BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Compressing BERT – An Evaluation and Combination of Methods

Master Thesis of

Sebastian Jäger

Matriculation Number: 886469

Beuth University of Applied Sciences Berlin
Fachbereich VI - Informatik und Medien
Data Science and Text-based Information Systems (DATEXIS)

Reviewer:	Prof. Dr.-Ing. habil. Alexander Löser
Second reviewer:	Prof. Dr. rer. nat. Felix Bießmann
Advisor:	Betty van Aken

08.04.2020 – 10.09.2020

Beuth Hochschule für Technik Berlin
Fachbereich VI - Informatik und Medien
Luxemburger Straße 10
13353 Berlin

Abstract

Using pre-trained Language Models (LMs), such as ELMo [54], BERT [19], or GPT models [55, 56, 7], show good empirical results on a wide range of Natural Language Processing (NLP) downstream tasks. However, these large models consist of millions or even billions of parameters, making training and inference slow and computationally expensive, especially in resource-constrained environments. In this thesis, we discuss and compare four approaches to compress BERT based on literature research and choose *Theseus Compression (TC)* as most promising for further experimental evaluation. We introduce and show the benefits of changing TC's initialization procedure and present a comprehensive analysis of its hyperparameters. A concluding qualitative error analysis reveals that TC efficiently compresses the original model's knowledge into a smaller model. For a comprehensive evaluation of TC's performance, we use three datasets of two domains, hate speech and medical domain, for two binary-classification downstream tasks, German hate speech detection and in-hospital mortality prediction. Our best experiments show that domain-specific models compressed with Theseus Compression are 1.67× smaller, train downstream tasks more than 2× faster, retain up to 99% prediction performance, and increase inference speed, on average, 1.94× on CPU, and 1.73× on GPU.

Contents

List of Figures

List of Tables

List of Abbreviations

AUC	Area Under the Curve
BERT	Bidirectional Encoder Representations from Transformers
Bi-RNN	Bidirectional RNN
ELMo	Embeddings from Language Models
GLUE	General Language Understanding Evaluation
KD	Knowledge Distillation
LM	Language Model
MIMIC-III	Medical Information Mart for Intensive Care
ML	Machine Learning
MLM	Masked Language Model
NLP	Natural Language Processing
NN	Neural Network
NOHATE	Overcoming crises in public communication about refugees, migration, foreigners
NSP	Next Sentence Prediction
PKD	Patient Knowledge Distillation
QA	Question Answering
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
RPP	Reweighted Proximal Pruning
SMT	Statistical Machine Translation
SOTA	State-of-the-art
TC	Theseus Compression

1 Introduction

Natural Language Processing (NLP) is a widely studied research area and powers a broad set of applications, such as translation systems, voice assistants, or chatbots. In recent years, the rise of transfer learning, which means using pre-trained Language Models (LMs), increased the development cycle speed of NLP systems. Besides this, the usage of large-scale pre-trained LMs, such as ELMo [54], BERT [19], or GPT models [55, 56, 7], also significantly improved the performance in almost all sub-areas of NLP. On the other hand, this poses the problem that these models consist of millions or even billions of parameters, making them computationally expensive and memory inefficient. Further, the model sizes make it difficult to apply them on real-world applications, where conditions, e.g., maximum inference time, need to be met.

For this reason, in the past few years, *model compression* gained attention. It aims to reduce the number of parameters without or with as little performance loss as possible [8]. To underline the importance of this research field, a brief overview of NLP's history follows.

Rule-based natural language processing Research in NLP began in the 1950s when Alan Turing proposed the famous *Turing Test* [72] as a criterion of intelligence [17]. In the very early years, NLP research was dominated by the creation of dictionaries and research on the syntax of language [36]. At that time, NLP systems were based on hand-written rules, like SHRDLU [76], published in 1971. However, rule-based applications' disadvantages are that rules rely on expert knowledge, which is expensive, and need to be continuously adjusted.

Statistical natural language processing Later on, the availability of machine-readable text grew rapidly, reinforced by the rise of the Internet [36]. Starting in the early 1980s [36], instead of using fixed rules, *statistical NLP* "comprises all quantitative approaches to automated language processing, including probabilistic modeling, information theory, and linear algebra. While probability theory is the foundation for formal statistical reasoning [...] encompassing all quantitative approaches to data" [47]. This means that statistical models are trained based on the automated analysis of large text corpora.

A well-studied research area is, e.g., *Statistical Machine Translation (SMT)*, which powered systems such as Google Translate[1] that was launched in 2006[2].

[1]Link: `https://translate.google.com`

[2]Source: `https://ai.googleblog.com/2006/04/statistical-machine-translation-live.html` - Accessed July 09, 2020

Deep learning-based natural language processing Bengio et al. [6] proposed the first Neural Language Model that computes *word embeddings*. A *LM* tries to compute the probability of a word w_t given the previous words, i.e., $P(w_t|w_1, ..., w_{t-1})$, and, in this context, word embeddings are vector representations of the input in a lower-dimensional space [48]. Although Bengio et al. [6] laid the foundation for modern models in 2000, Neural Networks (NNs) have only been applied to real-world applications from 2010 [17]. As an example, Google first experimented in 2014 with Recurrent Neural Networks (RNNs) [69] for their translation system Google Translate and finally switched from the previous SMT to Neural Machine Translation in 2016[3].

Pre-trained language models In 2015, Dai and Le [13] proposed to reuse pre-trained LMs for downstream tasks and presented this technique's strong performance. Since then, the research community developed several advanced model architectures that further improved the model's performance. The most important architectures for this thesis are described in more detail in Section 2.1.

In recent years, the release of the model *Embeddings from Language Models (ELMo)* [54] started a trend toward even vaster general-purpose LMs (see Figure 1.1), followed by *GPT* [55] and finally, the State-of-the-art (SOTA) NLP model *Bidirectional Encoder Representations from Transformers (BERT)*, which is described in more detail in Section 2.1.4. These models boosted the performance on all sub-tasks of the General Language Understanding Evaluation (GLUE)-benchmark [74], a benchmark specifically designed for models that share linguistic knowledge across downstream tasks.

Nevertheless, the steadily increasing model sizes lead to the downside that these models are difficult to apply on real-world applications. This directly leads to this thesis' motivation for investigating model compression, described in detail in the following section.

1.1 Motivation

Using LMs is SOTA and show good empirical results on a wide range of downstream tasks [54, 55, 19]. Besides the increase in performance, this has the advantage that the computational expensive unsupervised pre-training process does not have to be repeated [19]. Moreover, compared to pre-training, only a downstream task-specific labeled dataset and relatively weak hardware and time constraints are necessary for fine-tuning.

For example, Devlin et al. [19] stated that pre-training of the basic configuration of BERT with 110M parameters was performed on "4 Cloud TPUs in Pod configuration (16 TPU chips total) [and] took 4 days to complete"[4]. However, the training of the GLUE downstream tasks took "at most 1 hour on a single Cloud TPU, or a few hours on a GPU" [19].

[3]Source: `https://ai.googleblog.com/2016/09/a-neural-network-for-machine.html` - Accessed July 09, 2020

[4]Tensor Processing Units (TPUs) are Google's highly specialized chips for deep learning. Source: `https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning` - Accessed July 10, 2020

The trend toward vaster models The trend shown in Figure 1.1 goes clearly toward bigger model architectures. In the last month, much vaster models have been released, e.g.,

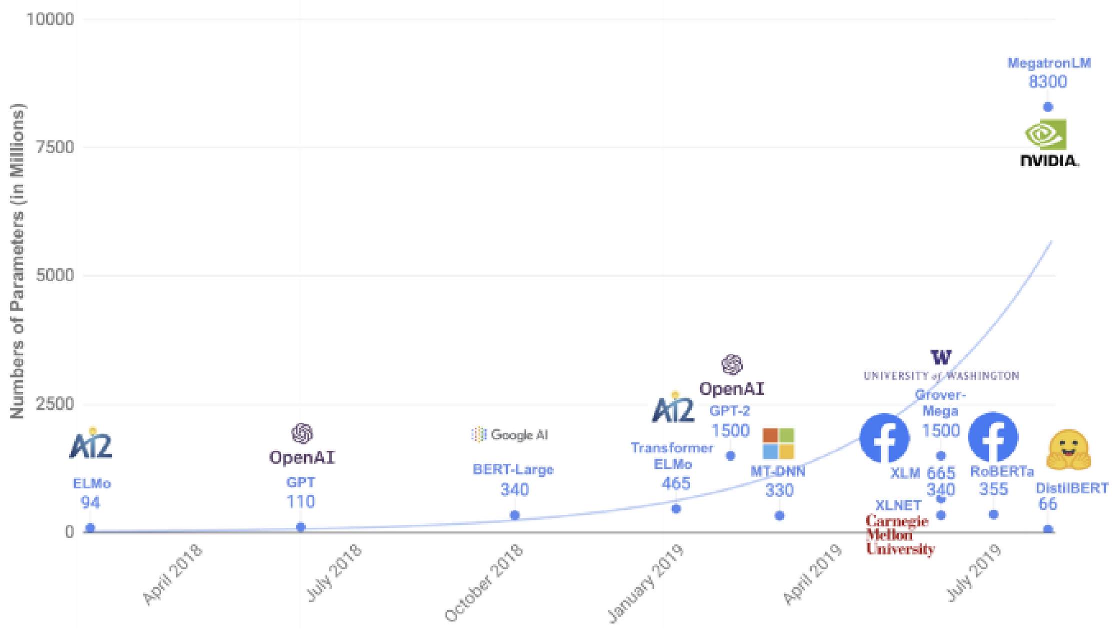


Figure 1.1: Development of released Natural Language Processing model sizes over time. [60]

Conditional Transformer Language (CTRL) with 1.63B parameters, T5 with 11B parameters [57], and GPT-3 with 175B parameters [7]. The increasing number of parameters, of course, also affects the training time of these models. Table 1.1 presents the training time of some selected models based on the evaluation of Strubell et al. [65]. Note that the models' training uses different hardware. The chip's theoretical performance is shown in the column *TeraFLOPS*[5].

Model	Params	Hardware	TeraFLOPS	Hours (Total)	~ Hours (1 Chip)
ELMo	94M	P100x3	5.3[6]	336	1008
$\text{BERT}_{\text{base}}$	110M	TPUv2x16	180[7]	96	1536
GPT-2	1.5B	TPUv3x32	420[8]	168	5376

Table 1.1: Pre-training time of Language Models. Even with more advanced hardware, the pre-training time of vaster Language Models explodes. [65]

[5]TeraFLOPS means trillion floating-point operations per second

[6]Source: `http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-datasheet.pdf` - Accessed July 12, 2020

[7]Source: `https://cloud.google.com/tpu/` - Accessed July 12, 2020

[8]Source: `https://cloud.google.com/tpu/` - Accessed July 12, 2020

Reasons for compression Besides the increasing pre-training times for these LMs, this trend raises additional concerns. First, huge LMs need long inference times [34], especially for deployments on weaker or non-specialized hardware such as CPUs. Second, longer inference times consequentially lead to higher training times for downstream tasks. Third, deploying such models to low resource environments, e.g., web applications or even edge-devices is impossible due to the computational and memory constraints [60].

Combining those disadvantages of vast models with the finding that trained models consist of redundant parameters [18, 15, 22] results in increasing effort of *model compression*, which tries to reduce the number of parameters without or with as little performance loss as possible [8]. The starting point for BERT compression represents the release of the with *knowledge distillation* (see Section 2.2.3) compressed model *DistilBERT* that is small enough to run on edge-devices [60].

From another perspective, during research, it is often necessary to perform a high number of experiments. Using smaller models that potentially need shorter training times helps to iterate more quickly. Therefore, a compressed BERT is valuable for supporting research. On the other hand, applying BERT without fine-tuning its language model does not always lead to satisfying performance. As an example, Lee et al. [41] stated that "pre-training BERT on biomedical corpora is crucial in applying it to the biomedical domain". This raises the question, if model compression can help to train domain-specific compressed BERT that supports fast experiment iterations, which is the central research objective of this thesis.

1.2 Aim of Research and Methodology Outline

This thesis's main objective is to evaluate whether the usage of a domain-specific compressed BERT helps to speed up experiment iterations. For this, we discuss and compare existing compression approaches and choose one that will be used for further experimental evaluation. Therefore, we use two binary classification downstream tasks in two domains, the *medical* and *hate speech* domain. We choose these for two reasons: first, both domains are not considered in the GLUE-benchmark and consist of very different texts, and second, the DATEXIS[9] research group in which this work was carried out works, among others, in these areas.

Methodology outline We start by defining the downstream tasks *hate speech detection* and *in-hospital mortality prediction* and the choice of corresponding datasets (see Sections 3.2 and 3.3). The evaluation metrics are two folded and refer to the model's prediction performance and the performance of the compression process itself (see Section 3.4).

The benchmark framework we use to evaluate this thesis's main objective experimentally is as follows. To maximize the comparability, we start our evaluation with the implementation and training of baseline models (see Section 3.5.1). We then experiment with Theseus Compression (TC)'s initialization procedure (see Section 3.5.2) and use the most promising for further hyperparameter analysis (see Section 3.5.3). Moreover, based

[9]Research group "Data Science and Text-based Information Systems (DATEXIS)" at Beuth University of Applied Sciences Berlin. Link: `https://prof.beuth-hochschule.de/loeser/`

on their prediction performance, the best models are used for an in-depth evaluation of TC.

1.3 Thesis Structure

This thesis is structured into six chapters as follows. Chapter 2 lays the theoretical foundations. It describes the model BERT and presents different approaches for its compression. Subsequently, it compares the compression approaches and chooses the most promising for further experimental evaluation. Chapter 3 introduces our methodology, which gives an overview of the downstream tasks, datasets, metrics, benchmark framework, and infrastructural environment. Chapter 4 presents our implementations in detail, while Chapter 5 shows our experiments, evaluates and discusses them. In the end, Chapter 6 concludes and gives an outlook of possible future work.

2 Theoretical Foundations

This chapter lays the theoretical foundations for this thesis. We start with the description of several developments that finally led to the model BERT. All of these milestones built up on each other and increased the performance of models at that time.

In the second part, we introduce several compression approaches for BERT and discuss them based on related work. For this, we outline their advantages and disadvantages, as well as show their results.

In the end, we define categories to compare the compression approaches. We use the presented papers as proxies and, finally, choose Theseus Compression as most promising for further experimental evaluation.

2.1 Major Milestones toward Model BERT

In this section, we present the developments that led to BERT and introduce the mathematical notations for this thesis. Starting with the encoder-decoder architecture, several improvements increased its performance and enabled the SOTA Transformer architecture. BERT's authors [19] resemble this architecture, introduce two pre-training procedures, and describe an easy-to-use framework.

2.1.1 Encoder-Decoder Model and Attention Mechanism

In 2014, Cho et al. [11] proposed a novel model architecture for SMT, the *RNN Encoder-Decoder*, that outperforms former models. It first encodes an input sequence into a context vector and uses this intermediate representation to generate an output sequence. This idea still powers the most recent SOTA models [19, 54, 56], even though they were developed further.

Encoder The *encoder* of the by Cho et al. [11] proposed model architecture, illustrated in Figure 2.1, receives the input sequence $\mathbf{x} = (x_1, ..., x_n)$ step-by-step and compresses the information of the first word into a fixed-length hidden state vector $\overline{\mathbf{h_1}}$ of arbitrary size. Subsequently, each following step takes the last hidden state $\overline{\mathbf{h_{i-1}}}$ and the respective input token x_i into account to compute the hidden state:

$$\overline{\mathbf{h_i}} = f(\overline{\mathbf{h_{i-1}}}, x_i) \tag{2.1}$$

where f is a non-linear activation function. By applying Equation 2.1 n times, the encoder computes a fixed-length context vector $\mathbf{c} = \overline{\mathbf{h_n}}$ that captures the entire input sequence.

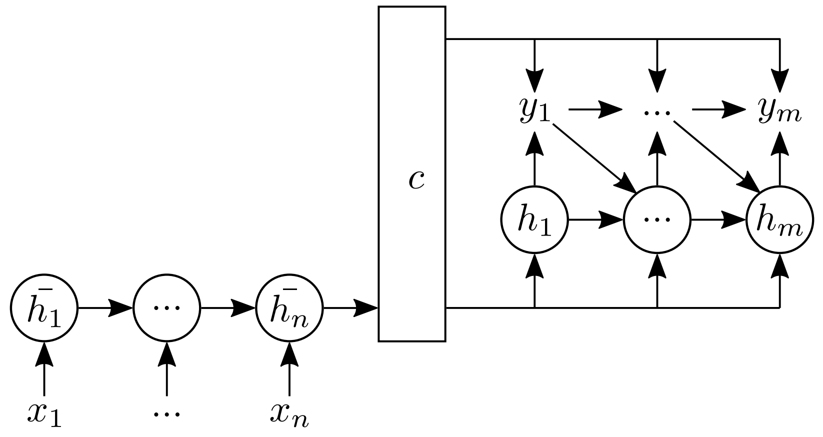


Figure 2.1: Encoder-Decoder model. The decoder uses for each step the fixed-length hidden state c, the previous decoder hidden state h_{i-1}, and the previous output y_{i-1} to compute the output y_i. [11]

Decoder The *decoder* generates a new output sequence. For each decoding step $i \in [1, m]$, where m is the output sequence length, the decoder computes a decoder hidden state $\mathbf{h_i}$ and output $\mathbf{y_i}$. Therefore, it takes the previous decoder hidden state $\mathbf{h_{i-1}}$, the previous output $\mathbf{y_{i-1}}$, and the context vector $\mathbf{c}$ into account:

$$\mathbf{h_i} = f(\mathbf{h_{i-1}}, \mathbf{y_{i-1}}, \mathbf{c}) \tag{2.2}$$

Finally, it feeds the hidden state $\mathbf{y_i}$ into another activation function g, e.g., the softmax function:

$$\mathbf{y_i} = g(\mathbf{h_i}) \tag{2.3}$$

This produces a probability distribution over the set of possible output tokens, i.e., in the context of SMT, the vocabulary of the target language. [11]

Attention mechanism The *Attention Mechanism* improves the encoder-decoder architecture and tackles the shortcomings of the single point of communication in the form of a fixed-length context vector. This approach instead provides a separate context vector for each decoder step and gives the decoder the ability to learn to pay attention to the input sequence's important parts. [5]

Bahdanau, Cho, and Bengio [5] proposed to use Bidirectional RNNs (Bi-RNNs), which consist of two RNNs, a forward and backward RNN. The forward RNN computes the *forward hidden states* $(\overrightarrow{h_1}, ..., \overrightarrow{h_n})$ based on $\overrightarrow{x} = (x_1, ..., x_n)$. The backward RNN computes the *backward hidden states* $(\overleftarrow{h_1}, ..., \overleftarrow{h_n})$ based on $\overleftarrow{x} = (x_n, ..., x_1)$, correspondingly. An annotation for input token x_i that depends on all surrounding words in the input sequence is then obtained by concatenating the forward and backward hidden states:

$$\overline{h_i} = [\overrightarrow{h_i}; \overleftarrow{h_i}] \tag{2.4}$$

The usage of several context vectors changes the definition of the decoder hidden states to

$$\mathbf{h_i} = f(\mathbf{h_{i-1}}, \mathbf{y_{i-1}}, \mathbf{c_i}) \tag{2.5}$$

where the context vector $\mathbf{c_i}$ is computed as a weighted sum of the encoder hidden states $\overline{h_i}$ and a simultaneously learned feedforward neural network. As the authors stated: "Intuitively, this implements a mechanism of attention in the decoder. The decoder decides parts of the source sentence to pay attention to" [5].

