^T \bar{y_2} \tag{43}$$

$$\bar{x_2} = \bar{y_2} + (\frac{\partial \mathbf{F}}{\partial x_2})^T \bar{z_1} \tag{44}$$

$$\bar{x_1} = (\frac{\partial z_1}{\partial x_1})^T \bar{z_1} = \bar{z_1} \tag{45}$$

$$\bar{\mathbf{W}_\mathbf{F}} = (\frac{\partial \mathbf{F}}{\partial \mathbf{W}_\mathbf{F}})^T \bar{z_1} \tag{46}$$

$$\bar{\mathbf{W}_\mathbf{G}} = (\frac{\partial \mathbf{G}}{\partial \mathbf{W}_\mathbf{G}})^T \bar{y_2} \tag{47}$$

where $\bar{y_1}$ and $\bar{y_2}$ (the total derivatives with respect to the loss C) are already determined and used following the chain rule.
To understand the process fully, the equations are explained in detail.

- Equation (39): To compute the total derivative of z_1, we identify all nodes in the computational graph influenced by z_1. From Figure 18, it's clear that y_1 and y_2 (through $\mathbf{G}$) are affected by z_1. Therefore, we sum the contributions from these two paths according to R2. Each path's contribution to the derivative is calculated using the chain rule. The partial derivatives with respect to y_1 and y_2 are then scaled by their respective total derivatives with respect to the loss C using R1. This is done to determine the derivative of the loss C. Taking equation (35) into account, the derivative of y_1 w.r.t. z_1 is 1 leading to the simplified second equation.

- Equation (40): x_2 influences y_2 and z_1 through $\mathbf{F}$. Also y_1 is affected by x_2 through z_1. However, we already captured this dependency with $\bar{z_1}$. Hence, we only scale with $\bar{z_1}$ instead of with the full dependency to y_1 to make use of already computed properties. Again R2 is used to add the paths and R1 to determine the derivative on the path.

- Equation (41): x_1 affects y_1 and y_2 through z_1. Since z_1 is computed by adding x_1 and $\mathbf{F}(\mathbf{x_2})$ in equation (33), z_1 is influenced by x_1 directly and hence the derivative of z_1 w.r.t. to x_1 is 1. In summary the total derivative of x_1 equals the total derivative of z_1

- Equation (42): $\mathbf{W}_\mathbf{F}$ denote the parameter of the function $\mathbf{F}$. Since the parameter affect z_1 through $\mathbf{F}$ and y_1 and y_2 through z_1 the total derivative can be determined by considering all the paths to z_1 from $\mathbf{F}$ and scale afterwards with the total derivative of z_1 to account for the affect in y_1 and y_2 and their corresponding effect on the loss C.

- Equation (43): $\mathbf{W}_\mathbf{G}$ denote the parameter of function $\mathbf{G}$. The parameter influence y_2 through $\mathbf{G}$. To determine the final derivative of C w.r.t. $\mathbf{W}_\mathbf{G}$ we scale with the total derivative of y_2.

By utilizing computational graphs, we gain a significant advantage in visualizing and conceptualizing the backpropagation process in neural networks, especially in reversible layers. The computational graph not only illustrates the forward flow of computations but also serves as a roadmap for the backward propagation of gradients.

In the context of reversible layers, focusing on subgraphs allows us to isolate specific portions of the network for detailed analysis. This isolation simplifies the understanding of gradient flows and their accumulation, particularly in complex networks with numerous intertwined paths. The key benefit of this approach lies in its ability to reduce the complexity of backpropagation to manageable, localized computations within these subgraphs.

The chain rule, a fundamental principle in calculus, underpins the backpropagation process. It enables the decomposition of derivative calculations into simpler, sequential steps. By acknowledging that $\bar{y_1}$ and $\bar{y_2}$ store all gradient information of the loss C with respect to succeeding blocks and parameters, we can effectively apply the chain rule within these subgraphs. This approach means that, to compute the overall impact on the loss C, we only need to multiply the local derivatives of y_1 and y_2 with respect to internal block parameters by $\bar{y_1}$ or $\bar{y_2}$. Such a method allows for a granular yet comprehensive view of gradient computation, concentrating on individual subgraph nodes while implicitly accounting for the broader network context.

This targeted focus on subgraphs is not just a theoretical convenience but a practical necessity in deep learning. It streamlines the computational process, reduces the potential for errors in derivative calculations, and enhances our ability to understand and optimize complex neural networks.

4.3.5 Backpropagation in reversible layers

The previous section introduced the computation of the gradient in reversible layers. To summarize the processes, a simple algorithm is derived to highlight how a individual layer should implement the backpropagation and reversible steps to correctly propagate the gradient information along the network. This is especially useful for implementing the process in code later.

Steps 2 to 4 in Algorithm 1 recompute the needed input and intermediate activations based on equations (36) to (38). Steps 5 to 9 determine the relevant gradients rooted in equations (39) to (43). Finally, the gradients w.r.t. to the parameter of the functions **F** and **G** are returned in addition to the recomputed inputs.

Algorithm 1 Algorithm for backpropagation in reversible layers

```
1: function BackpropReversibleBlock($(y_1, y_2, \bar{y_1}, \bar{y_2})$)
2:     $z_1 \leftarrow y_1$
3:     $x_2 \leftarrow y_2 - \mathbf{G}(z_1)$
4:     $x_1 \leftarrow z_1 - \mathbf{F}(x_2)$
5:     $\bar{z_1} \leftarrow \bar{y_1} + (\frac{\partial \mathbf{G}}{\partial z_1})^T \bar{y_2}$
6:     $\bar{x_2} = \bar{y_2} + (\frac{\partial \mathbf{F}}{\partial x_2})^T \bar{z_1}$
7:     $\bar{x_1} \leftarrow (\frac{\partial z_1}{\partial x_1})^T \bar{z_1} = \bar{z_1}$
8:     $\bar{\mathbf{W}}_{\mathbf{F}} \leftarrow (\frac{\partial \mathbf{F}}{\partial \mathbf{W}_{\mathbf{F}}})^T \bar{z_1}$
9:     $\bar{\mathbf{W}}_{\mathbf{G}} \leftarrow (\frac{\partial \mathbf{G}}{\partial \mathbf{W}_{\mathbf{G}}})^T \bar{y_2}$
10:    return $(x_1, x_2, \bar{\mathbf{W}}_{\mathbf{F}}, \bar{\mathbf{W}}_{\mathbf{G}})$
11: end function
```

4.4 Electra Pretraining

4.4.1 Review of current Pretraining Techniques for Masked LMs

As detailed in Section 2.6, the leading method for pretraining Masked LMs adheres to the BERT pretraining protocol. This involves taking a sentence represented by a sequence of n tokens $(x_1, x_2, x_3, ...x_n)$ and randomly substituting certain tokens, usually 15%, with a unique identifier. For instance, if x_3 is replaced, the sequence becomes $(x_1, x_2, mask, \ldots, x_n)$. The model's task is then to accurately predict the masked token using the context provided by the surrounding tokens. This process is highly automated, eliminating the requirement for labeled data during the pretraining phase.

