Ludwig-Maximilians-Universität München
Department of Statistics

Master's Thesis

Benchmarking Down-Scaled Versions of Large Pre-Trained Language Models

Patrick Schulze

supervised by
Prof. Dr. Christian Heumann, Matthias Aßenmacher

December 17, 2020

Declaration of Authorship

I confirm that this Master's thesis is my own work and I have documented all sources and material used.

This thesis was not previously presented to another examination board and has not been published.

Signed: 

Date: 17.12.2020

Abstract

The aim of this thesis is to benchmark different pre-trained NLP systems. The performance of pre-trained NLP systems is determined by many different factors. Since state-of-the-art systems are often trained under vastly different conditions, it is therefore difficult to ascribe a good performance of a system to a specific component. We attempt to provide a more informative picture by systematically evaluating the effect of various factors.

Since automatically searching over the full systems is too costly, we benchmark downscaled versions of the original systems. Specifically, we train BERT-style, GPT-2-style and RoBERTa-style systems for different shape parameters and model sizes. Furthermore, we vary the number of training steps and the batch size. We can reproduce several results from previous studies, in particular the importance of model size compared to other factors, such as the number of training steps or the exact shape. Based on our observations, we finally scale up several systems.

Contents

List of Figures

List of Tables

List of Abbreviations

Architectures & Frameworks

CBOW Continuous Bag-of-words, p. 22

GAN Generative Adversarial Network, p. 21

LBL Log-bilinear Language Model, p. 22

LSTM Long Short-term Memory, pp. 17, 23, 24

RNN Recurrent Neural Network, pp. 9, 23, 24, 29

Seq2Seq Sequence To Sequence, pp. 24, 30, 43

Training Objectives

CTL Contrastive Learning, pp. 16, 21, 22, 50

DAE Denoising Autoencoding, pp. 16, 18

LM Language Modeling, pp. 16–18, 17, 19, 20, 24, 30, 47, 52, 58–60, 74, 83, 88

MLM Masked Language Modeling, pp. 16, 18–20, 22, 53, 57–59, 72, 88

MTL Multi-task Learning, pp. 16, 25, 26

NCE Noise-contractive Estimation, pp. 21, 22

NEG Negative Sampling, p. 22

NSP Next Sentence Prediction, pp. 19, 22, 53, 58, 59, 72, 75

PLM Permutation Language Modeling, p. 18

RTD Replaced Token Detection, p. 21

SOP Sentence Order Prediction, pp. 19, 22

TLM Translation Language Modeling, p. 19

Metrics

Acc Accuracy, p. 11

MCC Matthew's Correlation Coefficient, pp. 12, 64

Benchmarks

GLUE General Language Understanding Evaluation, pp. 3, 19, 63

RACE Large-scale ReAding Comprehension Dataset From Examinations, p. 19

SQuAD Stanford Question Answering Dataset, pp. 19, 51, 53, 65

General Tasks and Applications

IR Information Retrieval, p. 10

MT Machine Translation, pp. 10, 37, 38, 83

NER Named Entity Recognition, p. 23

NLI Natural Language Inference, pp. 10, 26

NMT Neural Machine Translation, pp. 15, 24

POS Part-of-speech Tagging, p. 23

QA Question Answering, pp. 10, 18, 23, 49

GLUE Tasks

CoLA Corpus Of Linguistic Acceptability, p. 64

MNLI Multi-Genre Natural Language Inference, p. 65

MRPC Microsoft Research Paraphrase Corpus, p. 64

QNLI Question-answering NLI, p. 65

QQP Quora Question Pairs, p. 65

RTE Recognizing Textual Entailment, p. 66

SST-2 Stanford Sentiment Treebank, p. 64

STS-B Semantic Textual Similarity Benchmark, p. 65

WNLI Winograd NLI, p. 66

Pre-Trained NLP Systems

ALBERT A Lite BERT, pp. 19, 22, 24

BERT Bidirectional Encoder Representations From Transformers, pp. 19, 20, 22, 24, 53

DistilBERT A Distilled Version Of BERT, p. 19

ELECTRA Efficiently Learning An Encoder That Classifies Token Replacements Accurately, pp. 21, 22, 24