Since the model's decoder can learn to pay attention to specific parts of the entire encoded sequence, it is also known as *global attention* [46]. However, this global attention can not be used for the encoder step since it obliges extra input. Therefore, the question arises if the approach of attention can also be applied to the encoder stage of such models, the answer was introduced by Lin et al. [43] and is called: *Self-attention Mechanism*.

2.1.2 Self-attention Mechanism

Encoder-decoder models increased the performance of NLP models and influenced the development of most recent SOTA models. The attention mechanism tackled the shortcoming of a single point of communication between the encoder and decoder step in form of a fixed-length context vector. However, the approach of global attention is restricted to the decoder step. Another type of attention, the *self-attention*, addresses this issue.

Lin et al. [43] proposed a self-attentive model for sentence embeddings that overcomes this shortcoming by letting the sequence pay attention to its context and weights the relevance of the respective parts of the input.

The objective "is to encode a variable-length sentence into a fixed size embedding" [43]. Let $\mathbf{x} = (x_1, ..., x_n)$ be a sequence of word tokens. First, Lin et al. [43] computed the hidden states consistent with Equation 2.4 with a Bi-RNN with u hidden units. Hence, the matrix $H = (h_1, h_2, ..., h_n)$ containing n hidden states: $H \in \mathbb{R}^{n \times 2u}$

$$a = softmax(\mathbf{w_{s2}} tanh(W_{s1} H^\top)) \tag{2.6}$$

$W_{s1} \in \mathbb{R}^{d_a \times 2u}$ is a learned matrix and $\mathbf{w_{s2}}$ a learned vector of size d_a, where d_a is a hyper-parameter and can be arbitrarily chosen. Finally, the attention context vector $\mathbf{c}$ is the sum of scaled hidden states and gets calculated as

$$c = \mathbf{a}^T H \tag{2.7}$$

Multi-hop Attention Further, Lin et al. [43] introduced a *multi-hop attention* to capture different parts of the input sentence. To calculate r representations, the authors extended $\mathbf{w_{s2}}$ into a matrix $W_{s2} \in \mathbb{R}^{r \times d_a}$. The corresponding attention scores vector gets matrix A:

$$A = softmax(W_{s2} tanh(W_{s1} H^\top)) \tag{2.8}$$

and the attention context vector gets a matrix C correspondingly

$$C = AH \tag{2.9}$$

Penalization term The authors [43] found that this attention context matrix C often focus on similar or the same aspects multiple times. Thus, they introduced a penalty term P

that punishes this redundancy and forces the attention context vectors to focus on single aspects. P is defined as

$$P = ||(AA^\top - I)||_F^2 \tag{2.10}$$

where $||\bullet||_F^2$ is the squared Frobenius norm of a matrix.

2.1.3 Transformer Model

The attention mechanism allows the decoder to focus on the important parts of the input sequence and, therefore, overcomes the shortcomings of a fixed-length context vector. Further, self-attention addresses the lack of benefits of attention in the encoder step. The usage of self-attention in the encoder and attention in the decoder successfully increase the performance of models based on RNNs. However, the sequential nature of these models reduces the parallelization possibilities and memory efficiency, especially during training [73]. The *Transformer* architecture was developed to overcome these shortcomings by abandon the usage of RNNs.

Figure 2.2 shows the architecture overview of the Transformer model proposed by Vaswani et al. [73]. Both the encoder and decoder are composed of stacks of identical layers, which are very similar and powered by *Multi-Head Attention* that is described in Section 2.1.3.1.

Encoder The left-hand side of Figure 2.2 depicts the encoder stacks that consist of two sub-layers: A form of self-attention (see Section 2.1.3.1) based on multi-head attention (*Multi-Head Attention*, orange) and a position-wise fully connected feed-forward network (*Feed Forward*, blue). Each of the two sub-layers is surrounded by a residual connection [29] and followed by layer normalization (*Add & Norm*, yellow) [4]. The former helps to build deeper models by letting the model learn to skip layers that would harm its performance; the latter turns out to improve training time and generalization performance. Based on [73, 42], the position-wise feed-forward network can be seen as responsible for learning cross-attention-head interactions. The layer's output is input for the next one in the stack; the top-most output is input for the decoder, correspondingly. Therefore, outputs of all layers need to be of identical dimension, defined by the hyperparameter d_{model} that can be arbitrarily chosen. [73]

Decoder The right-hand side of Figure 2.2 shows the decoder stacks. First, it applies self-attention (*Masked Multi-Head Attention, orange*) on the already generated output sequence $(y_1, ..., y_{i-1})$. Therefore, it masks the future context $(y_i, ..., y_m)$, where m is the output length. Second, based on the encoder's output, the decoder incorporates the concept of global attention (*Multi-Head Attention*, orange). Finally, it uses a position-wise fully connected feed-forward network (*Feed Forward*, blue). Identically to the encoder, all three sub-layers use residual connections and layer normalization and feed its output to the next layer in the stack where the output of the top-most layer is the decoder's output. This is fed to a linear and softmax layer to compute the model's next output token y_i. [73]

For both stages encode and decode, the sequences are first of all embedded (*Input Embedding*, red), and annotated with a positional encoding. [73]

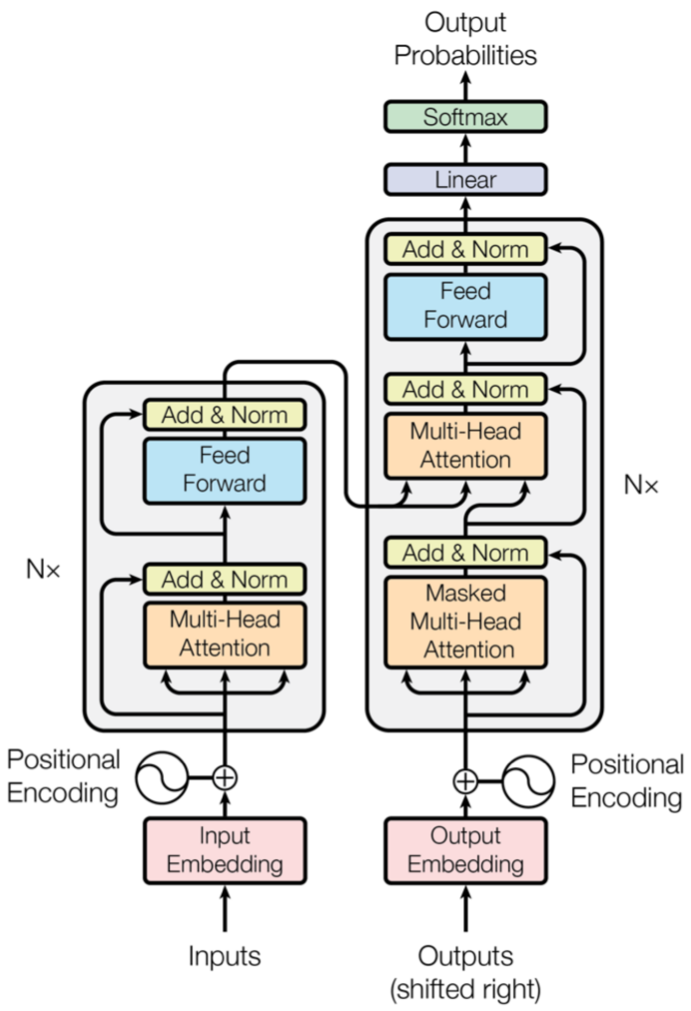


Figure 2.2: Transformer architecture overview. On the left N-stacked encoder layers, and on the right N-stacked decoder layers. [73]

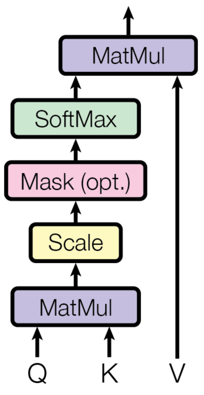
2.1.3.1 RNN-less Attention Mechanisms

Vaswani et al. [73] developed the concept of attention further to abandon RNNs. The authors described the attention mechanism as "mapping a query and a set of key-value pairs to an output". Their proposed *Scaled Dot-Product Attention* is similar to Equation 2.8. However, they introduced a scaling factor $\frac{1}{\sqrt{d_k}}$. Similar to Section 2.1.2, the attention is applied simultaneously on a set of query Q, keys K, and values V. Therefore, the vectors extend to matrices:

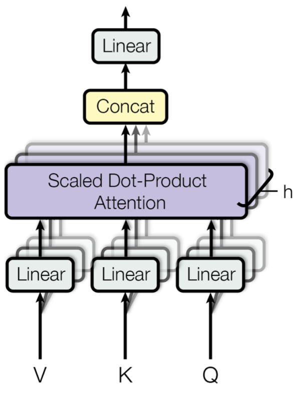
$$Attention(Q, K, V) = softmax(\frac{QK^\top}{\sqrt{d_k}})V \tag{2.11}$$

The computation graph of this is shown in Figure 2.3a.

Multi-head attention Another improvement in their type of attention is the idea of using *multi-head attention.* Vaswani et al. [73] found that learning h linear projections for queries, keys and values to d_q, d_k and d_v dimensions increase the performance of the



(a) Scaled Dot-Product Attention.



(b) Multi-Head Attention.

Figure 2.3: Attention mechanisms in the Transformer model. [73]

model, where h is a hyperparameter and can be arbitrarily chosen. By using smaller dimensions $d_k = d_v = \frac{d_{model}}{h}$ for each attention head, the overall computational complexity stays similar. Each head applies the attention function in parallel. Their results then get concatenated and again linearly projected, resulting in the final output shown in Figure 2.3b.

$$MultiHeadAttention(Q, K, V) = [head_1, ..., head_h]W^O \tag{2.12}$$

Where the projection matrix $W^O \in \mathbb{R}^{hd_v \times d_{model}}$ are learned parameters. The computation for each $head_i$ extends, correspondingly, to:

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{2.13}$$

Where the projections are also learned parameters: $W_i^Q \in \mathbb{R}^{d_{model} \times d_q}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$ and $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$.

As shown in Figure 2.2, multi-head attention is used in three different ways [73]:

Encoder Self-attention

Keys, values, and queries come from the output of the previous layer. Therefore, each position in the encoder can pay attention to all positions in the previous layer.

Decoder Self-attention

Similar to the encoder self-attention, each position in the decoder can pay attention to all positions in the decoder up to and including that position. Therefore, the scaled dot-product attention masking out (setting to $-\infty$), see Figure 2.3a.

Global Attention

Queries are from the previous decoder layer, keys and values come from the decoder's output. Therefore, every position in the decoder can pay attention to all positions of the input sequence.

2.1.3.2 Positional Encoding

The Transformer model does not need RNNs to apply attention; however, it loses the position of input sequences [73]. Therefore, the authors introduced *Positional Encodings* to include position information to word embeddings.

Vaswani et al. [73] discussed and experimented with different choices of positional encodings, *learned* and *fixed*. They decided to choose fixed sinusoidal functions due to the hypothesis "it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training" and nearly identical results during their experiments.

The used positional encoding functions are [73]:

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{\frac{2i}{d_{model}}}}) \tag{2.14}$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}}) \tag{2.15}$$

where *pos* is the absolute position of the input and *i* the functions' dimension. The resulting positional encodings have the same length as the input embeddings, so that they can be summed.

A wide range of models are transformer-based, e.g., GPT models [55, 56, 7], or Transformer-XL [14]. However, the most well known is BERT, which is described in the following section.

2.1.4 Bidirectional Encoder Representations from Transformers (BERT)

Using LMs show good empirical results on many NLP tasks [55, 32, 53]. ELMo showed that building bidirectional LMs significantly improves their performance [54]. However, it shallowly concatenates the forward and backward context on input sequences. In contrast to this, GPT model [55] leverages the more advanced transformer decoder architecture but neglects future context. For this reason, Devlin et al. [19] introduced BERT to overcome the shortcomings of a unidirectional LM.

Language Model Manning, Raghavan, and Schütze [48] defined that a Language Model (LM) tries to compute the probability of a word w_t given its context, e.g., $P(w_t|w_1, ..., w_{t-1})$. A *unidirectional* LM only incorporates previous tokens as context to predict the following word. Consistently, a *bidirectional* LM uses all available tokens for prediction.

Cloze procedure Building a bidirectional LM based on transformer models entails challenges. The multi-layered transformer model has to ensure that the input token does not see itself. Otherwise, predictions are trivial because the target word is part of the input. Therefore, Devlin et al. [19] proposed *Masked Language Model (MLM)* that is inspired by the idea of *cloze procedure* [71]. Instead of using the preceding words for prediction of the succeeding one, MLM masks certain words (targets) and uses the remaining for prediction. 15% of the input tokens are dedicated as targets. If the i-th token is chosen, the following rules are applied for replacements:

- 80% replace it with the token `[MASK]`:
  `I am going to miss being a student -> I am going to miss being a [MASK]`
- 10% replace it with a *random* word:
  `I am going to miss being a student -> I am going to miss being a cat`
- 10% keep it unchanged:
  `I am going to miss being a student -> I am going to miss being a student`

Next Sentence Prediction For specific tasks, e.g., Natural Language Inference and Question Answering (QA), the models need to understand relationships between sentences. For this, Devlin et al. [19] introduced Next Sentence Prediction (NSP). The objective of this task is to predict whether a sentence B follows A. Specifically, 50% of the time B is the actual next sentence, and 50% is a randomly selected sentence. With the combination of both MLM and NSP, the model learns a more comprehensive language understanding.

Input representation BERT can handle a variety of downstream tasks to make it applicable for transfer learning. For this, it uses WordPiece embeddings and a collection of metadata [19]. WordPiece embeddings reduce the vocabulary size and simultaneously increase the performance for rare words by splitting them into common sub-word units (see tokens of the word *"playing"* in Figure 2.4) [77]. Let $A = (\text{"}My\text{"}, \text{"}dog\text{"}, \text{"}is\text{"}, \text{"}cute\text{"})$, and $B = (\text{"}He\text{"}, \text{"}likes\text{"}, \text{"}playing\text{"})$ be sentences of the input X. First of all, the model adds the unique tokens $[CLS]$ and $[SEP]$ to the input sequence, see Figure 2.4. $[CLS]$ is always the first token, the final hidden state corresponding to this token then represents the whole aggregated sequence. The $[SEP]$ token separates the input sentences. The augmented input is word-piece tokenized [77] to create the *Token Embeddings*. Besides the usual *Position Embeddings* of the transformer model, BERT adds *Segment Embeddings* that represent whether the token corresponds to sentence A or B. These three representations of the input are finally summed up and used as the input representation for BERT. [19]

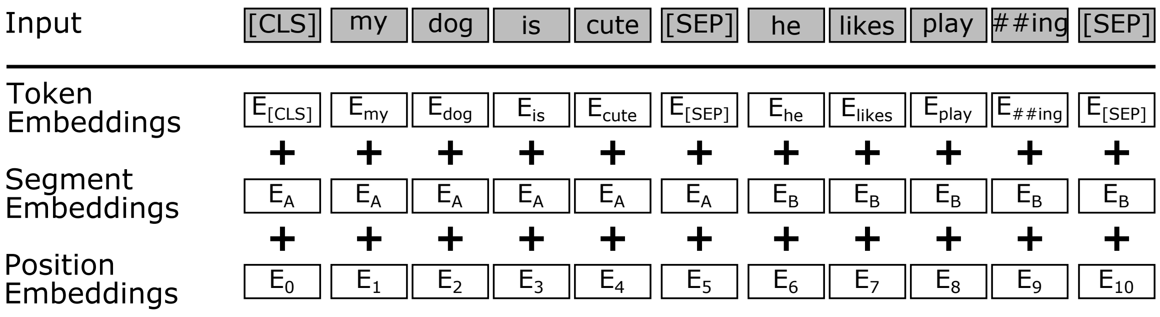

Input
[CLS]
my
dog
is
cute
[SEP]
he
likes
play
##ing
[SEP]
Token Embeddings
E[CLS] Emy Edog Eis Ecute E[SEP] Ehe Elikes Eplay E##ing E[SEP]
Segment Embeddings
EA EA EA EA EA EA EB EB EB EB EB
Position Embeddings
E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 E10

Figure 2.4: Input representation of BERT. The input embeddings are the sums of token embeddings, segment embeddings, and position embeddings. [19]

In summary, Devlin et al. [19] introduced a simple framework that consists of *pre-training* and *fine-tuning* a model. They focused on usability and made sure that BERT can easily be adapted for several downstream tasks. Most of the time, adding a single layer is sensible to reuse BERT's knowledge and fine-tune the model regarding a specific task. The authors performed the computationally expensive part[1], the pre-training, and released two model sizes:

BERT$_{\text{base}}$
: $L = 12$ transformer layers, $h = 12$ attention heads, and $d_{model} = 768$ hidden size. Total parameters: 110M

BERT$_{\text{large}}$
: $L = 24$ transformer layers, $h = 16$ attention heads, and $d_{model} = 1024$ hidden size. Total parameters: 340M

Using a high number of parameters to achieve good model performance almost always comes with the drawback of increasing memory consumption and prediction time. In this thesis, we focus on compressing $BERT_{base}$. The following section describes the concept of model compression and approaches to compress BERT.

2.2 Compression Approaches for BERT

Early research in the field of model compression compressed ensemble models into a single model [8]. Later on, the usage of SOTA feature extractors, like VGG for Computer Vision [63] and BERT for NLP [19], and transfer learning became popular. This shifts the focus to compress single models and reduce the size of pre-trained large-scale generic models, e.g., [60, 34, 25, 80, 62].

Definition and terminology In general, model compression aims to reduce the number of parameters without or with as little performance loss as possible [8]. Nevertheless, the reasons for model compression are manifold. For example, reducing the model's size for deployment on edge devices, increasing the model's speed to reach real-time requirements, or decreasing the training time based on a compressed model. Throughout this thesis, we use terminology as follows.

Model Performance or Prediction Performance
: Refers to the model's prediction performance on the trained downstream task. Common metrics are *accuracy*, *F1*, or *ROC/AUC*.

Model Size
: The storage used to save the model's parameters to disc. It is often specified in MB. The smaller the better.

Model Speed and Prediction or Inference Time
: Model Speed can be viewed as $\frac{1}{prediction\ time}$. The model's prediction time for one example or a batch of examples is specified in seconds (s) or milliseconds (ms), and the smaller the better. Thus, higher model speed is better.