4.4.2 Generative and Discriminative models

Deep learning focuses on developing complex and layered models that encapsulate various data types, essentially representing their probability distribution (Bengio (2009)). These models typically fall into two categories: Generative and Discriminative models.

A generative model, when provided with input x and labels y, learns the joint probability distribution $p(x, y)$. From this, it calculates $p(y|x)$ and selects the most probable label y. In essence, generative models explicitly determine the distribution for each label y_i.

On the other hand, discriminative models directly learn the posterior probability $p(y|x)$, bypassing the need to understand the joint distribution $p(x, y)$. Consequently, discriminative models establish a straightforward mapping between inputs and outputs (Ng and Jordan (2001)). Refer to Figure 19 for a visualization of the two approachés.

The current trend in pretraining is predominantly a generative learning approach, where the model tries to learn the distribution of each masked token over all potential tokens.

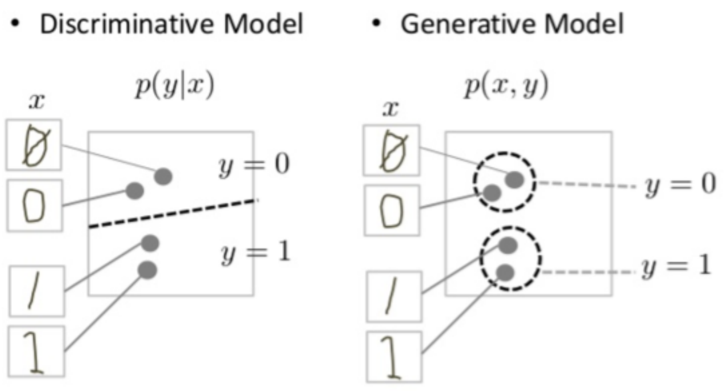


Figure 20: Distinguishing between generative and discrimnative models. As we can see discriminative models from a decision boundary and learn the mapping between input and labels directly, while generative models learn the distribution on which a classification can be done afterwards (Google Developers (2022)).

4.4.3 Discriminative Pretraining with Electra

Considering traditional Masked LMs, the models do not only face substantial computational costs due to the costly attention computation, but also because the model only learns from a limited subset of tokens per example. This subset consists of the masked tokens which usually account for 15% of the per example tokens. This leads to a rather imbalanced training, since in order to let the model learn good representation of each token a high amount of examples is required. Consequently, the computational costs can quickly escalate (Clark et al. (2020)).

As an alternative Clark et al propose a new training paradigm, the *replaced token detection* pretraining. Instead of predicting the correct token over all possible tokens, the model should learn to distinguish real input tokens from "plausible but synthetically generated replacements"(Clark et al. (2020)). This is achieved by using two BERT-like models, a generative and a discriminative one which both consist of an encoder which map an input sequence $x = (x_1, x_2, x_3 \ldots, x_n)$ to sequence of contextualized vector representation $h(x) = (h(x_1), h(x_2), h(x_3) \ldots, h(x_n))$ (Clark et al. (2020)) like the BERT Encoder we have already seen .

Given an input sequence x consisting of tokens $(x_1, x_2, x_3 \ldots, x_n)$ where tokens were randomly masked, the generative model, also called generator solely, is, again, trained on predicting the masked token. Following the masked tokens in the input sequence are replaced with the generator predictions. Let the masked input sequence be $(x_1, x_2, mask, \ldots, x_n)$, where for simplicity only the token x_3 was masked. Let y be the prediction of the generator for the masked token x_3. By replacing the masked token with t, the resulting sequence S^{post} can be written as $(x_1, x_2, y_3, \ldots, x_n)$, where x^{post}

can be seen as the with the generator outputs corrupted input sequence x (Clark et al. (2020)).

The discriminative model, or discriminator solely, is then trained to determine which tokens were corrupted given the input sequence x^{post}. Consequently, sticking to our example, the discriminator should predict that token y_3 is a replacement, while all the others are not. The whole network is then pretrained as a discriminator in contrast to traditional generative Masked LM pretraining, where the model is trained as agenerator to predict the original identies of the corrupted tokens. The authors introduce two main advantages of using the discriminative pretraining approach (Clark et al. (2020)):

- The replacement of masked tokens with artificial generated tokens from the generator fixes the mismatch that the model sees the masked tokens during pretraining but not during finetuning tasks, because the tokens are only masked for pretraining purposes (Clark et al. (2020)).
- The model learns from all input tokens instead of just the masked token subset per example. This makes it much more computationally efficient (Clark et al. (2020)).

After pretraining, only the discriminator is used for further fineuning tasks.

The generator and discriminator usually follow the same model architecture, for example the BERT architecture. However, the model size of the generator is downsampled to a fraction of the discriminator's size. The downsampling is usually done for the models hidden dimension, number of attention heads and number of attention blocks. Experiments have shown that using generators 1/2 or 1/4 of the discriminator's size have yielded the best results. The authors speculate that a too strong generator may offer a too challenging task for the discriminator, hindering it from learning effectively. This approach was named 'Electra' which stands for "Efficiently Learning an Encoder that Classifies Token Replacements Accurately" (Clark et al. (2020)).

In summary Electra frames the pretraining paradigm as a discriminative problem rather than a generative one, in order to allow for fewer computational ressources, less training time and possibly better token representations.

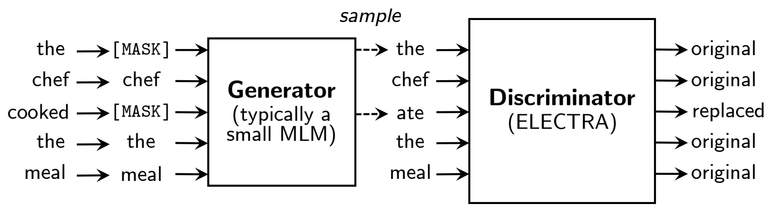


Figure 21: Replaced token detection in Electra. Masked tokens are replaced with generator predictions. The discriminator learns to recognize generated tokens. After pretraing only the discriminator is used (Clark et al. (2020)).