ELMo Embeddings From Language Models, pp. 17, 18, 20, 23

GloVe Global Vectors For Word Representation, p. 23

GPT Generative Pre-trained Transformer, pp. 17, 24

GPT-2 Generative Pre-trained Transformer 2, p. 17

GPT-3 Generative Pre-trained Transformer 3, pp. 17, 46

RoBERTa Robustly Optimized BERT Approach, pp. 19, 24

T5 Text-to-Text Transfer Transformer, p. 24

Transformer-XL Transformer-XL, p. 17

Word2Vec Word2Vec, pp. 9, 22, 23

XLNet XLNet, pp. 17, 18, 24

General

CV Computer Vision, pp. 15, 37, 59

GELU Gaussian Error Linear Unit, pp. 48, 51, 72

Part I

Introduction and Theory

Chapter 1

Introduction

The introduction of the Transformer architecture [117] together with the application of transfer learning approaches [113] has led to major advances in NLP in recent years. While many different lines of research exist, the most attention is generally paid to the largest systems, presumably because these systems often reach new state of the art. The current trend is to scale up such systems to ever new orders of magnitude: 213M parameters in the original Transformer [117], 300M parameters in BERT [25], 1.5B parameters in OpenAI GPT-2 [86] and recently 175B parameters in OpenAI GPT-3 [11]. The different systems are usually pre-trained on corpora of widely varying sizes, for a different number of training steps and with different batch sizes. At the same time, new systems often apply fundamentally different methods, such as using a different pre-training objective or modified architectural hyperparameters. While altering multiple components simultaneously can help achieve state of the art, which is an important endeavor, it is difficult to disentangle the effects of the various factors. Though there exist various ablation studies, these studies often show only a small excerpt from the broad spectrum of experimental opportunities and are thus not suitable for providing a comprehensive picture.

In contrast, the benchmarking study presented in Part II of this thesis has the purpose of systematically investigating the effect of a broad variety of different factors. In particular, we study different pre-training objectives, different architectural hyperparameters of the Transformer and the effect of model size, number of training steps and batch size, mainly on the downstream performance on the *General Language Understanding Evaluation* (GLUE) benchmark [120]. A more detailed outline of our objectives is given in Chapter 6, after having established relevant methods, such as transfer learning and the Transformer architecture.

Outline

This thesis is divided into two parts. In this part, we start by providing a theoretical background of important methods used in modern NLP, while the second part is dedicated to our benchmarking study.

In Chapter 2 we start by discussing several preliminary steps, such as pre-processing, and provide information on general concepts relevant in modern NLP. We also define several NLP tasks and evaluation metrics that are relevant to our experiments in Part II. Chapter 3 introduces the concept of transfer learning and discusses how and why transfer learning has been used in NLP. In particular, we categorize modern NLP systems based on a taxonomy from different perspectives, such as pre-training objectives and methods of transfer. In Chapter 4 we describe different architectures of neural NLP systems, with a focus on the Transformer. In Chapter 5, we close Part I by presenting several state-of-the-art NLP systems that are relevant to our benchmarking study in Part II.

We start Part II by discussing related work and by defining the specific objectives of our benchmarking study. Furthermore, we introduce the pre-training corpus and provide a description of the GLUE Benchmark, which will be used to fine-tune and evaluate different systems. In Chapter 7 we then present our experiments. We start with some preparatory steps, such as discussing important definitions and the specific format of the inputs during pre-training, before presenting our results in section 7.5. Finally, we address several limitations, propose topics for further research and conclude our discussion in Chapter 8.

Chapter 2

Preliminaries

In this chapter we shall establish some basic concepts that underly all modeling approaches discussed throughout this thesis. We start with pre-processing and proceed chronologically by describing how the resulting inputs are processed by the NLP systems that we consider. Furthermore, we introduce several NLP tasks and metrics.

2.1 Pre-Processing

Before NLP systems can be trained and evaluated on text, the text must first be converted into a usable format. To achieve this, several pre-processing steps are necessary, which are briefly described in the following.

2.1.1 Normalization

Normalization is the first step of pre-processing. The aim of normalization is to convert text into a standard format, which is necessary because real-word text can appear in many different forms. Well-known examples of normalization are lowercasing and stripping exceeding whitespace.