[1]"Training of $BERT_{base}$ was performed on 4 Cloud TPUs in Pod configuration (16 TPU chips total)" [19]

Model Training Speed
The time until the model performance is maximized. The smaller the better.

Model Memory Consumption
The necessary main memory used for predictions. The smaller the better.

The following sections describe the most important compression approaches for BERT.

2.2.1 Quantization

Quantization in general means the reduction of precision. In the context of compressing BERT, this refers to the representation of the model's weights [23]. Usually, models are using 32-bit floating numbers for the parameter representations because modern hardware cannot handle high precision number calculations with less than 32 bits [75]. Truncating each 32-bit weight to target bitwidth, e.g., 16-bit reduces the model's size in this case by a factor of two. This very naive *static* approach often produces a sizable performance drop [23].

Quantization aware training In contrast to the static approach, quantization aware training needs additional training steps to adjust the quantized weights. However, it can, therefore, retain more of the model's performance. A more advanced technique also incorporates the layer's sensitivity regarding quantization and uses *mixed-precision quantization* to adapt the target bitwidth. [62, 80]

2.2.2 Pruning

Pruning is based on the fact that models are typically over-parameterized, and after training, consist of significant redundancy [18]. Pruning identifies and removes the redundant or less important weights or parts [23]. This sometimes increases the model's performance and makes it more robust [23]. Ganesh et al. [23] divide pruning approaches applied on BERT into two categories: *Elementwise Pruning* and *Structured Pruning*.

Elementwise pruning The goal of elementwise pruning, also known as *sparse pruning*, is omitting individual weights. For this, it uses measurements to determine the weight's importance and, finally, defines a set of least important weights. Possible importance measurements are, e.g., based on their absolute values or gradients. [23]

Structured pruning In contrast to elementwise pruning, structured pruning focuses on removing architectural components. For BERT, deleting attention heads or even whole encoder layers are the most common approaches. Since the original $BERT_{base}$ consists of $L = 12$ layers with each $h = 12$ attention heads (see Section 2.1.4), removing redundant parts holds great potential for compression. [23]

2.2.3 Knowledge Distillation

Knowledge Distillation (KD) "is a compression technique in which a compact model – the *student* – is trained to reproduce the behaviour of a larger model – the *teacher* – or an ensemble of models" [60]. For this, it introduces one or more additional L_{distil} distillation loss functions used to transfer BERTs language model capabilities. Ganesh et al. [23] distinguished between three categories of KD: *distillation on logits output*, *distillation on encoder outputs*, and *distillation on attention maps*.

Distillation on logits output The simplest KD scenario treats the distillation as an end-to-end task. An additional loss function L_{distil} enables the student to take the teacher's output into account, i.e., the student can directly learn from the teacher's logits. For this, Hinton, Vinyals, and Dean [31] introduced a *temperature* T hyperparameter to the softmax function that controls to which extent the student relies on the teacher. The adjusted *softmax* is computed as

$$softmax(z_i, T) = \frac{exp(\frac{z_i}{T})}{\sum_j exp(\frac{z_j}{T})} \tag{2.16}$$

where z_i is the model score for class i. The student's training objective is a linear combination of the distillation loss L_{distil} with the supervised training loss.

For distillation on logits output, the student does not need to be a smaller version of the teacher. Often, this type of KD is used to transfer the teacher's knowledge into another simpler model architecture [70, 44].

Distillation on encoder outputs In contrast to quantization, pruning, and distillation on logits output, distillation on encoder outputs is not model agnostic. As the name suggests, it creates a smaller BERT by either reducing the number of layers L, the number of attention heads h, or both. [23]

More concrete, Jiao et al. [34] defined L_{distil} to reduce h as follows:

$$L_{distil} = L_{hidden} = MSE(\mathbf{H}^S \mathbf{W}_h, \mathbf{H}^T) \tag{2.17}$$

where the matrices $\mathbf{H}^S \in \mathbb{R}^{n \times d_{student}}$ and $\mathbf{H}^T \in \mathbb{R}^{n \times d_{teacher}}$ refer to the student's and teacher's hidden states and n denotes the input's length. Usually, one chooses the $d_{student} < d_{teacher}$ to obtain a smaller student model. Matrix $\mathbf{W}_h \in \mathbb{R}^{d_{student} \times d_{teacher}}$ are learned parameters to transform the student's hidden states into the same space as teacher network's hidden states. For compression, L_{hidden} is applied on several layers. One challenge is that the student can not learn from the teacher's output, which can be tackled by simultaneously reducing the number of layers L [23].

Reducing L forces the student's layers to learn from a sequence of teacher's layers. For this, a student model with $M < N$ layers is used, where student layers $m \in \{1, ..., M\}$ and teacher layers $n \in \{1, ..., N\}$, and Equation 2.17 with $d_{student} = d_{teacher}$ is applied. This encourages the student layer to learn from all teacher layers $< n$. [23, 34]

Distillation on attention maps The before mentioned distillation approaches can be directly applied to reduce model size. Distillation on attention maps is used besides. Like Equation 2.17, Jiao et al. [34] defined L_{attn} as:

$$L_{distil} = L_{attn} = \frac{1}{h}\sum_{i=1}^{h} MSE(head_i^S, head_i^T) \tag{2.18}$$

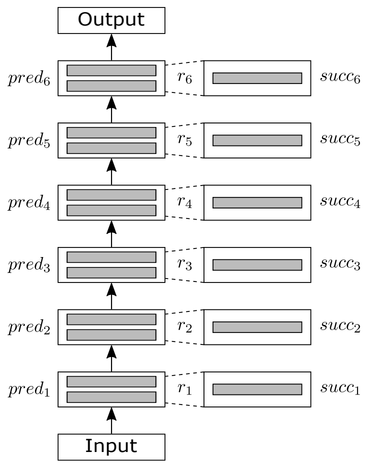
This means the *mean squared errors* of the h attention heads.

2.2.4 Theseus Compression

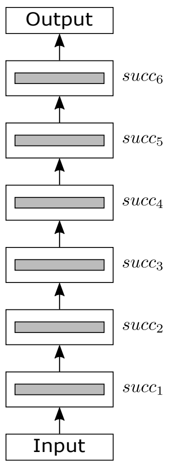
TC is the latest of the before mentioned compression approaches. It was proposed by Xu et al. [78] in 2020 and is "inspired by the famous thought experiment 'Ship of Theseus' in Philosophy, where all components of a ship are gradually replaced by new ones until no original component exists". The main idea is the progressive replacement of parts of the original BERT model with modules of fewer parameters. Like KD, TC encourages the compressed model – the *successor* – to behave like the original model – the *predecessor*.

Module replacing Xu et al. [78] divided the predecessor model P into n *predecessor modules* and defined for each a *successor module*. Thus, $P = \{pred_1, ..., pred_n\}$ and $S = \{succ_1, ..., succ_n\}$ where $pred_i$ and $succ_i$ denote the predecessor modules and their corresponding successor modules, respectively. This defines the single constraint of TC: $succ_i$ must have same input and output sizes as $pred_i$.

$BERT_{base}$ with $L = 12$ layers (see Section 2.1.4) as the predecessor model serves as continuous example. Further, $pred_i$ consists of two and $succ_i$ of one transformer layer, Figure 2.5a shows this setting. The initialization of the successor model S corresponds to the first six layers of the predecessor model. The output of the i-th module is denoted as



(a) Module Replacement.



(b) Successor Fine-tuning and Inference.

Figure 2.5: The two phases of Theseus Compression. (a) Module Replacement and (b) Successor Fine-tuning. [78]

a_i, then the forward step of one layer can be described in the form of:

$$a_i = pred_i(a_{i-1}) \tag{2.19}$$

During compression, TC replaces $pred_i$ with $succ_i$ by a chance of the *replacement rate* an independent Bernoulli random variable

$$r_i \sim Bernoulli(p) \tag{2.20}$$

which has probability p to be 1, and $(1 - p)$ to be 0. Thus, Equation 2.19 can be adapted to

$$a_i = r_i * succ_i(a_{i-1}) + (1 - r_i) * pred_i(a_{i-1}) \tag{2.21}$$

where $*$ denotes the element-wise multiplication. During training, predecessor modules' weights are frozen, and only the successor modules' weights are updated. For this, TC only uses the task-specific loss function, e.g., Cross Entropy.

Successor fine-tuning In the second phase of TC, all successor modules are getting collected and used for *successor fine-tuning*. This phase is equal to use replacement rate $p = 1$ (see Figure 2.6a). Since each $succ_i$ is smaller than $pred_i$ in size, the predecessor model P is compressed into a smaller model S. [78]

Curriculum replacement Moreover, Xu et al. [78] highlighted the improvements in using a dynamic replacement rate p_d instead of the constant replacement rate p. The *curriculum replacement* starts with a low value for p_d that increases linearly throughout training (see Figure 2.6b). Therefore, in the early stages of the training process, the model has more guidance from the predecessor modules. Later on, the successor model gradually learns to perform without help of predecessor modules. The dynamic replacement rate p_d is defined as

$$p_d = min(1, ks + b) \tag{2.22}$$

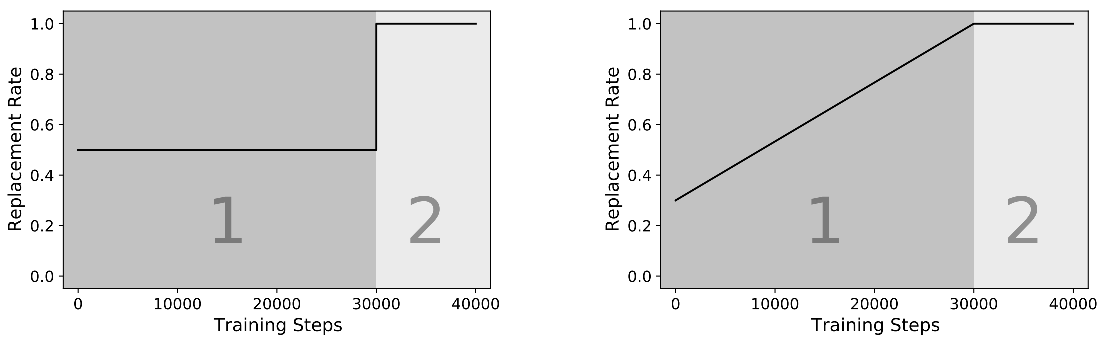
where s is the compression step, b is the starting replacement rate, and $k > 0$ is the coefficient which controls the replacement rate changes. Figure 2.6 shows the two replacement strategies throughout the training process. Where the constant strategy strictly divides the two TC phases, curriculum replacement uses a much smoother transition.

2.2.5 Summary of Selected Related Work

This section gives an overview of related work for the compression approaches described in Sections 2.2.1 to 2.2.4. It summarizes the most important insights and results for the comparison in Section 2.3.

2.2.5.1 Quantization

As a reminder, quantization truncates the bitwidth of weights representation (see Section 2.2.1). Although many research focuses on model quantization [12, 20, 33, 81], only a few papers apply quantization to BERT.



(a) Constant Replacement. (b) Curriculum Replacement.

Figure 2.6: Replacement Strategies. Constant replacement (a) with a hard transition from module replacement phase into the successor fine-tuning phase and curriculum replacement (b) with a smooth transition between these phases. [78]

Shen et al. [62] presented 13× compression by losing at most 2.3% prediction performance. They applied mixed-precision quantization based on layer-wise analysis with Hessian information to find less sensitive parameters for quantization. Further, the authors proposed *group-wise quantization* that partitions the weight matrices into different groups. Each group then uses its own quantization scheme. Group-wise quantization helps to reduce performance degradation. However, it increases hardware complexity.

Zafrir et al. [80] compared post-training quantization and quantization aware training on the GLUE-benchmark. Using a target bitwidth of 8-bit, they achieved a 4× compression by adding less than 1% relative error, excluding the task *recognizing textual entailment* which is an outlier. Implementation is available[2].

Both have in common that they compress the models during fine-tuning for downstream tasks.

2.2.5.2 Pruning

In the context of compressing BERT, pruning received much more attention than quantization. Following Section 2.2.2, it is possible to prune specific weights, known as elementwise pruning, or remove attention heads or complete layers called structured pruning.

Elementwise pruning Guo et al. [28] proposed a new pruning method specifically designed for large-scale LMs: Reweighted Proximal Pruning (RPP). It finds a sparse weight pruned BERT with respect to specified downstream tasks. The authors presented that RPP achieves 59.3% sparsity without losing prediction performance on downstream tasks.

Gordon et al. [26] applied magnitude weight pruning during pre-training BERT. Magnitude weight pruning removes weights close to zero. They stated that pruning 30-40% does not decrease the prediction performance for downstream tasks. Implementation is available[3].

[2] `https://github.com/NervanaSystems/nlp-architect`
[3] `https://github.com/mitchellgordon95/bert-prune`

Structured pruning Michel, Levy, and Neubig [50] iteratively pruned attention heads based on the Taylor expansion method. They showed that it is possible to prune up to 20-40% without reducing the model's prediction performance on the downstream task. However, pruning was applied during fine-tuning and its efficiency is depending on the downstream task and dataset. Further, GPU-based[4] experiments with a 50% pruned BERT achieved 17.5% speedup with batch size bigger than 16. For smaller batches, the speedup effect decreases. Implementation is available[5].

Kovaleva et al. [39] confirmed that not all heads are equally important. They even showed that disabling parts of BERT could increase its performance for downstream tasks. The best case shows 3.2% absolute improvement by removing a whole layer. However, this is cumbersome because testing permutations of $\text{BERT}_{\text{base}}$'s 144 attention heads is time-consuming.

Fan, Grave, and Joulin [21] introduced *LayerDrop*, a regularization technique that skips complete layers during training time. They applied LayerDrop on pre-training from scratch and showed that it is possible to select shallow sub-networks during inference time without fine-tuning them. However, they presented their results only on RoBERTa [45], a BERT-like model.

2.2.5.3 Knowledge Distilling

KD aims to transfer knowledge of a teacher model into a student with fewer parameters. The student must not necessarily follow the teacher's network architecture (see Section 2.2.3).

Sun et al. [67] used distillation on encoder outputs on several intermediate layers. They called their solution Patient Knowledge Distillation (PKD) and compressed directly regarding specific downstream tasks. Their baseline implementation is very similar to [60]; however, less efficient since applied during fine-tuning. Implementation is available[6].

Sanh et al. [60] applied KD's initial idea proposed by [31] and used distillation on logits output. They distilled pre-trained $\text{BERT}_{\text{base}}$ on the same corpus as it was originally trained, into a general-purpose 6-layered BERT structure and named it *DistilBERT*. The distilling process took 90 hours on 8 GPUs[7]. For better comparison, this is similar to train 720 hours on a single GPU. DistilBERT retains 97% of language understanding capabilities measured on GLUE-benchmark, although the model is 1.67× smaller and 1.67× faster. Implementation is available[8].

Jiao et al. [34] introduced two-stage KD and released the compressed model under the name *TinyBERT*. The first stage is similar to DistilBERT. A general-purpose TinyBERT is trained that performs worse than BERT because of the 4-layered structure and reduced the hidden size to $d_{model} = 312$. During the second stage, TinyBERT learns from every third layer and gets fined-tuned based on an augmented task-specific dataset. TinyBERT

[4]Used GPU: NVIDIA GeForce GTX 1080Ti

[5]`https://github.com/pmichel31415/are-16-heads-really-better-than-1`

[6]`https://github.com/intersun/PKD-for-BERT-Model-Compression`

[7]Used GPUs: NVIDIA Tesla 16GB V100

[8]`https://github.com/huggingface/transformers/tree/master/examples/distillation`

achieves 96% of BERT$_{\text{base}}$'s performance on GLUE-benchmark while being 9.4× faster and 7.5× smaller. Implementation is available[9].

We would like to point out that there is more research on KD, e.g., [68, 82, 9, 10, 70]. However, those use additional techniques, like weight sharing or changed model architectures, which are not part of this thesis.

2.2.5.4 Theseus Compression

As a reminder, TC uses module replacement to perform compression. It progressively substitutes predecessor modules with successor modules with fewer parameters (see Section 2.2.4).

Xu et al. [78] used the GLUE-benchmark to compare their result with KD baselines. Like DistilBERT, they used BERT$_{\text{base}}$ as predecessor and replaced two BERT layers by one, resulting in a 1.67× smaller successor model. Although the authors compressed during downstream task training, the compressed model retains more than 98% performance on GLUE of the predecessor, outperforming DistilBERT. Further, TC does not need computationally expensive pre-training, fine-tuning the GLUE downstream tasks took at most 20 GPU[10] hours. Implementation is available[11].

2.3 Comparison of Compression Approaches

We aim to support fast experiment iterations for future research, as pointed out in Section 1.2. This results in constraints for the compression approach. In this section, we compare different approaches based on the papers discussed in Section 2.2.5. Subsequently, we choose the most promising approach that will be analyzed and evaluated in more detail in the following chapters.

Categories for comparison A first category to compare the compressing approaches with each other is whether they need specialized hardware or not. This is of importance because we do not have access to any specialized hardware, which can handle very low bit precision efficiently, and want to come up with an appropriate result for a broad range of researchers. Second, for fast experiment iterations, the compression outcome has to be robust against different downstream tasks. A general-purpose BERT is preferable since it is not appropriate to compress a model for each experiment. Thus, we analyze whether the compression is applied during pre-training or fine-tuning and discuss possible generalization possibilities later on. Further, the compression efficiency is another important consideration. This incorporates the compression ratio, retained performance, computational costs of the compression process, and if it is a manual or automated process. Lastly, publicly available source code is preferable.

[9] `https://github.com/huawei-noah/Pretrained-Language-Model/tree/master/TinyBERT`

[10] Used GPUs: NVIDIA Tesla 16GB V100

[11] `https://github.com/JetRunner/BERT-of-Theseus`

Comparison of related work To allow a better comparison, Table 2.1 presents the related work papers from Section 2.2.5. It summarizes the most important aspects and classifies them into the described comparison categories.

Category / Paper	Comp. Ratio	GLUE	Time	PT	FT	Auto	HW	Code
Q-BERT [62]	13×				✗	✗	✗	
Q8BERT [80]	4×				✗	✗	✗	✗
RPP [28]	1.68×	100%			✗	✗		
Gordon et al. [25]	1.43 − 1.67×	100%		✗		✗		✗
Michel et al. [50]	1.25 − 1.67×	100%			✗	✗		✗
Kovaleva et al. [39]		>100%			✗			
Fan et al. [21]				✗				
PKD [67]	1.67×				✗	✗		✗
DistilBERT [60]	1.67×	97%	720h	✗		✗		✗
TinyBERT [34]	7.5×	96%		✗	✗	✗		✗
Bert-of-Theseus [78]	1.67×	98%	20h		✗	✗		✗

Table 2.1: Comparison of the discussed papers of Section 2.2.5. Column description: *Comp. Ratio*: the stated compression ratio; *GLUE*: the retained GLUE score compared to original model; *Time*: the reported time necessary for compression process with one GPU; *PT*: if the outcome is general-purpose compressed BERT; *FT*: if the outcome is a fine-tuned BERT; *Auto*: if the compression process is automated; *HW*: if specialized hardware is necessary; *Code*: if the implementation is available, links are embedded.