4.4.4 Training algorithm for Electra

Training Electra involves training the generator and the discriminator. Training the generator is crucial since the discriminator should be challenged. The token replacements of the generator should somewhat make sense. If the generator solely generates out of context words without any connection to the input sequence, the discriminator will not face any problems determining them. However, by also optimizing the generator, the corrupted tokens become more and more challenging to detect and the discriminator has to learn even better word representations. This interaction between generator and discriminator may already be familiar to some readers in General Adversarial Networks (GANs) (Goodfellow et al. (2014)). However, the generator is not trained in an adversarial manner but with maximum likelihood given the difficulty of applying GANs to textual data (Caccia et al. (2018)).

Algorithm 2 Training algorithm for Electra

```
    Input: Input sequence $x = (x_1, x_2, \dots, x_n)$ of length $n$, masked token probability $p$, generator $\mathbf{G}$, discriminator $\mathbf{D}$, loss scaling $\lambda$
 1: $m_i \sim \text{unif}\{1, n\}$ with probability $p$   for $i = 1, \dots, n$
 2: $m \leftarrow \{m_1, m_2, ...., m_l\}$ such that $l = floor(pn)$
 3: $x^{masked} \leftarrow (x_1, x_2, mask, \dots, x_n)$
 4: $y^{gen} \leftarrow \mathbf{G}(x^{masked})$
 5: $x^{post} \leftarrow (x_1, x_2, y_3^{gen}, \dots, x_n)$
 6: $y^{disc} \leftarrow \mathbf{D}(x^{post})$
 7: $\mathcal{L}_{gen} \leftarrow \sum_{i \in m} -\log(p_{y_i^{gen}|x^{masked}})$
 8: $\mathcal{L}_{disc} \leftarrow \sum_{i=1}^{n} -1(x_i^{post} = x_i)\log(y_i^{disc}) - 1(x_i^{post} \neq x_i)\log(1 - y_i^{disc})$
 9: $\mathcal{L}_x \leftarrow \mathcal{L}_{gen} + \lambda \mathcal{L}_{disc}$
10: $\min_{\theta_{\mathbf{G}}, \theta_{\mathbf{D}}} \sum_{x \in \mathcal{X}} \mathcal{L}_x$
```

Given an input sequence x, we draw a mask token m_i from a uniform distribution with a probability p. All masked tokens m_i are combined into a set m of masked tokens. The set m has a size of $l = floor(pn)$ where $floor$ rounds pn to the next smallest integer. (step 1 and step 2). We then replace the tokens m_i of x with the unique masked token identifier, for simplicity we name it $mask$ (step 3). The generator $\mathbf{G}$ generates predictions y^{gen} for all the masked tokens in x^{masked}. The masked token at position i is then replaced with the generation y_i^{gen} for all masked tokens to receive the corrupted input sequence x^{post}(step 4 and 5). This sequence is then fed into the discriminator $\mathbf{D}$ to predict which tokens in x^{post} were replaced by using the discriminator output y^{disc} (step 6). Afterwards the generator loss $\mathcal{L}_{gen}$ is computed as plain binary cross entropy. The discriminator loss $\mathcal{L}_{disc}$ is calculated as multiclass cross entropy (step 7 and 8). We use a weighted sum of the both with weighting factor λ to obtain our overall per example loss $\mathcal{L}_x$. Finally, we sum over all input sequences $x \in \mathcal{X}$ to receive the final loss $\mathcal{L}$. This loss is then minimized with respect to the parameters $\theta_{\mathbf{G}}$ of the generator and $\theta_{\mathbf{D}}$ of the discriminator.

4.5 Summary

The last sections have covered a variety of approaches to overcome the limits of the attention mechanisms. Starting with adjusted attention mechanisms to create sparsified attention matrices, continuing with reversible residual layers that reduce memory consumption during training, and finishing with a new pretraining paradigm. The next section will adopt these methods in trying to build a small-scale domain-specific LM that can be trained on moderate GPU sizes.

5 Pretraining the model on financial text data

5.1 Introduction

In the following section, the focus will be set to constructing an LM-Encoder model that is pretrained on financial text data under the constraint of limited computational resources. Consequently, the goal is to use the approaches of the previous section, construct different models, and pretrain them on financial text data. During and after the training the models will be evaluated to obtain insights into which model architecture performed best for this domain-specific data and the given computational limits. The repository can be found in the appendix.

Pytorch (Paszke et al. (2019)) and the transformer library by Hugging Face (Wolf et al. (2020)) was used to implement and train the models. The implementation of the reversible blocks and dilated attention was inspired by the Meta research team and the xFormers repository (Lefaudeux et al. (2022)). The code for the Electra framework was highly inspired by Mercier and its Electra repository (MERCIER (2021)).

5.2 Tokenization and Embeddings

For tokenization, we employ the BytePair Encoding (BPE) technique, similar to that used in the classic RoBERTa model (Liu et al. (2019)). To ensure optimal tokenization for our domain-specific dataset, we train a separate tokenizer on the financial text data. This approach ensures enhanced tokenization coverage, crucial for accurately processing the unique terminology and syntax inherent in financial texts.

5.3 Proposed Models

5.3.1 Reversible and Non-Reversible Dilated BERT

In our proposed model, following the classic BERT architecture, we opt to learn word embeddings during training. This strategy allows the model to develop domain-specific word embeddings, potentially capturing nuanced information more relevant to our financial dataset than generic, pre-trained embeddings.

To streamline the learning of contextual representations, we integrate pre-determined positional encodings, specifically sinusoidal encodings as detailed in section 2.5.3 of the original Transformer paper. Our model adheres to BERT's pretraining approach, including the use of segment embeddings S and a class token C (as outlined in section 2.9.1 and Figure 11), which are added to the token embeddings. The final representation of a token T_i and the entire sequence T are formally expressed as:

$$T_i = E_i^{pre} + S_i + PEi \tag{48}$$

$$T = Concat_{col}(C, E^{pre}) + S + PE \tag{49}$$

Our first proposed encoder model features dilated attention blocks, with the first token of the sequence designated as a global token. Each block can be configured as a reversible attention block, with the option to make all or none of the blocks reversible. The memory efficiency of reversible blocks is most pronounced when they are used in sequence, as this setup requires storing only the activations of the last layer. Breaking this chain necessitates additional memory for storing the activations of the new chain, thus incrementally increasing memory usage and complexity. Therefore, the reversibility can either be activated or deactivated for all blocks.