Normalization has usually been implemented as hand-crafted rules. In contrast, in many modern NLP applications normalization is conducted as a standardized procedure, which allows for better reproducibility [53]. For instance, a normalization step that is usually performed is Unicode normalization, which is the conversion of different Unicode sequences representing the same characters into unique code point sequences. In the common tokenization libraries used by most state-of-the-art NLP systems, normalization is automatically performed before the actual tokenization, which will be discussed in

the next section. For example, the *SentencePiece* library [53] performs Unicode NFKC normalization by default.

2.1.2 Tokenization

Tokenization, which is a fundamental pre-processing step in almost all NLP applications, refers to the task of splitting text into smaller units, so-called *tokens*. Tokenization can be divided into two stages. First, a *token vocabulary* $\mathcal{V}$ is generated from the training data of an NLP system. Subsequently, the training data is encoded with tokens from the previously generated vocabulary $\mathcal{V}$. The same token vocabulary must also be used for each new input of the trained system.

Formally, each token $x_i \in \mathcal{V}$ is represented as a unique positive integer. Thus, the token vocabulary can be written as $\mathcal{V} = \{1, \dots, V\}$, where V is the vocabulary size. All systems we consider take a probabilistic approach. That is, an observed sequence of tokens $\boldsymbol{x} = [x_1, \dots, x_n]$ is assumed to consist of realizations of discrete random variables, taking on values from $\mathcal{V}$. We adopt this notion in the remainder of this thesis.

Most state-of-the-art systems construct their inputs by using different variants of *subword tokenization*. To demonstrate the benefits of subword tokenization we start by considering two alternative methods.

Word-level Tokenization

A natural approach to split text into tokens is to use spaces between words as split points and identify the token vocabulary with the set of distinct words occurring in the text. However, this approach has several shortcomings:

- **Out-of-vocabulary words** A new text to which a trained model is applied to perform a language task might contain out-of-vocabulary words, i.e., words that are not contained in the training data. Such words cannot be evaluated by the language model. Typical examples of out-of-vocabulary words when using word-level tokenization are neologisms and proper names.

- **Large vocabulary size** Storing each word separately results in a large vocabulary size. While using techniques that reduce words to their root forms can alleviate this problem, applying these techniques requires a separate pre-processing step. Furthermore, such techniques are usually not language agnostic.

- **Languages without spaces** While languages such as German or English separate words by whitespaces, other languages, such as Chinese, have no spaces between words. Word-level tokenization for these languages is therefore infeasible.
- **Suboptimal encoding of meaning** The meaning of a word can often be derived from smaller units contained in that word. When each word is encoded separately, it is more difficult to share the meaning of these small units across their various occurrences in the text.

Character-level Tokenization

At the other end of the spectrum, the text could be split into individual characters or, even further, into UTF-8 bytes. However, compared to word-level tokenization, processing linguistic meaning when using such small units is even more difficult. Another problem is increased computational complexity due to the large number of units per sentence. In Chapter 4 we show that Transformer-based systems, which make use of a so-called *attention mechanism* [5], are especially sensitive to such large numbers of tokens. As Radford et al. [86] mention, byte-level language models are currently not competitive with word-level language models.

Subword Tokenization

Subword tokenization is a middle ground between word-level and character/byte-level tokenization. As the name suggests, this technique involves dividing words into smaller units. While there exist many different variants of subword tokenization, the general idea is to split the text into units based on their frequency or likelihood within a probabilistic model. Such approaches generally produce a subword vocabulary consisting of full words that carry meaning on the one hand, and commonly occurring small units, such as affixes and single characters, on the other hand. This high degree of granularity allows for efficient encoding of a text, so that subword tokenization does not require a large vocabulary. Crucially, the subword vocabulary can be used to represent out-of-vocabulary words.

There are a variety of algorithms for dividing text into subwords, and often different NLP systems use different subword algorithms. *Byte pair encoding* (BPE) [100] and two variants thereof, *WordPiece* [99] and *byte-level BPE* [86], begin by initializing a small vocabulary, which is incrementally expanded. These approaches can therefore be described as bottom-up approaches. In contrast to BPE and its variants, the *unigram*

approach [52], which is based on a probabilistic unigram model, starts with a large superset of the final vocabulary, which is iteratively pruned to a desired size. The unigram approach is part of the *SentencePiece* library [53]. We discuss BPE, WordPiece and byte-level BPE in Appendix A.1, as these tokenization methods will be used in our experiments.