The two quantization techniques Q-BERT [62] and Q8BERT [80], can achieve impressive compression ratios with only losing minimal performance. However, both have to be deployed on specialized hardware, which can handle very low bit precision, to efficiently run these compressed models, which means that quantization will not be considered.

It is a disadvantage to rely on a non-automated process such as [39, 21]. Therefore, those will be excluded. Further, [28, 50] compressed during fine-tuning and stated that the performance and resulting compressed BERT heavily depends on the downstream task and dataset. The most promising pruning technique [26] uses elementwise pruning, which has the disadvantage that it creates irregular sparsity in the model. Yao et al. [79] demonstrated that random sparsity is difficult to speed up on GPUs and, therefore, it is questionable if [26] increases the compressed model's speed, which is our main objective.

Looking at KD, DistilBERT [60] outperforms PKD [67] and compresses a general-purpose BERT. Due to the fact that the power of TinyBERT [34] results from its two-staged approach, the intermediate general-purpose BERT is much worse than the final result (see Section 2.2.5.3). As we are interested in a general-purpose BERT, only DistilBERT and BERT-of-Theseus remain for further considerations.

Choosing BERT-of-Theseus Both DistilBERT [60] and BERT-of-Theseus [78] use BERT$_{\text{base}}$ for compression, achieve a compression ratio of 1.67×, are fully automated processes, do not need specialized hardware, and published their implementations. BERT-of-Theseus states a slightly higher retained GLUE score. However, DistilBERT has the huge advantage of creating a general-purpose compressed BERT. Further, the process to train DistilBERT takes 36× longer on a single GPU than BERT-of-Theseus.

The combination of retaining more GLUE score performance and the fast compression processes, finally, leads to the decision to evaluate Theseus Compression (TC) in greater detail.

2.4 Summary

This chapter starts with an overview of the major development steps that led to BERT. It reassembles the transformer encoder stacks, which implement encoder self-attention without RNNs, and uses the cloze procedure to build a bidirectional LM. BERT implements cloze procedure as the task MLM, which hides words at random, and BERT learns to predict these hidden words. Combined with another pre-training method, NSP, it achieves SOTA performance on a wide range of downstream tasks, e.g., the GLUE-benchmark. Pre-trained BERT models are easy-to-use for transfer learning, which is why they were quickly adopted to a wide range of applications.

However, BERT's usability and high performance come with the downside that it is a fairly huge model. In the second part of this chapter, we describe four approaches to compress BERT, where the main goal is to reduce the number of parameters without or with little as possible performance loss. The concepts of these approaches are very different:

- Quantization reduces the bitwidth
- Pruning removes not necessary or less important parts
- KD trains a student model to mimic the teacher based on intermediate loss functions
- TC trains a successor model by progressively replacing the predecessor modules

Moreover, we summarize related work for these approaches and present their results.

This chapter closes with a comparison of the compression approaches for the presented related work. Therefore, we define categories for the comparison and, finally, choose TC as most promising for our work because:

- It does not need specialized hardware that can handle very low bit precision
- The compression process is fully automated

- Its retained GLUE score is higher than DistilBERT
- Its compression process is 36× faster than DistilBERT
- Its code is available: `https://github.com/JetRunner/BERT-of-Theseus`

The following Chapter 3 presents the methodology on which we conduct our experimental evaluation.

3 Methodology

This chapter presents our methodology for the experimental evaluation of Theseus Compression. At the beginning, we lay down our hypothesis that defines the direction of this thesis.

We then describe the domains, hate speech, and medical domain of our experiments in more detail and define the binary classification downstream tasks, hate speech detection, and in-hospital mortality prediction we use for the evaluation. As an implication of these domains and the chosen downstream tasks, we present the datasets for our experiments. To be precise, we use three labeled datasets, two for hate speech detection and one for in-hospital mortality prediction. The fourth unlabeled dataset is used for LM fine-tuning for the hate speech domain. Subsequently, we define metrics on two dimensions for the evaluation: model performance and compression performance.

The end of this chapter gives a brief overview of the benchmark framework we use to evaluate TC and closes with a presentation of our infrastructural setup.

3.1 Hypothesis

We aim to increase the iteration speed of experiments with a compressed BERT (see Section 1.2). Therefore, our hypothesis is: *Using a domain-specific BERT compressed with Theseus Compression helps to speed up experiment iterations for downstream task models.* This potentially holds the advantages that fine-tuned downstream task models are smaller in size, can be deployed on weaker hardware, and are faster during inference.

More detailed, we adapt TC so that the compression process uses the pre-training task MLM. Thus, we fine-tune the model's LM and compress it simultaneously.

3.2 Definition of Downstream Tasks

To examine our hypothesis (see Section 3.1), we focus on downstream tasks in two domains: *medical* and *hate speech*. To minimize the complexity of our experimental evaluation, we use solely binary classification tasks for both domains.

Binary classification In general, classification is a supervised Machine Learning (ML) task, whose outcome consists of a discrete variable. Specifically, the outcome of binary classification has exactly two possibilities. Therefore, one observation belongs either to class A or to class B.

The following sections give a brief introduction of the medical and hate speech domain and define the downstream tasks.

3.2.1 Medical Domain

In healthcare, ML is not as widely used as in other areas, such as society or business [16]. Davenport and Kalakota [16] stated that ethical implications, like the model's interpretability or biased systems, and the lack of data availability are reasons for the slow adoption of ML to the medical domain. However, the widespread use of Electronic Health Records or at least basic digital systems in the USA in the last years [35] enabled the creation of structured databases [35, 59]. Further, [16] showed that in 2019 ML was used to support tasks, such as analyzing X-ray images or engaging people to live a healthier lifestyle based on personal data from smartphones and smartwatches. They postulated that ML systems "will not replace human clinicians on a large scale, but rather will augment their efforts to care for patient[s]" [16]. As an example, systems for Clinical Decision Support (CDS) aim to increase the quality of health outcomes by providing clinicians with additional information based on patient-specific data.

Downstream task: in-hospital mortality prediction One specific task of CDS is the *in-hospital mortality prediction.* This task predicts whether a patient dies during a hospital stay. This helps to provide better medical care to prevent their death. Early systems, such as MPM$_0$-II proposed by [30], used statistical models on a wide range of measured vital indicators. More recent approaches combine these features with NLP-based clinical texts analysis, e.g., admission letters [49].

We solely classify using admission letters whether a patient died within the given hospitalization.

3.2.2 Hate Speech Domain

Social media platforms like Facebook[1], Twitter[2], or Youtube[3] allow almost everyone to easily participate in online discussions. However, those discussions often follow certain patterns and do not seldom end up in the usage of insulting and offensive language [51]. The massive amount of online communication via social media makes it no longer possible to moderate discussions without technical support [66].

Downstream Task: hate speech detection Following [61], there are three directions of research in the domain of hate speech:

1. Differentiating offensive language and non-offensive language
2. Removing bias in existing hate speech detection systems
3. Distinguishing types of offensive language like racism, sexism, or violence against minorities

We focus on *hate speech detection*, i.e., a binary classification whether an input sequence contains offensive language.

[1]`https://www.facebook.com`
[2]`https://twitter.com/`
[3]`https://www.youtube.com`

3.3 Datasets for Experimental Evaluation

This section starts with a description of our pre-processing procedure. Subsequently, we present our datasets and describe their particularities and dataset statistics.

We use three labeled datasets for the two downstream tasks, presented in Sections 3.2.1 and 3.2.2. Since we have another available unlabeled dataset for the hate speech domain, we use this fourth dataset solely for fine-tuning the LM for the hate speech domain experiments. For more information on our testing series, see Section 3.5.

3.3.1 Pre-processing

Before we use the datasets for any further explorations or experiments, we execute some fundamental pre-processing steps: dropping duplicated and inconsistently labeled observations, excluding all data points that consist of an empty or white-space sequence, and creating an 80/20 data split for training and testing, which keeps the class distributions. Instead of optimizing a single model's performance, we examine our hypothesis (see Section 3.1) on a broad set of experiments, so we do not use a validation set.

3.3.2 MIMIC-III

In 2016, Johnson et al. [35] released the database Medical Information Mart for Intensive Care (MIMIC-III). This database comprises clinical data on patients admitted to the Beth Israel Deaconess Medical Center in Boston. The authors deidentified the patient's data, e.g., removed location, telephone number, name, and shifted dates.

MIMIC-III consists of detailed information on patients, such as diagnosis, prescriptions, and transfers. We solely use the two tables, `ADMISSIONS`, and `NOTEEVENTS`. The admissions table contains the column `HOSPITAL_EXPIRE_FLAG`, which indicates whether the patient died within the given hospitalization. Commonly, clinicians describe the patients' medical state in an ongoing document, leading to a discharge summary. We access these by filtering notes by the column `CATEGORY`. Since we are interested in using the document's state at the patients' admission, we identify sections known at admission time and delete all other sections. For example, sections known at the patient's admission are *Chief complaint*, *Allergies*, or *Family history*. From now on, the term *admission letters* refers to these cleaned discharge summaries.

Dataset statistics After pre-processing, MIMIC-III consists of $43,848$ observations, where $4,608$ are positive, and $39,240$ are negative examples. The average admission letter has $2,507.96$ characters with a standard deviation of $1,438.25$ and a median of 2284. The shortest has 153 characters and the longest $65,365$.

3.3.3 GermEval 19

GermEval is a competition that publishes different tasks focusing on NLP for the German language. In 2019, the second task of GermEval was about identifying offensive language. The authors published the corresponding dataset, *GermEval 19* [66].

Struß et al. [66] described that GermEval 19 was collected by sampling tweets from various users' timelines. In contrast to random sampling or on the basis of query terms, this approach helps to obtain a higher variance in topics and vocabulary. Before the authors hand-labeled the comments, they checked the alignment of their sense for hate speech. Subsequently, they hand-labeled the collected comments and used `OFFENSE` as the positive class label, i.e., comments with offensive language. The label `OTHER` marks the negative comments that belong to the negative class, i.e., without offensive language.

Dataset statistics After pre-processing, GermEval 19 consists of 7,011 observations, where 2,252 are positive, and 4,759 are negative examples. The average comment has 160.28 characters with a standard deviation of 108.65 and a median of 138. The shortest has 12 characters and the longest 4,536.

3.3.4 NOHATE

Overcoming crises in public communication about refugees, migration, foreigners (NOHATE)[4] is a joint research project that consists of *Freie Universität Berlin (FU)*, *Beuth University of Applied Sciences Berlin (BHT)*, and *VICO Research & Consulting*. It is funded by the German Federal Ministry of Education and Research (BMBF). "NOHATE aims to analyse hateful communication on social media platforms, in online forums and commentary sections [...] [and] develop methods and software for (early) recognition of hateful communication [...] and provide data for software development"[5].

The NOHATE dataset consists of comments from a wide range of online platforms, to name a few: FOCUS Online[6], ZEIT[7], Tagesspiegel[8], Facebook[9], Youtube[10], gutefrage[11], Tichys Einblick[12], and many more. Scientists from the FU developed a Code Book that describes a classification scheme to label hateful comments. In the end, a subset of the downloaded comments was, based on this Code Book, hand-labeled. To be consistent with GermEval 19, we simplify the multi-class dataset NOHATE into a binary-class dataset and used the labels `OFFENSE` and `OTHER` respectively. [3]

Dataset statistics After pre-processing, NOHATE consists of 11,935 observations, where 2,918 are positive, and 9,017 are negative examples. The average comment has 316.47 characters, with a standard deviation of 385.75 and a median of 197. The shortest has 1 character and the longest 9,717.

[4]`http://nohate.online`

[5]Source: `https://www.polsoz.fu-berlin.de/en/kommwiss/v/bmbf-nohate/index.html` - Accessed July 18, 2020

[6]`https://www.focus.de`

[7]`https://www.zeit.de/`

[8]`https://www.tagesspiegel.de`

[9]`https://www.facebook.com`

[10]`https://www.youtube.com`

[11]`https://www.gutefrage.net`

[12]`https://www.tichyseinblick.de`

3.3.5 NOHATE Language Modeling

Hand-labeling data is a particularly time-consuming and expensive process. However, pre-training BERT or fine-tuning its language model does not need labeled data (see Section 2.1.4). Therefore, we use the set of unlabeled comments (see Section 3.3.4) for LM fine-tuning. LM fine-tuning adapts the model's LM into a specific domain and helps to increase the downstream task's prediction performance.

Dataset statistics After pre-processing, NOHATE LM consists of $177,203$ observations. The average comment has 242.94 characters with a standard deviation of 745.24 and a median of 140. The shortest has 1 character and the longest $276,313$.

3.4 Metrics for the Evaluation of Experiments

For the evaluation of the experiments, we use metrics that aim at different aspects. First, to compare the models' prediction performance, we use metrics that depend on the downstream tasks. Subsequently, Section 3.4.2 defines metrics for the evaluation of the compression performance.

3.4.1 Model Performance

This section describes the metrics to measure the model's performance. As downstream tasks, we use exclusively binary classification (see Section 3.2).

Confusion Matrix, Precision, and Recall Common metrics for binary classification are *precision* and *recall* that can be easily explained and derived by looking at the *confusion matrix* in Table 3.1 [24]. Columns represent the predictions and rows the actual labels.

Actual \ Predicted	Negative	Positive
Negative	TN	FP
Positive	FN	TP

Table 3.1: Confusion Matrix. Rows present the actual labels, and columns present the models' predictions. The four quadrants (in case of binary classification) represent the four types of predictions: true negative (TN), false positive (FP), true positive (TP), and false negative (FN). Derived from [24].

In the case of binary classification, the four quadrants show the *true negative (TN)*, *false positive (FP)*, *true positive (TP)*, and *false negative (FN)* values. As an example, TN describes how many negative predicted examples are actually negative observations; the others are corresponding. Therefore, a perfect classifier would only have non-zero values on the diagonal.

Precision is the accuracy of the positive predictions, which describes how many of the positive predicted observations are TP; it is defined as:

$$precision = \frac{TP}{TP + FP} \tag{3.1}$$

Whereas recall shows how many of the positive class are predicted correctly as so, it is defined as:

$$recall = \frac{TP}{TP + FN} \tag{3.2}$$

In general, it is easier to compare models based on a single metric. For this reason, the *F1-score* combines and balances both.

3.4.1.1 F1-score

The F1-score is the harmonic mean of precision and recall. Therefore, a classifier only gets a high F1-score, if both precision and recall are high. A perfect classifier would have $F1 = 1$. It is defined as:

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 \times \frac{precision \times recall}{precision + recall} = \frac{TP}{TP + \frac{FN+FP}{2}} \tag{3.3}$$

The authors of GermEval 19 [66] stated that because the dataset is imbalanced (see Section 3.3.3), the systems will be ranked based on the *macro-average* F1-score. For this reason, we also use this metric to measure the performance for the datasets GermEval 19 and NOHATE.

Macro-average F1 For imbalanced datasets, classifiers tend to give more weight to the major class. For equal treatment of the classes, macro-averaging calculates the F1-score for each class separately and computes the average. However, this is only legible if the classes are equally important. [64]

3.4.1.2 ROC/AUC-score

A visual representation of a binary classifier's performance is the Receiver Operating Characteristic (ROC) curve. Shown in Figure 3.1, ROC plots the *true positive rate*, also known as recall, against the *false positive rate*, which is the ratio of negative observations that are incorrectly classified as positive. For the ROC curve, the false and true positive rates need to be computed with different cutoff points, i.e., moving the trade-off between precision and recall from one to the other extreme. [24]

For comparison of classifiers, the Area Under the Curve (AUC) can be used, represented as the gray area in Figure 3.1. The metric describes how well the model is capable of distinguishing the classes. A perfect classifier would achieve $ROC/AUC = 1$, whereas a random binary classifier has $ROC/AUC = 0.5$.

Since the ROC/AUC-score is widely used for in-hospital mortality prediction systems, we use this metric for comparison. [30, 49]

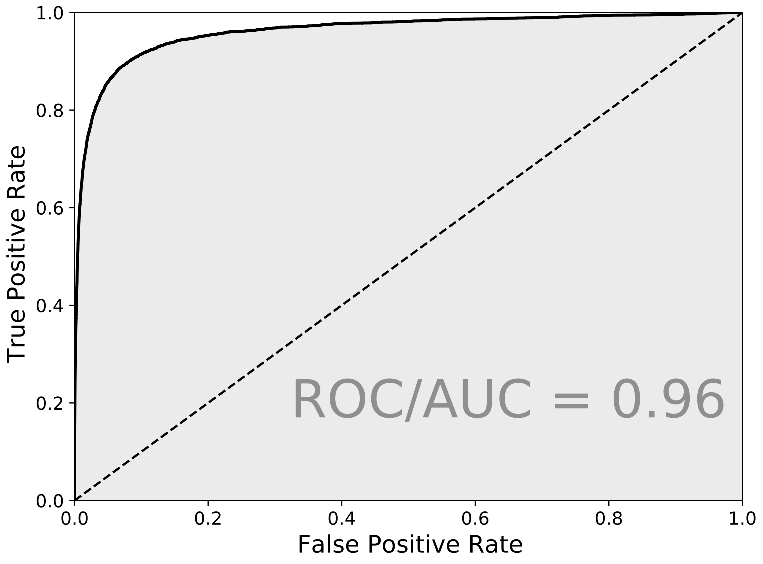


Figure 3.1: Example ROC/AUC plot. The gray area represents the ROC/AUC-score, which is equal to 0.96. The dashed line shows the ROC for a random binary classifier; its ROC/AUC-score is equal to 0.5. Derived from [24].

3.4.1.3 Perplexity

To measure the performance of language models, the most widely used metric is *perplexity*, e.g., see [19, 54, 55]. As a reminder, language modeling is the task to predict a token based on its context (see Chapter 2.1.4). Therefore, perplexity represents the uncertainty the model has in predicting the next word.

Entropy Perplexity is based on the information-theoretic concept of *entropy*, which is a measure of information. It represents the average information of a message with a corresponding probability distribution. A language model m tries to estimate the probability distribution p of the actual language. Therefore, the true entropy of p is defined as $H(p)$. By drawing sequences from p, one can use the simplified model m to estimate the language's entropy: $H(p, m)$. It turns out that the cross-entropy $H(p, m)$ is an upper bound on $H(p)$, mathematically: $H(p) \leq H(p, m)$. Thus, the difference between $H(p)$ and $H(p, m)$ is a measure of m's performance. By drawing a test sequence W with length N from p, it is possible to use Equation 3.4 to estimate LM's entropy. [38]

$$H(W) = -\frac{1}{N} log(p_i) \tag{3.4}$$

Where p_i is m's prediction, i.e., the probability for W. Usually, one draws a set of test sequences and computes the average cross-entropy.

However, entropy is not an intuitive measurement, which is the reason one uses perplexity. It represents how surprised the model was about the predictions, or in other words, from how many possibilities can be randomly chosen to get the same result. Thus, better models produce smaller perplexity. It is defined as [38]:

$$perplexity = e^H \tag{3.5}$$

These metrics are useful to compare models that are fine-tuned for the corresponding tasks. However, to represent the performance of TC itself, other metrics are necessary. These are defined in the following sections.