Embedded within the Electra pretraining framework, this model functions both as a generator and discriminator, given its nature as an encoder network. The hyperparameter set H_1 for this model includes:

$$H_1 = \{W, D, r, N^{seq}, N^{heads}, N^{blocks}, d_{model}, G_{size}\}$$

where W is the window size for local attention per layer, D is the dilation rate per layer, $r \in 0, 1$ indicates the reversibility of blocks, N^{seq} is the sequence length, N^{heads} the number of attention heads, N^{blocks} the number of attention blocks, d_{model} the embedding dimension, and $G_{size} \in]0, 1]$ the relative size of the generator compared to the discriminator.

5.3.2 Reformer

The Reformer, introduced in Kitaev, Kaiser, and Levskaya (2020), is a language model that incorporates LSH attention and reversible layers. Like our previous model, the Reformer employs a similar word embedding layer. However, it diverges in its use of axial position embeddings (Dufter, Schmitt, and Schütze (2021)) instead of sinusoidal ones.

The Reformer is also utilized as an encoder within the Electra framework. Its hyperparameter set H_2 is defined as:

$$H_2 = \{b, m, N^{seq}, N^{heads}, N^{blocks}, d_{model}, G_{size}\}$$

Here, b represents the number of hashes, and m the chunk length, with the other parameters being consistent with those in the previous model.

5.3.3 Plain BERT

Additionally, a plain BERT model is used as Electra encoder. There are two main reasons to use BERT here.

- BERT is an established encoder model which has shown very good results in the past
- Employing BERT in a setting with limited computational resources allows us to evaluate whether the modifications in other model architectures significantly impact resource usage during training. This comparison will be particularly insightful in terms of memory consumption, training efficiency, and accuracy under our specific constraints.

Since all the models above inherit their basic architecture from BERT, the hyperparameter set of BERT H_3 is as simple as:

$$H_2 = \{N^{seq}, N^{heads}, N^{blocks}, d_{model}, G_{size}\}$$

This set includes fewer hyperparameters compared to the enhanced models. However, it's important to note that the standard BERT model doesn't specifically address memory limitations. Our primary method for managing these constraints within BERT is to adjust the overall size of the model. Subsequent sections of this thesis will demonstrate that these adjustments can be quite effective, especially in scenarios where sequence lengths aren't excessively long and where the benefits of sparsifying attention mechanisms and reversible layers aren't as pronounced.

5.4 Domain specific data

The primary objective of this training is to evaluate the performance of a model that has been pretrained on domain-specific data, taking into account certain computational limitations. To achieve this, we utilize a dataset of financial texts provided by the source identified as Aman Khan (n.d.). This dataset comprises annual reports filed with the SEC EDGAR system by U.S. public companies from 1993 to 2020. Each report in the dataset is divided into 20 distinct sections, and further, each section is broken down into individual sentences. Altogether, the dataset encompasses approximately 67 million sentences, which serves as the foundation for training the model.

5.5 Computational ressources

For this training, we employ an Nvidia Titan V with 12 GB of VRAM, a choice that brings into sharp focus the challenges and constraints posed by limited memory resources. This hardware selection is particularly pertinent given our domain-specific approach to training models on financial texts. In the realm of specialized datasets, such as financial reports, the efficient utilization of computational resources becomes paramount, especially when operating under memory constraints.

Each model in this study is trained for an identical number of steps, utilizing roughly 12 GB of memory throughout the training process. This consistent methodology is crucial for two reasons. First, it allows for an equitable comparison of training efficiency and basic performance metrics across different models, a comparison that is especially vital in the context of domain-specific data. Second, it aids in a detailed evaluation of each model's effectiveness in masked token prediction, an aspect central to understanding model performance in handling specialized vocabularies and concepts unique to financial texts.

A key emphasis of this study is not merely on the individual performance of each model but also on their comparative performance, particularly in the context of domain-specific training. This comparative analysis is invaluable as it illuminates the trade-offs and design decisions each model architecture makes, especially under the constraints of limited computational resources. By focusing on domain-specific data, we gain a deeper insight into how each model navigates the intricate balance between resource constraints and the specialized nature of the dataset, thereby offering a richer understanding of model efficiency and applicability in specific domains like finance.

5.6 Training setup

5.6.1 General Configuration

In the following sections, we will detail the parameter choices for the models and the general training goal. Since our models are embedded in the Electra approach, we will follow the Electra pretraining paradigm. 15% of the tokens are masked. The generator should predict these, replace the masked tokens with the predictions and the discriminator will predict whether a token was replaced. Both the generator and discriminator loss will be monitored since the interaction of both is needed for a working training process. However, the main goal is to minimize the discriminator loss as the pretrained discriminator is the one that will be used for further downstream tasks. Therefore, a loss weighting factor of $\lambda = 50$ is chosen.

A consistent batch size of 64 was employed across all models, primarily because it represents the maximum capacity for the largest model, BERT, on our GPU. Adopting the same batch size for each model facilitates direct comparisons. The dataset, composed of individual sentences, led to the selection of a sequence length $N^{seq} = 128$. This length is sufficient as all sentences are shorter than 128 tokens, ensuring minimal end-padding and efficient memory utilization. Each model incorporates $N^{blocks} = 12$ attention blocks and, unless specified otherwise, $N^{heads} = 4$ attention heads, a configuration mirroring the small Electra model as proposed by Clark et al. (2020). The generator size was set at $G_{size} = 0.25$. The model-specific parameters are:

- **Reversible Dilated BERT**: This variant was configured with a uniform window size $W = 64, \ldots, 64$ across all twelve layers, accompanied by a consistent dilation rate $D = 2, \ldots, 2$. To optimize memory efficiency, each attention block was made reversible (denoted by $r = 1$). The model's dimension, set to $d_{model} = 256$, aligns with the small Electra model's architecture.
- **Reformer**: For this model, a dimension of $d_{model} = 512$ was chosen. Despite this increased dimension, the Reformer's efficient design ensures that its memory consumption remains comparable to other models. This higher dimension could potentially enhance learning capabilities without incurring additional memory costs. Additionally, the number of hashes was set to $h = 4$, and the chunk length to $m = 64$.
- **BERT**: The BERT model adheres to the general parameter settings, without any unique modifications, unlike the other models.