2.2 Neural NLP

In the remainder of this paper, we will focus on methods based on neural networks. Over the last decade, neural networks have emerged as the predominant modeling architecture in NLP. The current generation of NLP systems is therefore often described as *neural NLP*. Neural networks have a number of advantages over more traditional approaches and have therefore largely replaced methods from so-called *statistical* and *symbolic NLP*. For instance, as will be discussed in Chapter 3, the structure of neural networks is very suitable for transfer learning. Furthermore, neural networks allow for learning complex relationships. This is crucial in NLP, since there exist highly non-linear relationships in text [10]. Neural networks, in particular the Transformer architecture [117], are therefore a key driver of many breakthroughs in NLP in recent years.

2.3 Token Embeddings

When processing a sequence $\boldsymbol{x} = [x_1, \dots, x_n]$ with a neural network, each token of that sequence is first *embedded* in a vector space of embedding dimension H. This happens in the very first layer of the network, which is also called *embedding layer*.

Formally, for each token x_i, let $\mathbf{1h}(x_i)$ denote a one-hot vector of length V with index x_i set to 1. The *input embedding* $\boldsymbol{e}(x_i) \in \mathbb{R}^H$ of a token x_i is then obtained with a learnable linear transformation

$$\begin{aligned} \boldsymbol{e} \colon \mathcal{V} &\to \mathbb{R}^H \\ x_i &\mapsto \boldsymbol{W}_{\boldsymbol{e}}^T \mathbf{1h}(x_i), \end{aligned}$$

where $\boldsymbol{W}_{\boldsymbol{e}} \in \mathbb{R}^{V \times H}$ is the *embedding matrix*. Thus, $\boldsymbol{e}(x_i)$ simply corresponds to the (transposed) x_i-th row of $\boldsymbol{W}_{\boldsymbol{e}}$. As will be shown in Chapter 3, in modern NLP systems the input embedding is again used in the output layer of the network to calculate output probabilities over the token vocabulary. Using the same embedding for inputs and outputs has proved useful in neural NLP [83] and therefore has become the standard

approach [25, 117, 129].

2.4 Language Representation Learning

As Collobert and Weston [19] state, the primary goal of NLP is to obtain representations of language that are easy for computers to operate on. Learning representations to facilitate the extraction of useful information is in general known as *representation learning* [8]. Bengio et al. [8] argue that good representations should express general priors, which can help improve performance across a wide variety of tasks. Indeed, as will be discussed in Chapter 3, the transfer of general-purpose language representations to specific tasks has played a key role in NLP in recent years. There exist two broad categories of language representations, both of which consist in mapping text fragments into a series of real-valued vectors [84]:

Non-contextual representations Many prominent methods, such as Word2Vec [71], are based on learning so-called *word vector representations*. These approaches use large amounts of text data with the primary goal of learning the input embedding $\boldsymbol{e}(x_i)$ of each token x_i.[1] After training, each embedding serves as a static and hence *non-contextual representation* of the corresponding token. In a second stage, the learned embeddings can be used to represent the text input of a task-specific model, which often increases the performance of this model significantly when compared to starting with randomly initialized embeddings.

Contextualized representations A shortcoming of word vector representations is that a token x_i is always represented as $\boldsymbol{e}(x_i)$ regardless of the context it appears in. Therefore, the meaning of a text is often not properly reflected when encoded with the learned embeddings. In contrast, given an input sequence $\boldsymbol{x} = [x_1, \dots, x_n]$, the *contextualized representation* $\boldsymbol{z}_i$ of a token x_i is dynamically computed based on multiple tokens of the sequence $\boldsymbol{x}$. In more recent NLP systems, a trained *neural encoder* $\boldsymbol{z}$, such as a *recurrent neural network* (RNN) [96, 122] or a Transformer, outputs a sequence

$$\boldsymbol{z}(\boldsymbol{x}) = [\boldsymbol{z}_1, \dots, \boldsymbol{z}_n] \tag{2.4.1}$$

of contextualized representations $\boldsymbol{z}_i \in \mathbb{R}^H$ for each token x_i. There are many different possibilities to obtain such contextualized representations. For instance, in some cases each element $\boldsymbol{z}_i$ may depend on all tokens of the sequence $\boldsymbol{x}$, while in other cases the set

[1] As the name suggests, most word vector approaches simply use a word-level tokenization.

of tokens on which each representation may depend is restricted. We give an overview of different approaches in Chapter 3.