3.4.2 Compression Performance

Measuring the compression process's performance can be along several dimensions, such as *time*, *space*, or *predictive performance*. Since we are interested in whether TC can reduce the experiment iteration times (see Section 3.1), we focus on three metrics. First and foremost, the predictive performance that could be retained, the speedup of fine-tuning, and speedup of inference.

In general, we calculate the compression performance for the corresponding baseline. In the following sections, we denote the baseline with the subscript $baseline$.

Compression ratio The *compression ratio* achieved with TC is implicitly chosen by the definition of predecessor and successor modules. Thus, we do not use the compression ratio as a metric to measure compression performance. Nevertheless, we state the compression ratio *CR*, which is defined as follows in Equation 3.6, for each experiment.

$$CR = \frac{parameters_{baseline}}{parameters_{compressed}} \tag{3.6}$$

3.4.2.1 Retained Performance

Following Section 2.2, the goal of model compression is to reduce the number of parameters without or with as little performance loss as possible. Thus, the most important performance measure for the compression process is the *retained performance (RP)*. Therefore, the task-specific metric is used to compute how many percent of the baseline performance could be retained during compression. *RP* is defined as follows:

$$RP = \frac{performance_{compressed}}{performance_{baseline}} \tag{3.7}$$

Where $performance$ denotes the model's predictive performance measured as $F1$ for hate speech detection (see Section 3.4.1.1) and ROC/AUC for in-hospital mortality prediction (see Section 3.4.1.2).

3.4.2.2 Speedup of Fine-tuning

Using a smaller or compressed model during downstream task training theoretically increases the training speed because of the reduced computational effort. However, the training's duration until a satisfying model performance is a more liable metric. Therefore, we define the *speedup of fine-tuning* as a factor of decreased training duration for downstream task training. It is calculated as the ratio between the baseline and the compressed model:

$$speedup_{FT} = \frac{time\ until\ best\ model_{baseline}}{time\ until\ best\ model_{compressed}} \tag{3.8}$$

For measuring the training duration we choose the best model checkpoint, where *best* is measured by the model's predictive performance.

3.4.2.3 Speedup of Inference

As stated in Section 1.1, one reason for model compression is the long inference time, especially when models are deployed on weaker or non-specialized hardware. To gain insights on this, we introduce a very similar metric to Section 3.4.2.2, *speedup of inference*. It is the ratio of inference time between baseline and compressed model:

$$speedup_{n@hw} = \frac{inference\ time_{baseline}}{inference\ time_{compressed}} \tag{3.9}$$

Where n denotes the number of examples predicted at the same time, and hw states whether GPU or CPU was used. For example, $speedup_{1@CPU}$ represents an experiment based on a CPU with one test input, whereas $speedup_{16@GPU}$ stands for 16 examples simultaneously predicted on a GPU.

Section 3.4 laid the foundations for the evaluation of TC by defining the necessary metrics that will be used to compare the experiments. The following Section 3.5 gives an overview of the direction of evaluation.

3.5 Benchmark Framework to Evaluate Theseus Compression

To evaluate whether TC can reduce experiment iteration times, we use a set of experiments. The following sections give a brief outline of these. The experiment implementations are described in greater detail in Chapter 4, a discussion follows in Chapter 5.

Testing series As described in Section 3.3, we have three datasets to train the two downstream tasks. Since we want to examine our hypothesis whether a domain specific fine-tuned BERT helps to decrease iteration times, there are three experiments. These are comparing the during downstream task training compressed BERT with a during LM fine-tuning compressed BERT that are both afterwards trained for the corresponding downstream task. Additionally, we use the on NOHATE LM fine-tuned BERT and subsequently train the hate speech detection downstream task based on GermEval 19.

3.5.1 Baseline Strategies

To achieve maximum comparability, it is important to keep as many parameters as possible of the experimental environment unchanged. Available implementations for hate speech detection introduce additional data [52], use data augmentation and pre-processing [52, 58], or another base model [27], which makes it difficult to compare them to our system. Further, there are no published baseline scores for our definition of in-hospital mortality prediction. For the sake of maximum comparability, we implement baselines in a comparable setting. One can find a detailed evaluation in Section 5.2.

3.5.2 Initialization of Successor Model

The paper [78] that describes TC uses the first n layers of BERT as initialization for predecessor modules. However, this seems to be counter-intuitive because, e.g., $succ_6$ that replaces layer 11 and 12 of the original BERT is initialized with the parameters of layer 6. We hypothesize that using, in this example, layer 11 or 12 as initialization is beneficial for the compression process because its initial state is closer to the layers it will replace. Detailed information on the implementation is in Section 4.2.1 and an evaluation and discussion in Section 5.3.

3.5.3 Hyperparameter Analysis

To evaluate TC in more detail based on our real-world datasets, we use grid search to develop a deeper understanding of the hyperparameters' influence. However, to decrease the number of experiments, we adjust our testing series. Since the dataset GermEval 19 is very small and LM fine-tuning is not very helpful (see Sections 5.2 and 5.3), we focus on compressing during LM fine-tuning based on the datasets NOHATE and MIMIC-III. Evaluation and discussion are in Section 5.4.

3.6 Infrastructure for Experimental Evaluation

For our experiments, we use the DATEXIS Kubernetes cluster. It consists of 24 nodes with about $1,500$ CPU cores, where some nodes are additionally equipped with GPUs. In total, there are 8 Nvidia Tesla K80, 8 Nvidia Tesla P100, and 9 Nvidia Tesla V100 GPUs.

Our experiments are solely executed on a single GPU setup with one Nvidia Tesla V100 GPU. We use Docker containers based on Python version 3.7.5, CUDA[13] version 10.1.243, and cuDNN[14] version 7.6.5.32. The implementation of TC is based on the libraries `PyTorch` version 1.4.0 and `transformers` version 2.7.0.

3.7 Summary

This chapter starts with the formulation of our hypothesis: *Using a with Theseus Compression compressed domain-specific BERT helps speed up experiment iterations for downstream task models.*

[13]Toolkit for creating GPU accelerated applications. Source: `https://developer.nvidia.com/cuda-toolkit` - Accessed August 02, 2020

[14]The Deep Neural Network library provides GPU-accelerated primitives for NNs. Source: `https://developer.nvidia.com/cudnn` - Accessed August 02, 2020

Secondly, we define the downstream tasks used for the experimental evaluation of TC for the two domains, medical and hate speech. Both are binary classifications: in-hospital mortality prediction and hate speech detection.

The training of these downstream tasks is based on four datasets. For the medical domain, we create a dataset using the discharge summaries for each admission of the MIMIC-III database and delete sections added after the patient's admission (see Section 3.3.2). GermEval 19 and NOHATE are both datasets used to train hate speech detection, and the fourth unlabeled dataset, NOHATE LM, to fine-tune the LM of hate speech detection base models. All three labeled datasets are heavily unbalanced, and the length of their input sequences varies from, on average, about 160 (GermEval 19) characters to about $2,500$ (MIMIC-III).

Subsequently, this chapter describes the metrics that we use to evaluate our experiments and the metrics to assess the model's prediction performance. For hate speech detection, we use $F1$ and for in-hospital mortality prediction, the ROC/AUC-score. Whereas LMs are evaluated with the metric perplexity. On the other hand, we evaluate the performance of TC by calculating the retained prediction performance, speedup of fine-tuning, and speedup of inference.

We then give a brief overview of our benchmark framework for the experimental evaluation, starting with the implementation and training of our baseline models. Followed by experiments regarding the initialization procedure, we hypothesize the procedure as sub-optimal and propose a different one. We then use a grid search to train a wide range of models with different hyperparameters to gain a deeper understanding of their influence on the compression process.

In the end, we present the infrastructure of our experiments. For the implementations, we use `PyTorch` version 1.4.0 and `transformers` version 2.7.0 and execute them with Python version 3.7.5 in a single GPU setup with one Nvidia Tesla V100 GPU.

The subsequent Chapter 4 describes the implementation of TC and its compression process.

4 Implementation

This chapter presents our Theseus Compression implementation. It is based on *Hugging Face's*[1] `transformers`[2] library, which is why we start with a brief overview of this library and describe how to use it to implement and train BERT models.

Subsequently, we give detailed information on our implementations, especially the parts that we change in contrast to the original implementation. In the end, we outline some particularities of the compression process when applying TC.

4.1 Model Implementations with Hugging Face's Transformers Library

Hugging Face's `transformers` Python library for PyTorch[3] or TensorFlow 2.0[4] provides NLP model architectures in an easy to use fashion and a wide range of pre-trained models. As a reminder of Section 3.6, our software stack consists of Python 3.7.5, PyTorch 1.4.0, and `transformers` 2.7.0.

Model implementation with transformers BERT implementations based on the `transformers` library generally consists of four parts. First and foremost, the class `BertTokenizer` is responsible for creating token embeddings out of an input sequence. The class `BertModel` combines two parts: `BertEmbeddings` that create BERT's input representation (token + segment + position embedding) and `BertEncoder`, which is the actual stack of transformer encoders. Lastly, a task-specific layer is added on top of the encoder. On the one hand, for downstream task implementations, i.e., binary classification, we use the class `BertForSequenceClassification`. On the other hand, for LM fine-tuning, the `transformers` library provides the class `BertForMaskedLM`. Because there is recent work that questions the necessity of NSP [45, 37], we omit the pre-training task NSP for more straightforward experiment implementations.

Similarly to the original implementation of BERT[5], `transformers` BERT implementation can not handle input sequences with more than 512 tokens. Therefore, the library truncates sequences of > 512 tokens automatically.

[1]Link: `https://huggingface.co`

[2]Link: `https://huggingface.co/transformers/v2.7.0/`

[3]Link: `https://pytorch.org`

[4]Link: `https://www.tensorflow.org`

[5]Original BERT implementation: `https://github.com/google-research/bert` - Accessed August 12, 2020

Pre-trained base models Using pre-trained models for development or inference is extraordinarily simple. For this purpose, the base class `PreTrainedModel` is used for all models implemented with the `transformers` library and provides a method called `from_pretrained`. It takes either the name of a provided pre-trained model[6] and downloads it if necessary, or a local path to a directory containing model weights. These model weights are saved by using the model's `save_pretrained` method. Initializing a `BertTokenizer` with a vocabulary of a pre-trained model works in the same way.

The implementation of TC, explained in the following Section 4.2, uses inheritance to extend already implemented models from the `transformers` library to make use of these features.

4.2 Theseus Compression

Xu et al. [78] built their research code based on the `transformers` library, described in Section 4.1. TC is based on the idea of module replacement that randomly substitutes predecessor modules with their corresponding successor modules (see Section 2.2.4). Therefore, the implementation extends the `BertEncoder` class, as shown in Listing 1. Most

```
 1  class BertEncoderTheseus(BertEncoder):
 2      def __init__(self, config, succ_n_layer=6):
 3          super(BertEncoderTheseus, self).__init__(config)
 4  
 5          self.succ_n_layer = succ_n_layer
 6          self.compression_ratio = config.num_hidden_layers // self.succ_n_layer
 7  
 8          self.bernoulli = None
 9  
10          self.pred_layer = self.layer
11          self.succ_layer = []
```

Listing 1: Implementation of the class `BertEncoderTheseus`.

importantly, it defines the predecessor modules (Line 10) as the base model's encoder stack and adds the `succ_layer` attribute (Line 11). Detailed information on the initialization of the successor modules is presented in Section 4.2.1.

Moreover, the module replacement implementation is added to the model's `forward` function that is used for each forward path. Listing 2 shows the crucial parts of the implementation. The code is as straight forward as the module replacement idea itself. During training, a Bernoulli random variable r_i decides whether the predecessor $pred_i$ is replaced by its successor $succ_i$ (Line 5). Correspondingly, the attribute `layer` is changed because the parent's `forward` function, called in Line 15, uses this for inference.

[6]Link: `https://huggingface.co/transformers/pretrained_models.html`

```python
def forward(self, ...):  # omitted lengthy list of parameter definitions
    if self.training:
        self.layer = nn.ModuleList([])
        for i in range(self.succ_n_layer):
            if self.bernoulli.sample() == 1:  # REPLACE
                self.layer.append(self.succ_layer[i])
            else:  # KEEP the original
                for offset in range(self.compression_ratio):
                    self.layer.append(
                        self.pred_layer[i * self.compression_ratio + offset]
                    )
    else:  # inference always with compressed model
        self.layer = self.succ_layer

    return super(BertEncoderTheseus, self).forward(
        ...   # omitted lengthy list of parameters
    )
```

Listing 2: Implementation of module replacement.

BertModelTheseus class Since we replace the original encoder implementation by the class `BertEncoderTheseus`, we also need a custom `BertModelTheseus` class. It extends the original class and overrides the attribute `encoder`, as shown in Listing 3.

```python
class BertModelTheseus(BertModel):
    def __init__(self, config):
        super(BertModelTheseus, self).__init__(config)
        self.encoder = BertEncoderTheseus(config)
```

Listing 3: Implementation of the class `BertModelTheseus`.

Task-specific *Theseus class In the end, the task-specific model implementation for TC put all these parts together and initializes the successor modules, as shown in Listing 4. The class definition for the binary classification downstream tasks is respectively based on `BertForSequenceClassification` and called `BertForSequenceClassificationTheseus`.

As discussed in Section 3.5.2, we change the initialization procedure of the successor modules. The following Section 4.2.1 provides detailed information on the implementation.

4.2.1 Adjustment of Initialization Procedure

The original TC implementation initializes the successor modules with the first n layers of the original model. However, as hypothesized in Section 3.5.2, this seems to be sub-optimal. To examine the hypothesis, we change the initialization method, as shown in Listing 5.

```
1  class BertForMaskedLMTheseus(BertForMaskedLM):
2      def __init__(self, config, init_offset):
3          super(BertForMaskedLMTheseus, self).__init__(config)
4
5          self.bert = BertModelTheseus(config)
6          self.bert.encoder.init_theseus(init_offset=init_offset)
```

Listing 4: Implementation of the task-specific model class.

```
def init_theseus(self, init_offset):

    ... # omitted value checks for init_offset parameter

    self.succ_layer = nn.ModuleList([])
    if init_offset == "original":
        for index in range(self.succ_n_layer):
            self.succ_layer.append(self.pred_layer[index])
    else:  # init_offset = {0; 1} => first/second layer in predecessor module
        for index in range(self.succ_n_layer):
            self.succ_layer.append(
                self.pred_layer[index * self.compression_ratio + init_offset]
            )
```

Listing 5: Implementation of the initialization method.

These adjustments of the `init_theseus` method yield a maximum of flexibility over the initialization procedure of our experiments. During the initialization of a task-specific BERT model, the constructor executes this initialization method (see Listing 4, Line 6) and sets up the model for compression.

To control the replacement rate during the compression process, we use a replacement rate scheduler, described in the following Section 4.2.2.

4.2.2 Replacement Rate Scheduler

The missing piece of the TC implementation is the `CurriculumReplacementScheduler` that is responsible for the replacement rate changes throughout the compression process. Similarly to a learning rate scheduler, it takes care of the training step and calculates the corresponding replacement rate and subsequently sets the attribute `bernoulli` of the `BertEncoderTheseus` object. The encoders' `forward` method uses this attribute during compression to control the module replacement.

We make a minor change for convenience that calculates the coefficient k automatically based on the number of maximal compression steps, *starting replacement rate*, and the *ratio of replacement changes*. For example, if the ratio of replacement changes is set to 85% and the *starting replacement rate* $= 0.1$, the `CurriculumReplacementScheduler` calculates

k so that the replacement rate p is linearly increasing until it reaches the value $p = 1$ after 85% of training steps for the rest of the compression process.

Section 4.2 presented the implementation of TC itself, which mainly consists of extended classes that will be used in the training process. However, because the goal is to compress the model, we will refer to it as *compression process*. We present an overview of its implementation in the subsequent Section 4.3.

4.3 Compression Process

The implementation of the compression process only differs in a few minor aspects from a training process. Therefore, we closely follow the example implementation from the `transformers` library for training models for the GLUE-benchmark[7].

Usage of *Theseus models First and foremost, instead of the model implementations for the GLUE downstream tasks, we use our Theseus models: `BertForMaskedLMTheseus` or `BertForSequenceClassificationTheseus`. Since these model implementations extend the corresponding library implementations (see Section 4.2), we can use all these neat methods, such as `from_pretrained` or `save_pretrained`, without further changes.

Freeze predecessor modules As described in Section 2.2.4, during module replacement, TC only updates the successor modules' weights. For this reason, we distinguish the two phases of TC and handover the necessary weights to the optimizer. Listing 6 shows the corresponding lines of code.

```python
if fine_tune:
    parameters = model.parameters()
else:
    parameters = model.bert.encoder.succ_layer.parameters()

optimizer = AdamW(parameters, ...)  # omitted other parameters
```

Listing 6: During module replacement, Theseus Compression only changes the successor modules' weights. For successor fine-tuning, it updates all weights of the model.

Usage of replacement rate scheduler To apply TC as pointed out in Section 2.2.4, we use the `CurriculumReplacementScheduler`, which is responsible for the linearly increasing replacement rate and the smooth transition from the first phase of module replacement into the second phase, the successor fine-tuning. Therefore, the scheduler's `step` method is called once per training step and triggers the replacement rate's adaption.

In the end, we change the code to load our corresponding datasets, described in Section 3.3, and evaluate the models based on the metrics, described in Section 3.4.1.

[7]Source: `https://github.com/huggingface/transformers/blob/v2.7.0/examples/run_glue.py` - Accessed August 02, 2020

```python
replacement_rate_scheduler = CurriculumReplacementScheduler(
    bert_encoder=model.bert.encoder,
    starting_replacement_rate=starting_replacement_rate,
    max_training_steps=max_training_steps,
    ratio_replacement_changes=ratio_replacement_changes
)

# omitted other code and finally use the scheduler for replacement rate updates

replacement_rate_scheduler.step()
```

Listing 7: Initialization and usage of the `CurriculumReplacementScheduler`.

4.4 Summary

In this chapter, we first give an introduction of Hugging Face's `transformers` library and how to implement BERT downstream task models. It uses a nested structure of class objects that represents different parts of BERT, such as `BertTokenizer`, the `BertModel` that combines `BertEmbeddings` and `BertEncoder`, and finally, a downstream task-specific class that adds a corresponding layer.

We then focus on the presentation of our implementations. The class `BertEncoderTheseus` that extends the `transformers`' `BertEncoder` class distinguishes between the predecessor and successor modules, implements the module replacement by extending the `BertTokenizer`'s forward method, and takes care of the proper initialization of successor modules.

This chapter closes with a detailed description of the parts we adjust compared to a regular training process. To summarize, these are:

- Use the implemented `*Theseus` classes instead of the original downstream model
- Only train the successor module's parameters by freezing the weights of the predecessor modules
- Use the `CurriculumReplacementScheduler`, which is responsible for controlling the replacement rate during the compression process

Based on our implementations described in this chapter, Chapter 5 benchmarks, evaluates, and discusses our experiments.