5.7 Pretraining results

5.7.1 Relative comparison of ressources

We start by giving a quick overview of the ressources each model needed during training. Note that these specifics are based on the training process, i.e. the model and its gradient.

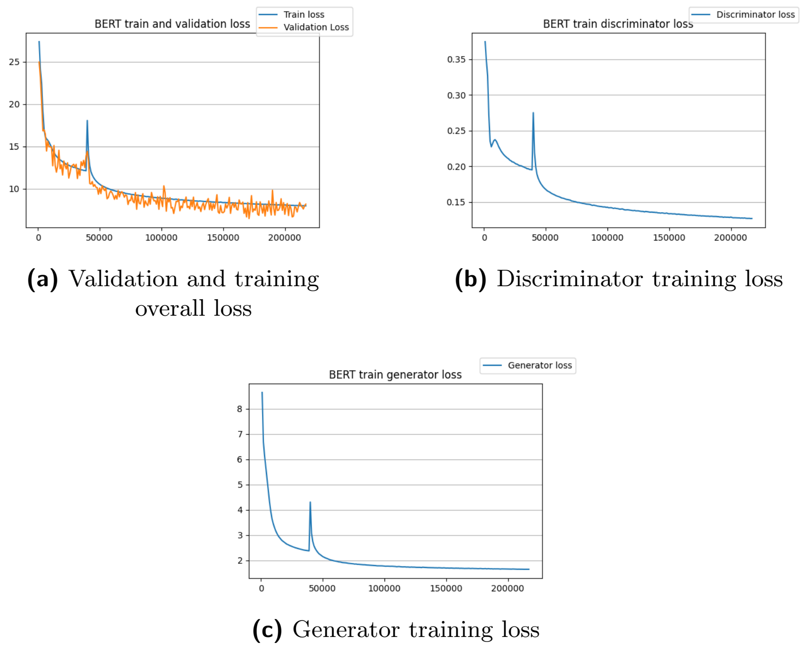
Model	MB memory during training	Seconds per iteration
BERT	8856	1.04
Reformer	8156	1.53
Reversible Dilated BERT	8450	1.14

Table 1: Comparison of model ressources during training

While the memory gains for the Reformer are notable considering a higher model dimension d_{model}, the overall observed memory enhancements fall short of expectations. This discrepancy likely arises because the benefits of the sparsified attention matrix become more pronounced at longer sequence lengths. Similarly, the memory efficiency improvements attributed to the reversible layer are more significant in larger models. Given that this thesis aims to evaluate small-scale models for their suitability in handling domain-specific data, it is crucial to note that despite variations in model architecture among the three examined models, the incremental benefits for small-scale models appear to be minimal.

5.7.2 Relative comparison of pretraining performance

The baseline BERT model outperformed others during the pretraining phase, with both the discriminator and generator exhibiting concurrent learning and reduction in loss. This simultaneous decrease indicates an effective learning process for both components. As depicted in Figure 22, the training process is characterized by loss spikes occurring at comparable intervals, yet the general trend is a decrease in loss over time. Moreover, the training and validation loss display parallel trends, implying that the model is learning in a robust manner.



(a) Validation and training overall loss

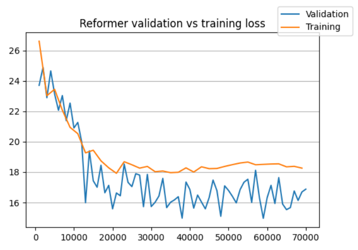
(b) Discriminator training loss

(c) Generator training loss

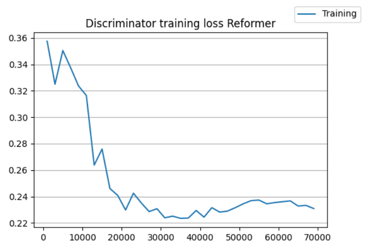
Figure 22: Overall loss, train discriminator and train generator loss

The Reformer exhibited notably slow learning, as illustrated in Figure 23, with minimal decreases in loss per thousand steps. Adjustments to hyperparameters yielded no significant improvements in the model's learning trajectory. The generator's loss decreased until approximately the 10,000th step, after which it began to increase and failed to stabilize in subsequent steps. On the other hand, the discriminator's loss showed a consistent decrease, plateauing around a value of 0.22. When compared to the BERT model's performance, as discussed in the previous section, it is evident that by the 70,000th step, both the generator and discriminator in the BERT model had achieved lower loss values. This is especially surprising given the fact that the model dimension $d_{model} = 512$ is notably larger for the Reformer model than for BERT.

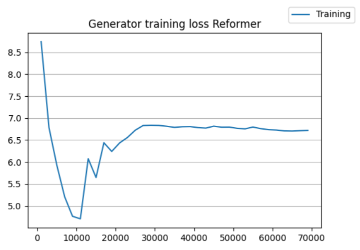
Despite the Reformer's lower memory consumption, it required nearly 50% more time for each iteration compared to the BERT model, as detailed in Table 1. Given the minimal memory savings for this particular sequence length, the Reformer's extended training duration and its sluggish, if any, convergence highlight that the modifications made to its architecture come at the expense of model performance in this specific application context.



(a) Validation and training overall loss



(b) Discriminator training loss

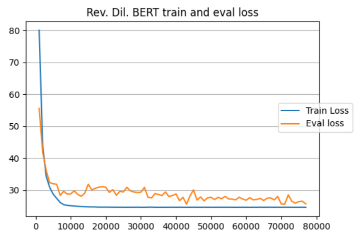


(c) Generator training loss

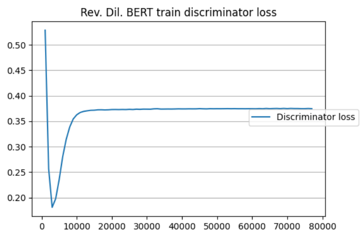
Figure 23: Overall loss, train discriminator and train generator loss

The reversible dilated BERT model demonstrated even poorer performance compared to the Reformer, with both validation and training losses decreasing simultaneously but not significantly. A critical issue was the discriminator's inability to converge, rendering the pretrained model ineffective for any downstream tasks. Additionally, the generator's loss plateaued at approximately 6.5 from step 15,000 to 80,000, indicating a significant stagnation in learning despite the model processing a large number of examples, given its batch size of 64. This suggests that the model had ample opportunity to learn from the data presented.