2.5 NLP Tasks

There exist many end applications of NLP, including for instance *machine translation* (MT), *information retrieval* (IR) and *dialogue systems*. Numerous sub-tasks, ranging from syntactic to semantic, have been identified to train and evaluate systems with respect to such applications. In the following, we focus on the task categories that are relevant to our experiments presented in Chapter 7. In addition, relevant pre-training tasks will be discussed in Chapter 3.

Sentiment analysis In *sentiment analysis*, subjective information is extracted from a given text by classifying its polarity. In most cases this polarity is binary (*positive* or *negative*) or ternary (*positive*, *negative*, or *neutral*). There exist different variants of sentiment analysis. In this paper, we will focus on sentence-level sentiment analysis. More challenging variants, such as aspect-level sentiment analysis, require determining the sentiment with respect to a particular target.

Natural language inference *Natural language inference* (NLI) consists in assessing a directional relation between text segments. The task is usually structured as as a classification problem, in which a model must predict whether a premise *entails*, *contradicts* or is *neutral* towards a hypothesis.

Question answering *Question answering* (QA) is a fine-grained variant of IR, which consists in retrieving information from data. Given a question, the task is to find specific answers, which typically can be inferred from an available document. The concrete answer format of the task may vary. For instance, in some scenarios, it is required to select the text span that contains the answer. In other cases, the answer must be selected from a set of predefined choices. Question answering intersects with other areas of NLP, such as *reading comprehension* or *common sense reasoning*.

Paraphrase detection & semantic textual similarity *Paraphrase detection* is the binary task of judging whether two text fragments are similar in meaning. *Semantic textual similarity* differs from paraphrase detection in that it assesses the *degree* of similarity between text fragments. While paraphrase and textual similarity are not considered natural NLP tasks, they have proved very useful to evaluate systems. Un-

derstanding semantic textual similarity is for instance relevant to applications such as MT and QA [13].

Note that most tasks can be structured as classification problems. There exist only few exceptions, such as textual similarity, which is a regression task. All tasks that we consider in Chapter 7 are furthermore sentence-level tasks, involving either single sentences or sentence pairs. A detailed description of these tasks is given in Chapter 6.

2.6 Evaluation of Model Performance

We close this chapter by briefly describing the general evaluation process in machine learning models and the metrics that are relevant to our experiments.

2.6.1 Training and Evaluation Process

A dataset is in general split into a training set, a validation set and a test set. A model is trained by minimizing the loss on the training set. The validation set can be used to evaluate the effect of different hyperparameters, as it provides an a-priori unbiased estimate of the performance of a model. Usually a specific hyperparameter configuration is then chosen based on the validation set performance. In this case, however, a-posteriori the validation set performance of the selected models is biased, because it is affected by the selection process. In order to obtain an unbiased estimate of the performance of a final model, it is therefore important to use a separate test set.

Regarding our experiments, note that we will generally evaluate model performance on the validation set. One reason for this is that we do not conduct any hyperparameter tuning. Another reason arises from a practical problem: the test set of the GLUE benchmark is not publicly available and submission of test set predictions is limited.

2.6.2 Metrics

As stated, many NLP tasks can be presented as classification tasks. In binary classification tasks, the performance of a model can be assessed on the basis of *true* and *false positives* (TP/FP), which give the number of predicted positives that were correct/incorrect, and similarly *true* and *false negatives* (TN/FP), which correspond to the correct/incorrect predicted negatives. The sum of these four quantities corresponds to the total number N of predicted observations.

		Actual class	
		P	**N**
Predicted class	**P**	**TP**	FP
	N	FN	**TN**

Figure 2.1: Confusion matrix for binary classification.