5 Benchmark, Evaluation, and Discussion

This chapter starts with the definition of hyperparameters that remain static throughout our evaluation and presents our baselines.

We then successively describe our experiments, present their results, and discuss them. Starting with the comparison of the original and our initialization method, followed by a more detailed hyperparameter analysis of TC's hyperparameters to learn how they influence the compression's performance.

Subsequently, we use the best compressed models to evaluate the compression performance on different dimensions. We look at the retained prediction performance, the time to compress the model and train the downstream task, and the models' prediction speed.

To learn whether TC transfers the predecessor's knowledge into the successor, we perform qualitative error analysis and use the error class distribution changes during the compression process as an indicator.

5.1 Experimental Setup

This section describes our settings that remain static throughout all experiments to decrease the search space of the hyperparameter analysis.

Max sequence length = 512 As pointed out in Section 4.1, the models can not handle input sequences longer than 512 tokens. After testing the lengths of tokenized input sequences, it turns out that for MIMIC-III, more than 62% of all observations are longer than 512 tokens. For the other three datasets, there are almost no token sequences too long. Truncating the token sequences to, e.g., 256 tokens would be reasonable for GermEval 19 and NOHATE, however, almost 90% of MIMIC-III observations are affected. Thus, the datasets' variance is much smaller, which leads to models with worse prediction performance. For the sake of consistency and a more straight forward experiment setup, we fix the *max sequence length* to 512 to achieve the maximal prediction performance for the models.

Batch size = 16 For pre-training BERT, one often uses big batch sizes, e.g., 256 [19], or $1,024$[1]. On the other hand, for downstream task training, batch sizes around 32 [19, 58] are common. We use relatively long input sequences, thus, the memory consumption during training is relatively high. Therefore, 16 is the maximal *batch size* that works for all our experiments.

[1]Training of German BERT. Source: `https://deepset.ai/german-bert` - Accessed August 03, 2020

Curriculum Replacement Scheduler Xu et al. [78] stated that using a curriculum replacement scheduler consistently outperforms compression processes with a constant replacement rate. For this reason, we focus on finding good values for its hyperparameters, i.e., *starting replacement rate* and the *ratio of replacement changes*.

Base models Since we use downstream tasks for different languages, we need corresponding base models for compression. For the in-hospital mortality prediction task, which data is in English, we use the original pre-trained $\text{BERT}_{\text{base}}$ [19]. The `transformers` library uses the name `bert-base-uncased` to reference this architecture and pre-trained weights.

As described in Sections 3.3.3 and 3.3.4, GermEval 19 consists of German tweets and NOHATE of comments from a wide range of German online platforms. Therefore, another base model that is pre-trained on a German corpus is needed. The Berlin-based company deepset[2] trained a German BERT[3], which is integrated into the `transformers` library under the name `bert-base-german-cased` and serves as base model for our hate speech detection experiments.

5.2 Baseline Models

As simple baselines, we fix all hyperparameters during the training process. We fine-tune the LMs for 50 epochs, and measure the prediction performance of all models within 15 epochs of downstream task training. Additionally to Section 5.1, we choose the *learning rate* $= 2 \times 10^{-5}$, following [78].

Table 5.1 presents the model with the best prediction performance on the test dataset for each testing series. As pointed out in Section 3.4.1, for hate speech detection models, the metric F1 macro is decisive, and for in-hospital mortality prediction ROC/AUC. Column *Time* represents the time (LM fine-tuning + downstream task training) necessary to train this model.

As a reminder, the training set of GermEval 19 consists of about $5.6k$ observations, which is why downstream task training starts overfitting after about 2.5 epochs and 4 minutes of training. Fine-tuning the LM of `bert-base-german-cased` for 50 epochs with GermEval 19 results in a highly overfitted LM with *perplexity* $= 11.94$. Not surprisingly, using this model as the base model for hate speech detection downstream task training only increases the prediction performance by 0.0005 macro F1. NOHATE LM drastically increases the prediction performance by 0.0627 macro F1.

After 50 epochs of LM fine-tuning `bert-base-german-cased` on NOHATE LM, it achieves perplexity of 6.13. Based on this model, only four epochs with about 13 minutes of downstream task training are necessary to increase the initial macro F1 by 0.0155.

Similarly to the NOHATE baseline, fine-tuning the LM of the model `bert-base-uncased` for 50 epochs on MIMIC-III (*perplexity* $= 1.85$) helps to decrease the necessary training time for the downstream task training. Here, about 11 epochs (2 h) increase the ROC/AUC score by 0.0139. Its baseline needs 36.5 epochs and about 11.5 h training.

[2]Berlin-based NLP company with focus on Open-Source development. Link: `https://deepset.ai`

[3]Source: `https://deepset.ai/german-bert` - Accessed August 03, 2020

Setting	Prediction Performance			Time
	F1 macro	ROC/AUC	Accuracy	
GermEval 19	0.6768	0.6726	0.7262	4 min
LM + GermEval 19	0.6773	0.6748	0.7267	36 min + 4 min
NOHATE LM + GermEval 19	0.7395	0.7331	0.7799	39.2 h + 15 min
NOHATE	0.7125	0.7019	0.7990	36 min
LM + NOHATE	0.7282	0.7118	0.8082	39.2 h + 13 min
MIMIC-III	0.6728	0.7057	0.8767	11.5 h
LM + MIMIC-III	0.6802	0.7196	0.8899	118.2 h + 2 h

Table 5.1: Baselines for experiments. Only the best model based on macro F1 for hate speech detection and ROC/AUC for in-hospital mortality prediction is shown for each experiment. Column *Time* represents the training for the given model.

5.3 Initialization Method

To examine the hypothesis presented in Section 3.5.2, we use our testing series, described in Section 3.5, vary the $initialization\ procedure = \{original; ours\}$, and fix all other hyperparameters. As in Section 5.2, we use the $learning\ rate = 2 \times 10^{-5}$ and additionally choose the starting replacement rate $b = 0.1$, and the $ratio\ of\ replacement\ changes = 85\%$.

Table 5.2 shows the best models' results, measured on their test dataset prediction performance within 15 epochs of downstream task training, for the original initialization procedure and our method. It only shows the experiments for which we use the predecessor modules' lower layer as initialization for the successor modules. Using the upper layer shows significantly and consistently worse results. As Xu et al. [78] pointed out, TC adds extra regularization, and thus the training process needs more time until it converges, which is why we compress all models for 50 epochs during LM fine-tuning. Section 5.2 shows that in general, LM fine-tuning for 50 epochs results in good LMs. Even if the language model for GermEval 19 is highly overfitted, we keep the hyperparameter $max\ epochs$ unchanged for our experiments.

On the other hand, we run experiments without LM fine-tuning for 50 epochs and compress them directly during downstream task training. This procedure is equal to the original approach of TC, proposed by Xu et al. [78].

Our initialization method increases TC's performance Our initialization procedure outperforms the original in almost all presented metrics. Further, it shows that during LM fine-tuning compressed models can retain a high degree of prediction performance. Moreover, it increases the effectiveness of using a pre-trained language model, especially with bigger datasets. As an example, it increases the ROC/AUC score of the in-hospital mor-

Setting	Prediction Performance					
	F1 macro		ROC/AUC		Accuracy	
	Orig	Ours	Orig	Ours	Orig	Ours
GermEval 19	0.6660	**0.6687**	**0.6675**	0.6673	0.7066	**0.7149**
LM + GermEval 19	0.6647	**0.6667**	0.6672	**0.6949**	0.7040	**0.7139**
NOHATE LM + GermEval 19	0.7019	**0.7210**	0.7008	**0.7110**	0.7419	**0.7703**
NOHATE	**0.7012**	0.6903	**0.6974**	0.6755	0.7835	**0.7922**
LM + NOHATE	0.7096	**0.7171**	0.7059	**0.7097**	0.7893	**0.7985**
MIMIC-III	0.6154	**0.6364**	0.6212	**0.6755**	0.8524	**0.8788**
LM + MIMIC-III	0.6395	**0.6491**	0.6693	**0.7097**	0.8425	**0.8618**

Table 5.2: Comparison of initialization procedures. It compares the original initialization procedure (columns *Orig*) with our method (columns *Ours*), where we use the predecessor modules' lower layer as initialization for the successor modules. Because these experiments consistently outperformed the initialization with the predecessor's upper layer, we omit the worse testing series. Better results within one metric are in bold.

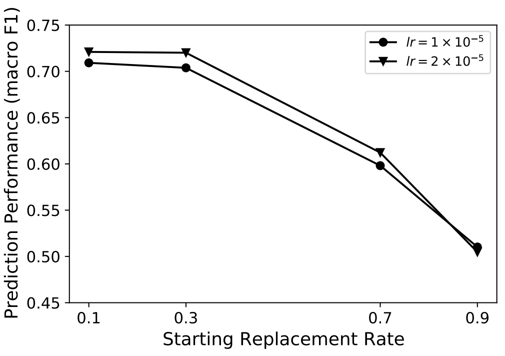
tality prediction task by 0.0342 (our initialization), which is higher than the baseline's performance lift (0.0139). Whereas, the smaller dataset NOHATE lifts the macro F1 score by 0.0268, where the baseline achieves an increase of 0.0155. For the smallest dataset, GermEval 19, using an LM decreases the performance.

Summary This series of experiments clearly shows that our initialization procedure, where the predecessor modules' lower layer is used as initialization for the successor modules' layer, is superior to the original initialization procedure. Therefore, we use our method for all upcoming experiments.

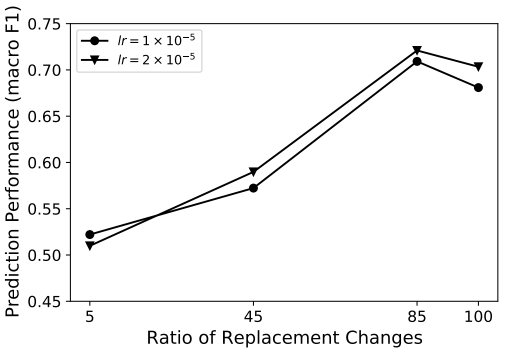
5.4 Hyperparameter Analysis

Based on the experiment in Section 5.3, we argue that using the predecessor's lower layer's weights to initialize the successor, i.e., our initialization procedure, is beneficial for the compression's performance. Thus we fix the hyperparameter $initialization\ procedure = ours$. With the other three hyperparameters, we build a grid for hyperparameter analysis: $learning\ rate = \{1 \times 10^{-5}; 2 \times 10^{-5}\}$, $starting\ replacement\ rate = \{0.1; 0.3; 0.7; 0.9\}$, and ratio of replacement changes $b = \{5\%; 45\%; 85\%; 100\%\}$.

Since we are interested in compressing models during fine-tuning their LM and the fact that GermEval 19 has proved too small, we compress models based on NOHATE and MIMIC-III for 50 epochs. Subsequently, we evaluate them on the test dataset by training their downstream task for 15 epochs with a learning rate equal to 2×10^{-5}.

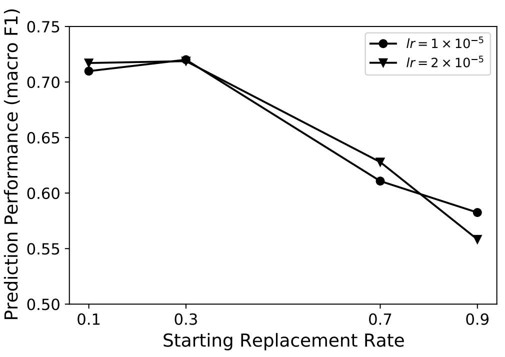


(a) Influence of *starting replacement rate* on prediction performance. Hyperparameter *ratio of replacement changes* is fixed to 85%.

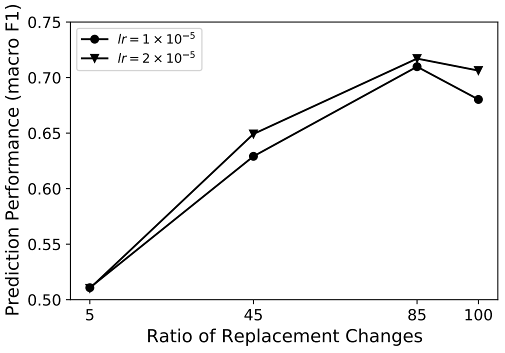


(b) Influence of *ratio of replacement changes* on prediction performance. Hyperparameter *starting replacement rate* is fixed to 0.1.

Figure 5.1: Hyperparameter analysis. Compressed on NOHATE and evaluated on GermEval 19. The lines distinguish experiments with different *learning rate*.



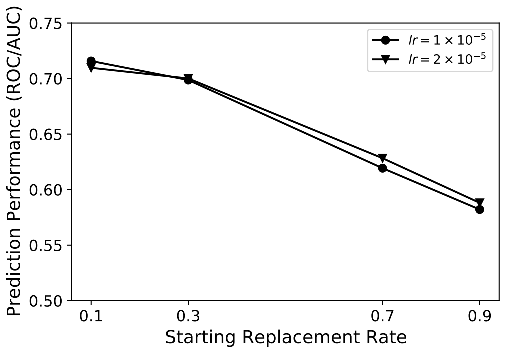
(a) Influence of *starting replacement rate* on prediction performance. Hyperparameter *ratio of replacement changes* is fixed to 85%.



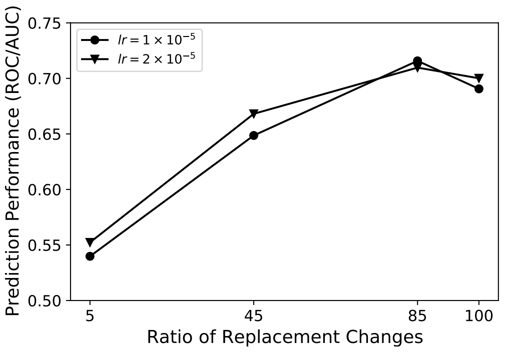
(b) Influence of *ratio of replacement changes* on prediction performance. Hyperparameter *starting replacement rate* is fixed to 0.1.

Figure 5.2: Hyperparameter analysis. Compressed and evaluated on NOHATE. The lines distinguish experiments with different *learning rate*.

Figures 5.1, 5.2, and 5.3 demonstrate the best results of the 15 epochs downstream task training. We decide to present our results either as a function of *starting replacement rate* or *ratio of replacement changes* to gain insights into the influence of these hyperparameters. The other is then fixed to *ratio of replacement changes* = 85% or starting replace-



(a) Influence of *starting replacement rate* on prediction performance. Hyperparameter *ratio of replacement changes* is fixed to 85%.



(b) Influence of *ratio of replacement changes* on prediction performance. Hyperparameter *starting replacement rate* is fixed to 0.1.

Figure 5.3: Hyperparameter analysis. Compressed and evaluated on MIMIC-III. The lines distinguish experiments with different *learning rate*.

ment rate $b = 0.1$ because in a wide range of experiments, these hyperparameters lead to good results.

Learning rate plays a minor role in TC's performance First and foremost, it stands out that the learning rate has no apparent influence on the general trend of models' prediction performance. In some specific settings, the learning rate hardly influences the final result, e.g., Figure 5.1a at 0.9, Figure 5.2a at 0.3, Figure 5.2b at 5, or Figure 5.3a at 0.3. This leads to the conclusion that the learning rate plays a minor role in compression performance.

Low starting replacement rate for 85% of compression steps creates best results Further, the hyperparameters introduced by TC, starting replacement rate, and ratio of replacement changes are much more influential on the performance. As described in Section 2.2.4, using a curriculum replacement scheduler has the advantage that at the beginning of the compressing process, the successor modules are trained with much guidance. The guidance decreases steadily throughout the module replacement phase because the replacement rate gradually increases. Our experiments clearly show that using a starting replacement rate bigger than 0.3 reduces the positive effect of decreasing guidance, and the prediction performance of final models drops.

Similarly, using a low ratio of replacement changes also harms the performance. Comparing Figures 5.2a and 5.2b, or Figures 5.3a and 5.3b, the results can be much worse than using a starting replacement rate that is too high. We assume a reason for this is that in the first, e.g., 5% of compression steps, the replacing rate jumps very quickly to 1, which vanishes the effect of guidance. For the remaining 95% of compression steps, the compression process is similar to pre-training from scratch because the successor modules are not yet adapted to the predecessor modules.

Omit successor fine-tuning phase harms TC's performance Moreover, in all experiments, the performance drops when the ratio of replacement changes increases from 85% to 100% (see Figures 5.1, 5.2, and 5.3). We conduct these experiments to test the importance of TC's second phase, the successor fine-tuning. It turns out that only using the module replacement phase harms the performance of TC significantly. This effect increases with decreasing learning rate.

Summary Based on our experiments, using TC works best when the replacement rate is curriculum scheduled with a relatively low starting replacement rate of around 0.1 to 0.3, and the successor fine-tuning phase starts after 85% of compression steps. The learning rate plays a minor role in the compression process in general; however, it influences the prediction performance of a concrete experiment.

5.5 Compression Performance Analysis for Best Models

The previous sections deal with the influence of different hyperparameters, such as the initialization method in Section 5.3, or the controlling of the replacement rate in Section 5.4. This Section evaluates and discusses compression performance in greater detail.

Table 5.3 presents the models that achieve the best results on our testing series, described in Section 5.4. These are compressed within 50 epochs of LM fine-tuning, either based on NOHATE or MIMIC-III and subsequently evaluated on all three datasets. For this, we choose the best models of the test dataset within 15 epochs downstream task training. The hyperparameter's abbreviations in Table 5.3 stand for: learning rate (lr), starting replacement rate (srr), and ratio of replacement changes (rrc). All models have in common that they reduce the number of parameters from originally about 110M to 66M parameters, which equals a compression ratio of $CR = 1.67\times$.

Model Settings		Prediction Performance	Retained Performance	Speedup Fine-tuning	Time
NOHATE LM + GermEval 19	$lr = 2 \times 10^{-5}$ $srr = 0.1$ $rrc = 85$	0.7210 F1	0.9750	2.5	30.5 h + 6 min
LM + NOHATE	$lr = 1 \times 10^{-5}$ $srr = 0.3$ $rrc = 85$	0.7201 F1	0.9889	2.16	30.5 h + 6 min
LM + MIMIC-III	$lr = 1 \times 10^{-5}$ $srr = 0.1$ $rrc = 85$	0.7158 ROC/AUC	0.9947	2.14	111 h + 56 min

Table 5.3: Best models after hyperparameter analysis. Abbreviations: learning rate (lr), starting replacement rate (srr), and ratio of replacement changes (rrc).