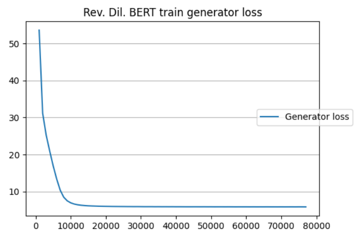
In comparison, the BERT model has proven that effective learning is indeed possible, showcasing better performance metrics. The reversible dilated BERT model consumes slightly more memory than the Reformer but less than the traditional BERT model, positioning it in the middle in terms of resource usage. Its training time also falls between that of the two models. These observations imply that the modifications to the attention mechanism, intended to enhance efficiency, fail to adequately capture the nuances of language for the given sequence length and in this particular domain. This indicates a potential misalignment between the architectural adjustments and the requirements for effective language understanding in this context.



(a) Validation and training overall loss



(b) Discriminator training loss



(c) Generator training loss

Figure 24: Overall loss, train discriminator and train generator loss

In conclusion, it is evident that the BERT model outperforms the competing models in terms of performance. The slight increase in memory usage by BERT is, within the context of this thesis—focusing on small-scale models with limited sequence lengths—insignificant. However, this factor may become more pertinent in scenarios involving larger models and extended sequences. Moreover, BERT's reduced training duration facilitates faster convergence, thereby optimizing resource utilization.

In Table 2 a quick overview of each model's performance is given after 70,000 steps. After this amount of steps, the three models were evaluated and the best was chosen for further pretraining and downstream tasks. It is clear, that the BERT model outperformed the other two models. Hence, for the following downstream tasks, the discriminator of the BERT model is used after being trained on 210,000 steps with a batch size of 64.

Model	Validation Loss	Train loss
BERT	8.99	9.51
Reformer	17.22	18.14
Reversible Dilated BERT	25.60	24.58

Table 2: Comparison of model performance during pretraining after 70,000 steps

6 Finetuning on financial data

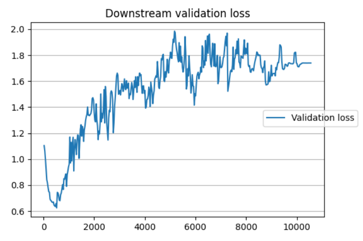
6.1 Financial sentiment analysis

The pretrained model was finetuned on a financial sentiment task. The dataset by Malo et al. (2014) was used for that. It consists of several sentences retrieved from financial news. Each of them is labeled with a sentiment class: neutral, positive, negative. An example sentence would be: *"Biohit already services many current Genesis customers and the customer base is expected to expand as a result of this agreement ."* with the label "positive".

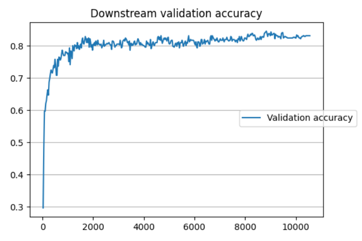
To train the model, the embedding of the [CLS] token in the discriminator was used. A linear layer predicting the logits for each one of three classes was set on top of the [CLS] embedding of the discriminator.

The model underwent fine-tuning for approximately 12,000 steps, achieving a validation accuracy of 84%. A noteworthy observation is that while the training loss stabilized, the validation loss began to diverge from the 500th step onward, indicating potential overfitting. Despite this, there was a simultaneous increase in validation accuracy.

This phenomenon suggests that although the model is accurately identifying the correct classes, it is doing so with diminishing confidence. This scenario exemplifies a common trade-off in the dynamics between loss—specifically, cross-entropy—and accuracy. Cross-entropy gauges the model's confidence in its predictions, whereas accuracy measures the proportion of correctly predicted classes. Given the high accuracy rates and considering the dataset's balance, we can infer that the fine-tuned model is predominantly making correct classifications, albeit with varying degrees of certainty.



(a) Downstream validation loss



(b) Downstream validation accuracy

Figure 25: Financial sentiment finetuning results

6.2 Financial news topic classification

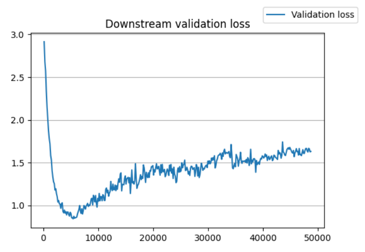
Additionally, the pretrained model was also finetuned to predict the topic of a financial tweet. The "zeroshot/twitter-financial-news-topics" (zeroshot (n.d.)) was used for this task. Possible topics are:

1. Analyst Update
2. Fed, Central Banks
3. Company, Product News
4. Treasuries, Corporate Debt
5. Dividend
6. Earnings
7. Energy, Oil
8. Financials
9. Currencies
10. General News, Opinion
11. Gold, Metals, Materials
12. IPO
13. Legal, Regulation
14. M&A, Investments
15. Macro
16. Markets
17. Politics
18. Personnel Change
19. Stock Commentary
20. Stock Movement

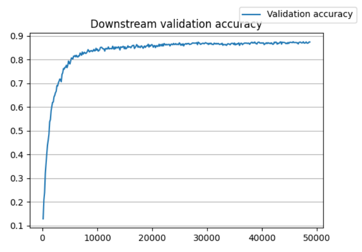
An example sentence would be: *"Netflix and its peers are set for a 'return to growth,' analysts say, giving one stock 120% upside"* with the label "Analyst Update". The classifier architecture was the same as for the finetuning of financial sentiments, but instead of three classes this model outputs 20.

The training process was identical to the previous model for downstream tasks. The [CLS] token was extracted and used for classification.

The finetuned model achieved a validation accuracy of about 87% after being trained for about 500,000 steps. The same phenomenon, even though a bit less severe, of diverging validation loss and increasing validation loss could be observed here, suggesting better, but less confident, predictions.



(a) Downstream validation loss



(b) Downstream validation accuracy

Figure 26: Financial topic finetuning results

6.3 Finetuning conlusion

Leveraging a domain-specific pre-trained model significantly facilitated the training process. As evidenced by Figures 25 and 26, the validation accuracy reached its peak early in the fine-tuning phase, underscoring the robust foundation provided by the pre-trained BERT discriminator. An advantageous aspect of adopting the Electra strategy is that fine-tuning the discriminator requires minimal resources. Unlike other larger pre-trained models, the discriminator constitutes only a part of the comprehensive pretraining framework. Yet, its sophisticated bidirectional context embeddings are readily employable by downstream models. This efficiency indicates that the Electra framework supports the use of smaller models (in our case, $d_{model} = 256$) during both pretraining and fine-tuning phases, with the latter not necessitating the generator. This approach not only streamlines the fine-tuning process but also highlights the Electra model's capacity for efficient adaptation and learning within a resource-constrained environment.