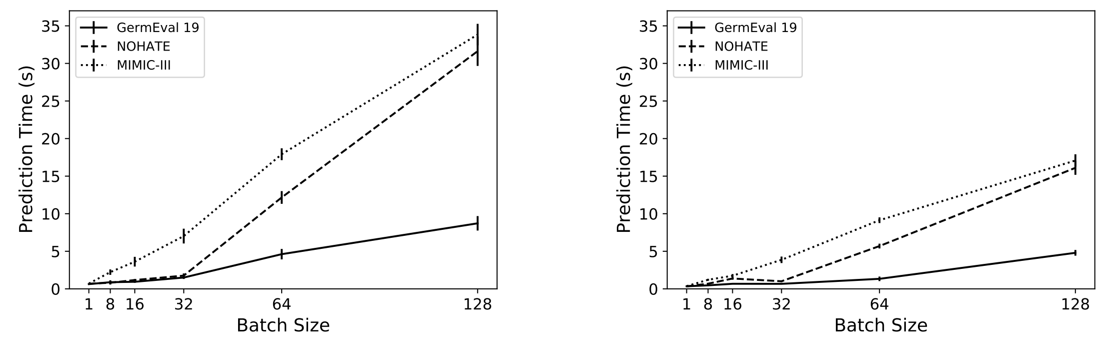
Dataset size has a positive influence on retained performance The values in column *Retained Performance* of Table 5.3 are calculated with the metric *RP*, described in Section 3.4.2.1. They clearly show that the training dataset's size has a positive influence on the compression's performance. However, the in-hospital mortality prediction experiment must be considered cautiously because the dataset for compression and downstream task training of these models is the same, even though a test set is used for evaluating the model's final performance. Thus, comparing the models' perplexity with their baselines gives more reliable insights since it is measured after the compression process before the dataset is reused for downstream task training. As a reminder, the baseline for NOHATE achieves a perplexity equal to 6.13 and 1.85 for MIMIC-III. The best compressed models reach 11.70 and 2.83, which is equal to retention of 47.15% (NOHATE), and 65.37% (MIMIC-III) of their language model capabilities. This underlines that the performance of TC increases with the size of the training dataset used for compression.

More than 2× speedup of fine-tuning and faster LM fine-tuning Another dimension of evaluating the performance of TC is the *time* dimension. The values in column *Speedup Fine-tuning* of Table 5.3 are calculated with the metric $speedup_{FT}$, described in Section 3.4.2.2. They show that using a compressed domain-specific LM reduces the necessary time for learning the actual downstream task. In our experiments, $speedup_{FT}$ is about 2.1 to 2.5. Further, the compression process is faster too. Compared to the baseline implementations (see Table 5.1), compressing based on NOHATE needs about 78%, and based on MIMIC-III, about 94% of the time necessary for LM fine-tuning of the corresponding datasets.

1.94× or 1.73× speedup of inference on CPU or GPU Moreover, we test the model's inference speed. *Inference* refers to the chain of taking the examples as a batch, tokenizing them into tensors, and using these tensors to compute the model's output. The tokenizer truncates the token sequences to maximum 512 tokens and pads the examples to the same length. We measure the time required for the entire prediction chain.

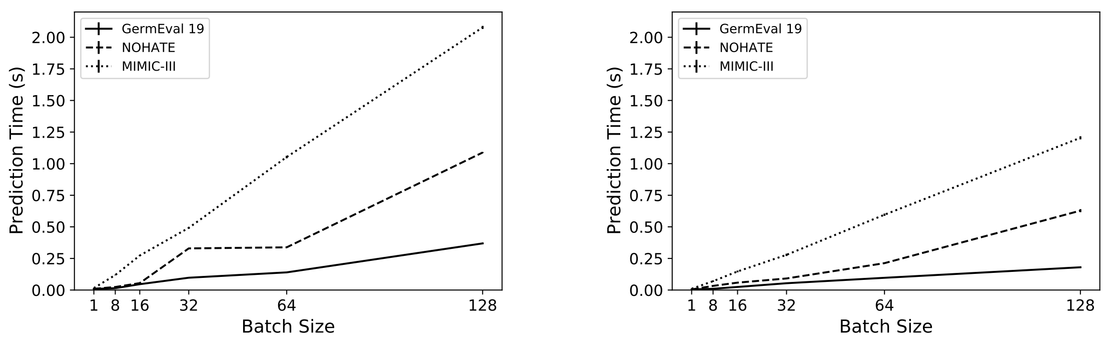
For a comprehensive comparison of the predecessor and successor models' speed, we use different batch sizes on CPU and GPU for each dataset. Figures 5.4 and 5.5 show the mean values and their corresponding standard deviations for 100 predictions for each setting. The datasets' different prediction times with the same batch size represent the dataset statistics. Observations from MIMIC-III are longest with, on average about $2,500$ characters, which is the reason for the highest prediction times. NOHATE and GermEval 19 with about 316 and 160 characters on average are much shorter and therefore need less time for inference. Comparing Figures 5.4a and 5.4b or Figure 5.5a and 5.5b, one can see an apparent reduction of prediction time when using the compressed model.

To take a closer look at the actual speedup of inference, we use the metric $speedup_{n@hw}$, described in Section 3.4.2.3. It uses the mean values of our measurements and presents the results in Figure 5.6. More detailed, one can see the CPU-based experiments, i.e., $speedup_{n@CPU}$ in Figure 5.6a, and GPU-based ($speedup_{n@GPU}$) in Figure 5.6b. On average, the speedup for CPU-based inference is 1.94× and for GPUs 1.73× with a standard deviation of 0.10 for both. However, some clear outliers for CPU and GPU drastically increase or decrease the speedup, which we do not evaluate any further. We assume that these effects



(a) Inference time of the baseline model. (b) Inference time of the best compressed model.

Figure 5.4: Inference time on CPU. Mean inference time and standard deviation for 100 predictions. Lines distinguish experiments with different datasets.



(a) Inference time of the baseline model. (b) Inference time of the best compressed model.

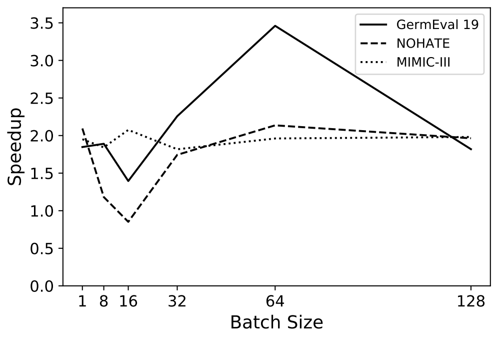
Figure 5.5: Inference time on GPU. Mean inference time and standard deviation for 100 predictions. Lines distinguish experiments with different datasets.

are based on the hardware architecture, such as changing efficiency of fetching data. It remains to say that it can be worthwhile to run experiments before deploying models in production to find the sweet spot of speedup.

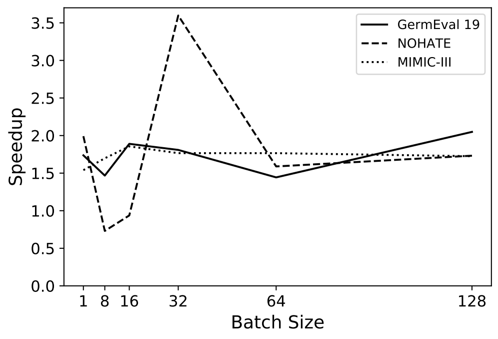
Summary Our experiments clearly show that using TC during LM fine-tuning can retain up to about 99% of the downstream task model's performance while reducing the LM fine-tuning time and speedup the downstream task training by a factor of more than 2. Moreover, the compressed model is about 1.94× on CPU and 1.73× faster for predictions on GPU.

5.6 Qualitative Error Analysis

For a more in-depth analysis of TC, we manually analyze 100 randomly sampled false positives (FP) and false negatives (FN) of our best models, shown in Table 5.3, and their corresponding baselines. Subsequently, we compare the frequencies of error classes to



(a) Speedup of inference measured on CPU ($speedup_{n@CPU}$).



(b) Speedup of inference measured on GPU ($speedup_{n@GPU}$).

Figure 5.6: Speedup of inference. The speedup of the compressed model compared to its baseline averaged over 100 predictions. Lines distinguish experiments with different datasets.

determine whether they change during the compression process. Because we only use 100 examples for the error analysis, where a single example already changes the error by 1 percentage point, we consider a change of less than three percentage points as similar.

Dataset: GermEval 19 For the task hate speech detection, van Aken et al. [1] conducted a detailed error analysis. Although they use different datasets, their introduced error classes for both false positives and false negatives, fit our needs. Therefore, we use their error classes to bootstrap our qualitative error analysis.

16% of FNs and 11% of FPs have *doubtful labels* For both types of misclassification, there are several observations for which we question the actual labels. By combining the examples, we find that 18% of 200 FPs and 15% of 200 FNs belong to the error class of *doubtful labels* for the compressed and baseline model. We assume, especially for the FPs, that the missing context this tweet was published in is why annotators have chosen the positive class. However, without knowing to what the comment refers to it is, in our point of view, free of hateful content.

Frequency of $\geq$ *5 mentions or hashtags* does not change during compression Another error class that applies for both misclassification types is when an example consists of $\geq$ 5 *mentions or hashtags*. 22% of the original and 20% of the compressed model's FPs belong to this class. On the other hand, for FNs, only 10% of the original and 11% of the compressed model consists of 5 or more mentions or hashtags. Besides the doubtful labels, this is the most significant error class for the models' FP misclassifications. However, the important part for this thesis is that the error's frequencies do not change during the compression process.

Distribution of error class *quotations or references* remains stable An error class for FPs is the usage of *quotations or references*. 10% of the original and 9% of the compressed model's examples belong to this class. As van Aken et al. [1] pointed out, the usage of offensive words in a context where it refers to, e.g., the author herself/himself or quotes a hateful comment, can confuse the classifier. Fortunately, the compression process does not increase this class frequency.

Compression decreases the frequency of *toxicity without swear words* FNs by 5 The most significant and problematic error class for our classifiers is *toxicity without swear words*. 45% of the original and 40% of the compressed model's FNs belong to this error class. Authors of such comments often use negatively connoted words for a given context to discredit people or groups. Interestingly, the compression process reduces the amount of FNs caused by this class.

Observations on misc error classes Further, for both misclassification types, which contain *rare or constructed words*, we can also observe a reduced frequency after compression. For FPs and FNs, the compression process decreases the frequency by 3 points to 9% and 10%. However, unfortunately, compression introduces the new error class of comments that consists solely of *capital letters* (3%) and harms the model's ability to handle emojis (+3%), especially for ≥ 5 *emojis* in a comment. It remains to say that capital letters and emojis are typical patterns that are pre-processed, such as change the emoji's unicode into another representation or the emoji's name [52].

Dataset: NOHATE For the dataset NOHATE, there are many more examples that belong to the error class of doubtful labels. We again combine the examples of the two models. 25% of 200 FP and 17% of FN examples are in our point of view wrongly labeled. Moreover, it seems like the observations are inconsistently labeled. A reason for this could be that the authors of GermEval 19 at least tried to measure the alignment of the annotators' sense for hate speech (see Section 3.3.3). Whereas NOHATE's authors used the Code Book, described in Section 3.3.4, as rules for annotation [3], which depends on the annotator's interpretation.

Compression decreases FP error class *rare or constructed words* by 3 Besides doubtful labels, the most problematic class for FP examples are *rare or constructed words* in comments. However, in contrast to GermEval 19, this class's frequency decreases during the compression process by 3 to 19%. This class's impact on FNs is much smaller, with 13%, and does not change during compression.

FN error classes frequencies remain unchanged after compression Further, by far, the largest error class with 46% of the predecessor's and 47% of the successor's FNs belong to *toxicity without swear words*. Other much smaller classes are *rhetorical questions* and *metaphors and comparisons*. Rhetorical questions formulate a hateful comment as questions and metaphors or comparisons often require context information or knowledge that is community-specific. These two error classes exist in 6% and 9% of the FNs, where their changes are very small with −1 and +1.

Dataset: MIMIC-III For people without a medical background, it is hard to understand the content of such admission letters. To the best of our knowledge, we try to find patterns based on the patient's age, pre-existing conditions, symptoms, and risky behavior such as smoking or alcohol use. Further, as described in Section 5.1, about 62% of all observations are too long and get truncated. For the two misclassification types, we find that 61% of the FPs and 58% of the FNs are truncated.

In the following analysis of the error classes, we omit the comparison between the original and compressed models because these distributions are extremely stable within a change of ±2.

Patient's pre-existing conditions, age, and behavior are more important than symptoms The in-hospital mortality classifier is especially sensitive to the patients' age, pre-existing conditions, and risky behavior. We find three symptoms, *abdominal pain*, *chest pain*, and *shortness of breath* that are for both misclassification types almost the same. FP: 15%, 7%, and 6%; FN: 13%, 9%, and 7%. Whereas, pre-existing conditions such as *hepatitis infection* (6%) or *cancer* (20%) are much more common for FPs than for FNs (1% and 10%). This is similar to risky behavior, 14% of FPs are smokers, and 13% drink regularly alcohol. For FNs, only 1% drink regularly alcohol, and 5% are smokers. Lastly, the distribution of patients between the ages of 60 and 90 is similar for both misclassification types. However, 12% of FPs are over 90 years, and no patient is younger than 50. On the other hand, only 6% of FNs are older than 90 years, and 6% of the patients are younger than 50. This clearly shows that the classifier preferably predicts young patients as negative and older as of the positive class.

Summary We find that the error classes we could determine remain surprisingly stable after the compression process. Therefore, we argue that TC compresses the knowledge of the predecessor into the successor model. Changes in the error class distributions would indicate that TC trains a new model rather than compressing the original. Furthermore, the bigger the dataset, the more stable the error classes.

5.7 Summary

In this chapter, we use our benchmark to run a wide range of experiments. Subsequently, we evaluate and discuss the results based on different visualizations and analysis techniques.

In Section 5.3, we show that our initialization method, i.e., using the predecessor modules' lower layer as initialization for the successor modules, is superior to the original initialization procedure. Therefore, we use our method in all experiments for the hyperparameter analysis, where we focus on developing a more in-depth understanding of the hyperparameter's influence on the compression performance.

The grid, we use for hyperparameter analysis, consists of the parameters: $learning\ rate$, $starting\ replacement\ rate$, and $ratio\ of\ replacement\ changes$. The experiments clearly show that the learning rate's influence on the compression's performance is marginal. Further, using a starting replacement rate of around 0.1 to 0.3 and linearly increase it until the successor fine-tuning phase starts after 85% of compression steps consistently achieves good results.

We use the best results for each of the datasets for an in-depth evaluation of the compression performance. First of all, we show that *using TC during LM fine-tuning can retain up to 99% of the final model's prediction performance.* TC does not only increase the downstream task training speed by more than 2× (depending on the dataset); the compression process is also faster than plain LM fine-tuning. The compressed models' inference on CPU is, on average, 1.94× faster and on GPU 1.73×.

In the end, we argue that TC reliably compresses the knowledge of the predecessor into the successor. This conclusion is based on the comparison of predecessor's and successor's error analysis and the fact that their error class distributions (before and after the compression) are incredibly stable.

We conclude that Theseus Compression, as presented by Xu et al. [78], has some drawbacks. We improve the initialization procedure and show that using TC during LM fine-tuning based on real-world datasets is a powerful tool to decrease model sizes and increase inference speed significantly. Especially in an environment where it is planned to use a domain-specific BERT and model speed or its size are essential or the limitating factor, Theseus Compression is worth considering. However, in our experiments, using a compressed model always comes with a loss of prediction performance even though the retained performances are impressive.

6 Conclusion and Future Work

In this thesis, we aim to evaluate whether a domain-specific TC compressed BERT helps speed up experiment iterations for downstream task models. Therefore, we adapt the implementation of Xu et al. [78] and propose changes to the successor model's initialization procedure.

We perform experiments to show the benefits of our initialization procedure and develop a deeper understanding of the hyperparameters' influence on the compression performance. For the hyperparameter analysis, we use grid search to get an extensive set of experiments. The best downstream task prediction performances are achieved when the compression process uses a starting $replacement\ rate$ around 0.1 to 0.3 and linearly increase it until after about 85% ($ratio\ of\ replacement\ changes$) of compression steps the successor fine-tuning phase starts.

Based on the best models, we comprehensively evaluate and present TC's compression performance. The following list gives a brief overview of the most important aspects:

- The compression process itself is faster than plain LM fine-tuning
- Compressed models are 1.67× smaller ($66M$ parameters)
- Compressed models retain up to 99% of prediction performance
- Compressed models' downstream task training is more than 2× faster
- Compressed models' inference is, on average, 1.94× faster on CPU and 1.73× on GPU

A closing qualitative error analysis shows that the models' error class distributions do not change during compression, revealing that TC transfers the predecessor's knowledge into the successor.

6.1 Conclusion

Based on our findings, we conclude that especially if LM fine-tuning is part of the new models' development process, Theseus Compression is very powerful. Thus it speeds up the rest of the process because of a compressed domain-specific BERT, which is smaller, faster, and downstream task training time is reduced. Further, it potentially enables the deployment of TC compressed final models on edge-devices [60][1], or where low model prediction times are crucial.

[1]The authors [60] experimented with a same-sized model on a recent smartphone.

Limitations On the other hand, if prediction performance is the most important metric, TC is not useful because it, at least in our experiments, always harms the model's prediction performance. However, this is consistent with the definition of model compression: losing less performance as possible during compression.

We argue that for industrial applications, it is often a trade-off between high prediction performance and speed/memory consumption. In this area, it is worth considering using Theseus Compression. For academia, primarily focusing on pushing the prediction performance boundaries, the downside of reduced prediction performance can outweigh the smaller model size and speed.

6.2 Future Work

Throughout this thesis, several aspects arose that are worth future research. Some are ideas to improve TC's performance or increase the compression process' speed. Others complement the in-depth evaluation.

Multi-GPU compression Pre-training [19, 45, 55, 56, 7], fine-tuning the LM of pre-trained models, or compression [60] is usually performed on a multi-GPU setup to speed up the training/compression time. However, all our experiments are based on a single GPU setup. Technically, there is no reason why the `*Theseus` models should not support multi-GPU compression because they are based on `transformer` classes. Nevertheless, benchmarking a distributed compression would provide evidence for or disprove this assumption.

Early stopping for the compression process We stop the compression process after 50 epochs for each of our experiments. Using *early stopping*, which is a regularization technique, potentially increases the models' performance. Early stopping ends the compression process, if the test/validation error starts to increase, which is a typical overfitting indicator. However, during the compression based on NOHATE and MIMIC-III, the model's perplexity does not start to increase. Thus, early stopping would compress the models for some more epochs, which leads to better perplexity. A distinct disadvantage is that the maximum number of compression steps is not known upfront and using a curriculum replacement scheduler gets more challenging because it is not possible to define the $ratio\ of\ replacement\ changes$ hyperparameter.

Experiment with Next Sentence and Sentence Order Prediction The pre-training procedure for BERT consists of two objectives: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). We adapt TC that it compresses the models during MLM and omit NSP. Some papers question whether NSP is the best objective to improve the model's performance [45, 40]. Thus, Lan et al. [40] proposed the Sentence Order Prediction (SOP) objective as a replacement. Gaining insights into whether adding the NSP or SOP objective to the compression process is beneficial for its performance is valuable for further improving TC.

Visually comparing predecessor and successor The goal of the qualitative error analysis is to understand whether TC transfers the predecessors' knowledge into the successor. Van Aken et al. [2] proposed a layer-wise visualization of BERT's hidden states to get insights into its reasoning process. Adapting this idea to compare the predecessor and successor models directly reveals if TC compresses the model. If so, they should show a similar reasoning process.

Apply TC on other model architectures The last two ideas for future work are similar to or already proposed by Xu et al. [78]. Technically, TC's single constraint is that the input and output sizes of a successor module have to match the predecessor module's sizes (see 2.2.4). This means it is not restricted to BERT or the Transformer architecture. It would be interesting to see TC's performance on other architectures.

Replace encoders with non-transformer based layers It is known that different parts of BERT learn specific aspects [39]. Thus, it would be interesting to explore if specialized layers used as successor modules can increase the compressed model's prediction performance.

Bibliography

[1] Betty van Aken, Julian Risch, Ralf Krestel, and Alexander Löser. "Challenges for Toxic Comment Classification: An In-Depth Error Analysis". In: *Proceedings of the 2nd Workshop on Abusive Language Online (ALW@EMNLP)*. 2018, pp. 33–42. DOI: 10.18653/v1/w18-5105.

[2] Betty van Aken, Benjamin Winter, Alexander Löser, and Felix A. Gers. "How Does BERT Answer Questions?: A Layer-Wise Analysis of Transformer Representations". In: *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*. 2019, pp. 1823–1832. DOI: 10.1145/3357384.3358028.

[3] Arndt Allhorn. "Transfer Learning for Hate Speech Detection". MA thesis. Beuth Hochschule für Technik Berlin, 2020.

[4] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: *CoRR* abs/1607.06450 (2016). arXiv: 1607.06450. URL: http://arxiv.org/abs/1607.06450.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *3rd International Conference on Learning Representations (ICLR)*. 2015. arXiv: 1409.0473. URL: http://arxiv.org/abs/1409.0473.

[6] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. "A Neural Probabilistic Language Model". In: *Advances in Neural Information Processing Systems 13 (NIPS)*. 2000, pp. 932–938. URL: http://papers.nips.cc/paper/1839-a-neural-probabilistic-language-model.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[8] Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. "Model Compression". In: *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2006, pp. 535–541. DOI: 10.1145/1150402.1150464.

[9] Daoyuan Chen, Yaliang Li, Minghui Qiu, Zhen Wang, Bofang Li, Bolin Ding, Hongbo Deng, Jun Huang, Wei Lin, and Jingren Zhou. "AdaBERT: Task-Adaptive BERT Compression with Differentiable Neural Architecture Search". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI).* 2020, pp. 2463–2469. DOI: `10.24963/ijcai.2020/341`.

[10] Yew Ken Chia, Sam Witteveen, and Martin Andrews. "Transformer to CNN: Label-scarce Distillation for Efficient Text Classification". In: *CoRR* abs/1909.03508 (2019). arXiv: `1909.03508`. URL: `http://arxiv.org/abs/1909.03508`.

[11] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).* 2014, pp. 1724–1734. DOI: `10.3115/v1/d14-1179`.

[12] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations". In: *Advances in Neural Information Processing Systems 28 (NIPS).* 2015, pp. 3123–3131. URL: `http://papers.nips.cc/paper/5647-binaryconnect-training-deep-neural-networks-with-binary-weights-during-propagations`.

[13] Andrew M. Dai and Quoc V. Le. "Semi-supervised Sequence Learning". In: *Advances in Neural Information Processing Systems 28 (NIPS).* 2015, pp. 3079–3087. URL: `http://papers.nips.cc/paper/5949-semi-supervised-sequence-learning`.

[14] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. "Transformer-XL: Attentive Language Models beyond a Fixed-Length Context". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL).* 2019, pp. 2978–2988. DOI: `10.18653/v1/p19-1285`.

[15] Fahim Dalvi, Hassan Sajjad, Nadir Durrani, and Yonatan Belinkov. "Exploiting Redundancy in Pre-trained Language Models for Efficient Transfer Learning". In: *CoRR* abs/2004.04010 (2020). arXiv: `2004.04010`. URL: `https://arxiv.org/abs/2004.04010`.

[16] Thomas Davenport and Ravi Kalakota. "The Potential for Artificial Intelligence in Healthcare". In: *Future Healthcare Journal* 6.2 (2019), pp. 94–98. DOI: `10.7861/futurehosp.6-2-94`.

[17] Li Deng and Yang Liu, eds. *Deep Learning in Natural Language Processing*. Springer Singapore, 2018. ISBN: 978-981-10-5209-5.

[18] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. "Predicting Parameters in Deep Learning". In: *Advances in Neural Information Processing Systems 26 (NIPS).* 2013, pp. 2148–2156. URL: `http://papers.nips.cc/paper/5025-predicting-parameters-in-deep-learning`.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT).* Minneapolis, Minnesota, 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[20] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. “HAWQ: Hessian AWare Quantization of Neural Networks With Mixed-Precision”. In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV).* 2019, pp. 293–302. DOI: 10.1109/ICCV.2019.00038.

[21] Angela Fan, Edouard Grave, and Armand Joulin. “Reducing Transformer Depth on Demand with Structured Dropout”. In: *8th International Conference on Learning Representations (ICLR).* 2020. URL: https://openreview.net/forum?id=SylO2yStDr.

[22] Jonathan Frankle and Michael Carbin. “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks”. In: *7th International Conference on Learning Representations (ICLR).* 2019. URL: https://openreview.net/forum?id=rJl-b3RcF7.

[23] Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Deming Chen, Marianne Winslett, Hassan Sajjad, and Preslav Nakov. “Compressing Large-Scale Transformer-Based Models: A Case Study on BERT”. In: *CoRR* abs/2002.11985 (2020). arXiv: 2002.11985. URL: https://arxiv.org/abs/2002.11985.

[24] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O’Reilly Media, 2019. ISBN: 978-1-492-03261-8.

[25] Mitchell A. Gordon, Kevin Duh, and Nicholas Andrews. “Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning”. In: *Proceedings of the 5th Workshop on Representation Learning for NLP (RepL4NLP@ACL).* 2020, pp. 143–155. URL: https://www.aclweb.org/anthology/2020.repl4nlp-1.18/.

[26] Mitchell A. Gordon, Kevin Duh, and Nicholas Andrews. “Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning”. In: *Proceedings of the 5th Workshop on Representation Learning for NLP (RepL4NLP@ACL).* 2020, pp. 143–155. URL: https://www.aclweb.org/anthology/2020.repl4nlp-1.18/.

[27] Tim Graf and Luca Salini. “bertZH at GermEval 2019: Fine-Grained Classification of German Offensive Language using Fine-Tuned BERT”. In: *Proceedings of the 15th Conference on Natural Language Processing (KONVENS).* 2019. URL: https://corpora.linguistik.uni-erlangen.de/data/konvens/proceedings/papers/germeval/Germeval_Task_2_2019_paper_15.BERTZH.pdf.

[28] Fu-Ming Guo, Sijia Liu, Finlay S. Mungall, Xue Lin, and Yanzhi Wang. “Reweighted Proximal Pruning for Large-Scale Language Representation”. In: *CoRR* abs/1909.12486 (2019). arXiv: 1909.12486. URL: http://arxiv.org/abs/1909.12486.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[30] Thomas L Higgins, Daniel Teres, Wayne S Copes, Brian H Nathanson, Maureen Stark, and Andrew A Kramer. “Assessing contemporary Intensive Care Unit Outcome: An Updated Mortality Probability Admission Model (MPM0-III)”. In: *Critical Care Medicine* 35.3 (2007), pp. 827–835.

[31] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. “Distilling the Knowledge in a Neural Network”. In: *CoRR* abs/1503.02531 (2015). arXiv: `1503.02531`. URL: `http://arxiv.org/abs/1503.02531`.

[32] Jeremy Howard and Sebastian Ruder. “Universal Language Model Fine-tuning for Text Classification”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL).* 2018, pp. 328–339. DOI: `10.18653/v1/P18-1031`.

[33] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2018, pp. 2704–2713. DOI: `10.1109/CVPR.2018.00286`.

[34] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. “TinyBERT: Distilling BERT for Natural Language Understanding”. In: *CoRR* abs/1909.10351 (2019). arXiv: `1909.10351`. URL: `http://arxiv.org/abs/1909.10351`.

[35] Alistair E.W. Johnson, Tom J. Pollard, Lu Shen, Li-wei H. Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G. Mark. “MIMIC-III, A Freely Accessible Critical Care Database”. In: *Scientific Data* 3.1 (2016), p. 160035. ISSN: 2052-4463. DOI: `10.1038/sdata.2016.35`.

[36] Karen Sparck Jones. “Natural Language Processing: A Historical Review”. In: *Current Issues in Computational Linguistics: In Honour of Don Walker.* 1994, pp. 3–16. ISBN: 978-0-7923-2998-5.

[37] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. “SpanBERT: Improving Pre-training by Representing and Predicting Spans”. In: *Trans. Assoc. Comput. Linguistics* 8 (2020), pp. 64–77. URL: `https://transacl.org/ojs/index.php/tacl/article/view/1853`.

[38] Daniel Jurafsky and James H. Martin. *Speech and Language Processing - An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice Hall, 2000. ISBN: 978-0-13-095069-7.

[39] Olga Kovaleva, Alexey Romanov, Anna Rogers, and Anna Rumshisky. “Revealing the Dark Secrets of BERT”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP).* 2019, pp. 4364–4373. DOI: `10.18653/v1/D19-1445`.

[40] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. “ALBERT: A Lite BERT for Self-supervised Learning of Language Representations”. In: *8th International Conference on Learning Representations (ICLR).* 2020. URL: `https://openreview.net/forum?id=H1eA7AEtvS`.

[41] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. "BioBERT: A Pre-trained Biomedical Language Representation Model for Biomedical Text Mining". In: *Bioinform.* 36.4 (2020), pp. 1234–1240. DOI: 10.1093/bioinformatics/btz682.

[42] Min Lin, Qiang Chen, and Shuicheng Yan. "Network In Network". In: *CoRR* abs/1312.4400 (2014). arXiv: 1312.4400. URL: http://arxiv.org/abs/1312.4400.

[43] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. "A Structured Self-attentive Sentence Embedding". In: *CoRR* abs/1703.03130 (2017). arXiv: 1703.03130. URL: http://arxiv.org/abs/1703.03130.

[44] Linqing Liu, Huan Wang, Jimmy Lin, Richard Socher, and Caiming Xiong. "Attentive Student Meets Multi-Task Teacher: Improved Knowledge Distillation for Pretrained Models". In: *CoRR* abs/1911.03588 (2019). arXiv: 1911.03588. URL: http://arxiv.org/abs/1911.03588.

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

[46] Thang Luong, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP).* 2015, pp. 1412–1421. DOI: 10.18653/v1/d15-1166.

[47] Christopher Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing.* MIT press, 1999. ISBN: 978-0262133609.

[48] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008. ISBN: 978-0521865715.

[49] Ben J. Marafino, Miran Park, Jason M. Davies, Robert Thombley, Harold S. Luft, David C. Sing, Dhruv S. Kazi, Colette DeJong, W. John Boscardin, Mitzi L. Dean, and R. Adams Dudley. "Validation of Prediction Models for Critical Care Outcomes Using Natural Language Processing of Electronic Health Record Data". In: *JAMA Network Open* 1.8 (2018), e185097–e185097. DOI: 10.1001/jamanetworkopen.2018.5097.

[50] Paul Michel, Omer Levy, and Graham Neubig. "Are Sixteen Heads Really Better than One?" In: *Advances in Neural Information Processing Systems 32 (NeurIPS).* 2019, pp. 14014–14024. URL: http://papers.nips.cc/paper/9551-are-sixteen-heads-really-better-than-one.

[51] Mainack Mondal, Leandro Araújo Silva, and Fabrício Benevenuto. "A Measurement Study of Hate Speech in Social Media". In: *Proceedings of the 28th ACM Conference on Hypertext and Social Media (HT).* 2017, pp. 85–94. DOI: 10.1145/3078714.3078723.

[52] Andrei Paraschiv and Dumitru-Clementin Cercel. “UPB at GermEval-2019 Task 2: BERT-Based Offensive Language Classification of German Tweets”. In: *Proceedings of the 15th Conference on Natural Language Processing (KONVENS)*. 2019. URL: https://corpora.linguistik.uni-erlangen.de/data/konvens/proceedings/papers/germeval/Germeval_Task_2_2019_paper_9.UPB.pdf.

[53] Matthew E. Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. “Semi-supervised Sequence Tagging with Bidirectional Language Models”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*. 2017, pp. 1756–1765. DOI: 10.18653/v1/P17-1161.

[54] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. “Deep Contextualized Word Representations”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. 2018, pp. 2227–2237. DOI: 10.18653/v1/n18-1202.

[55] Alec Radford. “Improving Language Understanding by Generative Pre-Training”. In: (2018).

[56] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. “Language Models are Unsupervised Multitask Learners”. In: (2019).

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

[58] Julian Risch, Anke Stoll, Marc Ziegele, and Ralf Krestel. “hpiDEDIS at GermEval 2019: Offensive Language Identification using a German BERT model”. In: *Proceedings of the 15th Conference on Natural Language Processing (KONVENS)*. 2019. URL: https://corpora.linguistik.uni-erlangen.de/data/konvens/proceedings/papers/germeval/Germeval_Task_2_2019_paper_10.HPIDEDIS.pdf.

[59] Mohammed Saeed, Mauricio Villarroel, Andrew T. Reisner, Gari Clifford, Li-Wei Lehman, George Moody, Thomas Heldt, Tin H. Kyaw, Benjamin Moody, and Roger G. Mark. “Multiparameter Intelligent Monitoring in Intensive Care II: A Public-access Intensive Care Unit Database”. In: *Critical Care Medicine* 39.5 (2011), pp. 952–960. DOI: 10.1097/CCM.0b013e31820a92c6.

[60] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter”. In: *CoRR* abs/1910.01108 (2019). arXiv: 1910.01108. URL: http://arxiv.org/abs/1910.01108.

[61] Alessandro Seganti, Helena Sobol, Iryna Orlova, Hannam Kim, Jakub Staniszewski, Tymoteusz Krumholc, and Krystian Koziel. “NLPR@SRPOL at SemEval-2019 Task 6 and Task 5: Linguistically Enhanced Deep Learning Offensive Sentence Classifier”. In: *Proceedings of the 13th International Workshop on Semantic Evaluation (SemEval@NAACL-HLT)*. 2019, pp. 712–721. DOI: 10.18653/v1/s19-2126.

[62] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. “Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT”. In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence.* 2020, pp. 8815–8821. URL: `https://aaai.org/ojs/index.php/AAAI/article/view/6409`.

[63] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *3rd International Conference on Learning Representations (ICLR).* 2015. arXiv: `1409.1556`. URL: `http://arxiv.org/abs/1409.1556`.

[64] Marina Sokolova and Guy Lapalme. “A Systematic Analysis of Performance Measures for Classification Tasks”. In: *Inf. Process. Manag.* 45.4 (2009), pp. 427–437. DOI: `10.1016/j.ipm.2009.03.002`.

[65] Emma Strubell, Ananya Ganesh, and Andrew McCallum. “Energy and Policy Considerations for Deep Learning in NLP”. In: *Proceedings of the 57th Conference of the Association for Computational Linguistics (ACL).* 2019, pp. 3645–3650. DOI: `10.18653/v1/p19-1355`.

[66] Julia Maria Struß, Melanie Siegel, Josef Ruppenhofer, Michael Wiegand, and Manfred Klenner. “Overview of GermEval Task 2, 2019 Shared Task on the Identification of Offensive Language”. In: *Proceedings of the 15th Conference on Natural Language Processing (KONVENS).* 2019. URL: `https://corpora.linguistik.uni-erlangen.de/data/konvens/proceedings/papers/germeval/GermEvalSharedTask2019Iggsa.pdf`.

[67] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. “Patient Knowledge Distillation for BERT Model Compression”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP).* 2019, pp. 4322–4331. DOI: `10.18653/v1/D19-1441`.

[68] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. “MobileBERT: A Compact Task-Agnostic BERT for Resource-Limited Devices”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL).* 2020, pp. 2158–2170. URL: `https://www.aclweb.org/anthology/2020.acl-main.195/`.

[69] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *Advances in Neural Information Processing Systems 27 (NIPS).* 2014, pp. 3104–3112. URL: `http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks`.

[70] Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. “Distilling Task-Specific Knowledge from BERT into Simple Neural Networks”. In: *CoRR* abs/1903.12136 (2019). arXiv: `1903.12136`. URL: `http://arxiv.org/abs/1903.12136`.

[71] Wilson L. Taylor. ““Cloze Procedure”: A New Tool for Measuring Readability”. In: *Journalism Quarterly* 30.4 (1953), pp. 415–433. DOI: `10.1177/107769905303000401`.

[72] A. M. Turin. “Computing Machinery and Intelligence”. In: *Mind* LIX.236 (1950), pp. 433–460. DOI: `10.1093/mind/LIX.236.433`.

[73] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30 (NIPS)*. 2017, pp. 5998–6008. URL: http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

[74] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding". In: *International Conference on Learning Representations (ICLR)*. 2019. URL: https://openreview.net/forum?id=rJ4km2R5t7.

[75] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. "Training Deep Neural Networks with 8-bit Floating Point Numbers". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2018, pp. 7686–7695. URL: http://papers.nips.cc/paper/7994-training-deep-neural-networks-with-8-bit-floating-point-numbers.

[76] Terry Winograd. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*. Tech. rep. Massachusetts Institute of Technology, 1971.

[77] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *CoRR* abs/1609.08144 (2016). arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144.

[78] Canwen Xu, Wangchunshu Zhou, Tao Ge, Furu Wei, and Ming Zhou. "BERT-of-Theseus: Compressing BERT by Progressive Module Replacing". In: *CoRR* abs/2002.02925 (2020). arXiv: 2002.02925. URL: https://arxiv.org/abs/2002.02925.

[79] Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. "Balanced Sparsity for Efficient DNN Inference on GPU". In: *The Thirty-Third AAAI Conference on Artificial Intelligence*. 2019, pp. 5676–5683. DOI: 10.1609/aaai.v33i01.33015676.

[80] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. "Q8BERT: Quantized 8Bit BERT". In: *CoRR* abs/1910.06188 (2019). arXiv: 1910.06188. URL: http://arxiv.org/abs/1910.06188.

[81] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. "LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks". In: *15th European Conference on Computer Vision (ECCV)*. Vol. 11212. Lecture Notes in Computer Science. 2018, pp. 373–390. ISBN: 978-3-030-01237-3.

[82] Sanqiang Zhao, Raghav Gupta, Yang Song, and Denny Zhou. "Extreme Language Model Compression with Optimal Subwords and Shared Projections". In: *CoRR* abs/1909.11687 (2019). arXiv: 1909.11687. URL: http://arxiv.org/abs/1909.11687.