7 Conclusion

This work embarked on an exploratory journey through the intricacies of optimizing transformer models for the nuanced field of financial NLP. Through a detailed examination of tokenization, embeddings, recurrent neural network challenges, and the revolutionary impact of attention mechanisms, this work has laid a comprehensive foundation for understanding the current landscape of NLP technologies. Particularly, it delved into the advancements brought about by transformer models, highlighting their superior capability in handling language-based tasks when compared to traditional RNNs.

The core of this thesis was dedicated to investigating the constraints imposed by hardware limitations on the pretraining and finetuning of transformer models, specifically within the financial NLP domain. By experimenting with various model architectures, including the Reformer, Plain BERT, and a custom reversible dilated BERT, this study sought to uncover efficient strategies for model optimization under resource constraints. The findings reveal that while these adjusted architectures offer theoretical benefits, particularly in scenarios of longer sequence lengths, their practical advantages are somewhat limited in small-scale environments with short sequence lengths and stringent resource limitations for this specific domain.

An intriguing aspect of this research was the successful application of the Electra pretraining approach to the BERT model, which demonstrated considerable efficiency in finetuning with satisfactory outcomes. This suggests that the Electra model, with its discriminator-focused training method, provides a viable path for enhancing model performance in domain-specific tasks without necessitating substantial computational resources.

Furthermore, the exploration of domain-specific pretraining on financial text data underscored the importance of tailored approaches to model development in achieving high accuracy and relevance in specialized fields. The relative comparison of resources and pretraining performance offered critical insights into the trade-offs between computational demands and model efficacy.

In conclusion, this work not only advances our understanding of the potential and limitations of small-scale transformer models in the context of financial NLP but also contributes valuable methodologies and insights for optimizing these models within the constraints of available hardware resources. The comparative analysis of different transformer architectures, alongside the practical application of the Electra pretraining method, provides a solid foundation for future research aimed at refining and enhancing the efficiency of NLP models in domain-specific and ressource-limited applications. As the field of NLP continues to evolve, the findings and approaches detailed in this work will undoubtedly serve as a cornerstone for subsequent investigations into the optimization of transformer models for specialized tasks

Bibliography

Almeida, Felipe and Geraldo Xexéo (2019a). “Word Embeddings: A Survey”. In: *CoRR* abs/1901.09069, pp. 1–2. arXiv: 1901.09069. URL: http://arxiv.org/abs/1901.09069.

– (2019b). “Word Embeddings: A Survey”. In: *CoRR* abs/1901.09069, pp. 3–4. arXiv: 1901.09069. URL: http://arxiv.org/abs/1901.09069.

Alyafeai, Zaid, Maged Saeed AlShaibani, and Irfan Ahmad (2020). “A Survey on Transfer Learning in Natural Language Processing”. In: *CoRR* abs/2007.04239, p. 5. arXiv: 2007.04239. URL: https://arxiv.org/abs/2007.04239.

Andoni, Alexandr et al. (2015). *Practical and Optimal LSH for Angular Distance.* arXiv: 1509.02897 `[cs.DS]`.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2016a). *Neural Machine Translation by Jointly Learning to Align and Translate.* arXiv: 1409.0473 `[cs.CL]`.

– (2016b). *Neural Machine Translation by Jointly Learning to Align and Translate.* arXiv: 1409.0473 `[cs.CL]`.

Beltagy, Iz, Matthew E. Peters, and Arman Cohan (2020). “Longformer: The Long-Document Transformer”. In: *CoRR* abs/2004.05150. arXiv: 2004.05150. URL: https://arxiv.org/abs/2004.05150.

Bengio, Yoshua (2009). “Learning Deep Architectures for AI”. In: *Foundations and Trends® in Machine Learning* 2.1, pp. 1–127. ISSN: 1935-8237. DOI: 10.1561/2200000006. URL: http://dx.doi.org/10.1561/2200000006.

Brown, Tom B. et al. (2020). “Language Models are Few-Shot Learners”. In: *CoRR* abs/2005.14165. arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

Caccia, Massimo et al. (2018). “Language GANs Falling Short”. In: *CoRR* abs/1811.02549. arXiv: 1811.02549. URL: http://arxiv.org/abs/1811.02549.

Chen, Liang-Chieh et al. (2016). “DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs”. In: *CoRR* abs/1606.00915. arXiv: 1606.00915. URL: http://arxiv.org/abs/1606.00915.

Cho, Kyunghyun et al. (2014a). “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *CoRR* abs/1406.1078, p. 1. arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

– (2014b). “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *CoRR* abs/1406.1078, p. 2. arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

Clark, Kevin et al. (2020). *ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators.* arXiv: 2003.10555 `[cs.CL]`.

Devlin, Jacob et al. (2018). “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *CoRR* abs/1810.04805. arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

Dosovitskiy, Alexey et al. (2020). “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *CoRR* abs/2010.11929. arXiv: 2010.11929. URL: https://arxiv.org/abs/2010.11929.

Dufter, Philipp, Martin Schmitt, and Hinrich Schütze (2021). “Position Information in Transformers: An Overview”. In: *CoRR* abs/2102.11090. arXiv: 2102.11090. URL: https://arxiv.org/abs/2102.11090.

Gomez, Aidan N. et al. (2017). “The Reversible Residual Network: Backpropagation Without Storing Activations”. In: *CoRR* abs/1707.04585. arXiv: 1707.04585. URL: http://arxiv.org/abs/1707.04585.

Goodfellow, Ian J. et al. (2014). *Generative Adversarial Networks*. arXiv: 1406.2661 [stat.ML].

Google Developers (2022). *Background: What is a Generative Model?* https://developers.google.com/machine-learning/gan/generative. Accessed: 2024-01-16.

He, Kaiming et al. (2015). “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

Hu, Yuhuang et al. (2018). “Overcoming the vanishing gradient problem in plain recurrent networks”. In: *CoRR* abs/1801.06105, p. 1. arXiv: 1801.06105. URL: http://arxiv.org/abs/1801.06105.

Keskar, Nitish Shirish et al. (2019). “CTRL: A Conditional Transformer Language Model for Controllable Generation”. In: *CoRR* abs/1909.05858. arXiv: 1909.05858. URL: http://arxiv.org/abs/1909.05858.

Khan, Aman (n.d.). *Financial Reports SEC*. URL: https://huggingface.co/datasets/JanosAudran/financial-reports-sec.

Khan, Anwar and Boreom Lee (2021). *Gene Transformer: Transformers for the Gene Expression-based Classification of Lung Cancer Subtypes*. arXiv: 2108.11833 [q-bio.QM].

Kitaev, Nikita, Lukasz Kaiser, and Anselm Levskaya (2020). “Reformer: The Efficient Transformer”. In: *CoRR* abs/2001.04451. arXiv: 2001.04451. URL: https://arxiv.org/abs/2001.04451.

Lan, Zhenzhong et al. (2019). “ALBERT: A Lite BERT for Self-supervised Learning of Language Representations”. In: *CoRR* abs/1909.11942. arXiv: 1909.11942. URL: http://arxiv.org/abs/1909.11942.

Lefaudeux, Benjamin et al. (2022). *xFormers: A modular and hackable Transformer modelling library*. https://github.com/facebookresearch/xformers.

Liu, Yinhan et al. (2019). “RoBERTa: A Robustly Optimized BERT Pretraining Approach”. In: *CoRR* abs/1907.11692. arXiv: 1907.11692. URL: http://arxiv.org/abs/1907.11692.

Malo, P. et al. (2014). “Good debt or bad debt: Detecting semantic orientations in economic texts”. In: *Journal of the Association for Information Science and Technology* 65.

Malte, Aditya and Pratik Ratadiya (2019a). “Evolution of transfer learning in natural language processing”. In: *CoRR* abs/1910.07370. arXiv: 1910.07370. URL: http://arxiv.org/abs/1910.07370.
– (2019b). “Evolution of transfer learning in natural language processing”. In: *CoRR* abs/1910.07370. arXiv: 1910.07370. URL: http://arxiv.org/abs/1910.07370.
MERCIER, François (2021). *Efficient transfer learning for NLP with ELECTRA*. URL: https://openreview.net/forum?id=Or5sv1Pj6od.
Mielke, Sabrina J. et al. (2021). “Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP”. In: *CoRR* abs/2112.10508. arXiv: 2112.10508. URL: https://arxiv.org/abs/2112.10508.
Ng, Andrew and Michael Jordan (2001). “On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes”. In: *Advances in Neural Information Processing Systems*. Ed. by T. Dietterich, S. Becker, and Z. Ghahramani. Vol. 14. MIT Press. URL: https://proceedings.neurips.cc/paper_files/paper/2001/file/7b7a53e239400a13bd6be6c91c4f6c4e-Paper.pdf.
Pascanu, Razvan, Tomás Mikolov, and Yoshua Bengio (2012a). “Understanding the exploding gradient problem”. In: *CoRR* abs/1211.5063, p. 1. arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.
– (2012b). “Understanding the exploding gradient problem”. In: *CoRR* abs/1211.5063, p. 2. arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.
Paszke, Adam et al. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. arXiv: 1912.01703 [cs.LG].
Qiu, Jiezhong et al. (2019). “Blockwise Self-Attention for Long Document Understanding”. In: *CoRR* abs/1911.02972. arXiv: 1911.02972. URL: http://arxiv.org/abs/1911.02972.
Raffel, Colin et al. (2019). “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *CoRR* abs/1910.10683. arXiv: 1910.10683. URL: http://arxiv.org/abs/1910.10683.
Rosendahl, Jan et al. (2019). “Analysis of Positional Encodings for Neural Machine Translation”. In: *International Workshop on Spoken Language Translation*, p. 2. URL: https://api.semanticscholar.org/CorpusID:211483460.
Sanh, Victor et al. (2019). “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter”. In: *CoRR* abs/1910.01108. arXiv: 1910.01108. URL: http://arxiv.org/abs/1910.01108.
Song, Xinying et al. (2020). “Linear-Time WordPiece Tokenization”. In: *CoRR* abs/2012.15524. arXiv: 2012.15524. URL: https://arxiv.org/abs/2012.15524.
Tay, Yi et al. (2020). “Efficient Transformers: A Survey”. In: *CoRR* abs/2009.06732. arXiv: 2009.06732. URL: https://arxiv.org/abs/2009.06732.
Vaswani, Ashish et al. (2017a). “Attention Is All You Need”. In: *CoRR* abs/1706.03762, pp. 2–3. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.
– (2017b). “Attention Is All You Need”. In: *CoRR* abs/1706.03762, p. 1. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.
– (2017c). “Attention Is All You Need”. In: *CoRR* abs/1706.03762, p. 6. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

Vaswani, Ashish et al. (2017d). “Attention Is All You Need”. In: *CoRR* abs/1706.03762, p. 4. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

Vennerød, Christian Bakke, Adrian Kjærran, and Erling Stray Bugge (2021). “Long Short-term Memory RNN”. In: *CoRR* abs/2105.06756, p. 2. arXiv: 2105.06756. URL: https://arxiv.org/abs/2105.06756.

Verma, Prateek and Jonathan Berger (2021). “Audio Transformers: Transformer Architectures For Large Scale Audio Understanding. Adieu Convolutions”. In: *CoRR* abs/2105.00335. arXiv: 2105.00335. URL: https://arxiv.org/abs/2105.00335.

Wang, Sinong et al. (2020). “Linformer: Self-Attention with Linear Complexity”. In: *CoRR* abs/2006.04768. arXiv: 2006.04768. URL: https://arxiv.org/abs/2006.04768.

Wolf, Thomas et al. (2020). *HuggingFace’s Transformers: State-of-the-art Natural Language Processing.* arXiv: 1910.03771 [cs.CL].

Wu, Yonghui et al. (2016). “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”. In: *CoRR* abs/1609.08144. arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144.

Zaheer, Manzil et al. (2020). “Big Bird: Transformers for Longer Sequences”. In: *CoRR* abs/2007.14062. arXiv: 2007.14062. URL: https://arxiv.org/abs/2007.14062.

zeroshot (n.d.). *zeroshot/twitter-financial-news-topic · Datasets at Hugging Face — huggingface.co.* https://huggingface.co/datasets/zeroshot/twitter-financial-news-topic. [Accessed 30-01-2024].

A Repository of the implementation

The repository for this thesis can be found here: https://github.com/Coluding/Assessing-Efficiency-in-Domain-Specific-Transformer-Models.git