A very common measure is *accuracy* (Acc), which is the fraction of correctly classified observations:

$$\text{Acc} = \frac{TP + TN}{N}. \tag{2.6.1}$$

Accuracy is a suitable metric if the classes are balanced and the cost of misclassification is not disproportionately large for observations of either class. However, if one class is considerably smaller than the other, the fraction of incorrect predicted examples from the small class is not adequately reflected in the accuracy, which is especially problematic if there is a high cost associated with wrong classification of these examples.

An alternative metric for binary classification is the F_1-score, which is defined as the harmonic mean between *precision* P and *recall* R:

$$F_1 = \frac{2}{P^{-1} + R^{-1}} = 2\frac{P \cdot R}{P + R}, \qquad \text{where} \tag{2.6.2}$$

$$P = \frac{TP}{TP + FP}, \qquad R = \frac{TP}{TP + FN}. \tag{2.6.3}$$

The F_1-score thus gives equal weight to precision, which measures the fraction of correctly predicted examples out of all positive predictions, and recall, the fraction of positive examples that were correctly classified as positive. If either the precision or the recall is zero, the F_1-score is also zero. In all other cases, the F_1-score is greater than zero, with a maximum value of one if and only if both precision and recall are one.

In contrast to accuracy, the F_1-score depends on which class is defined as the positive class. The F_1-score is unsuitable if the number of negative examples is small compared to the number of positive examples. In particular, a classifier which simply assigns all examples to the positive class will achieve a high F_1-score (and a high accuracy) in such a scenario.

If the labels are imbalanced, the *Matthew's correlation coefficient* (MCC) [66], which is defined as

$$\text{MCC} = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(FP + FN)(TN + FP)(TN + FN)}}, \tag{2.6.4}$$

is more suitable than both the F_1-score and accuracy [82]. In particular, it is the only metric in binary classification which generates a high score if and only if the majority of examples from both the positive and the negative class are predicted correctly [15]. MCC ranges in $[-1, 1]$, taking on the left and the right boundary in case of perfect misclassification and perfect classification, respectively. The case $MCC = 0$ corresponds to a classifier which randomly assigns each observation to the positive class with probability $\frac{1}{2}$. If one of the four sums in the denominator is zero, MCC is undefined. It can be shown that the limiting value is $MCC = 0$ in these cases [15]. Note that for a 2×2 contingency table the MCC is related to the chi-square statistic:

$$|\text{MCC}| = \sqrt{\frac{\chi^2}{n}}. \tag{2.6.5}$$

In the binary case, the chi-square statistic can be used to test whether observed pairs of binary data are realizations of two independent random variables. The null hypothesis that the outcomes are independent is rejected if χ^2 is sufficiently large. This connection allows for an intuitive interpretation of MCC.

Chapter 3

Transfer Learning

This chapter gives an introduction to *transfer learning* [113], which is of central importance in modern NLP. It also serves as an overview of NLP systems that leverage ideas from transfer learning. In transfer learning, knowledge obtained by learning a *source task* $\mathcal{T}_S$ is used to learn a *target task* $\mathcal{T}_T$, which is also known as a *downstream task* in NLP. Learning the downstream task is sometimes referred to as *adaptation* [94]. The setting in which source and downstream tasks are the same is known as *transductive* transfer learning, whereas *inductive* transfer learning describes the situation in which the source task is different from the downstream task [78]. Neural networks are particularly suitable for transfer learning, because in principle both the architecture and the parameters from the source task can be used as an initialization point, better known as a *pre-trained system*, for the downstream task [116]. This allows for efficient transfer of the pre-trained system, such that pre-training a single system can often increase the performance on a large variety of downstream tasks. Pre-training is particularly useful to prevent overfitting, which can be crucial when training neural networks, since large neural networks sometimes overfit on small datasets [91].

Before transfer learning became popular in NLP it had already been adopted in other fields, such as *computer vision* (CV). Indeed, Mou et al. [75] presented evidence that transfer learning might not be very effective in NLP, however, focusing on transfer between supervised source and downstream tasks. In contrast, Howard and Ruder [44] demonstrated that the question is not *whether* transfer learning is applicable to NLP, but that instead it is essential *how* the knowledge is transferred and especially how this knowledge is obtained by pre-training on a source task. In particular, Howard and Ruder [44] proposed inductive transfer learning with a *general-domain* pre-training approach, leveraging large amounts of unlabeled data.

Unsupervised pre-training had in fact already been successfully combined with subsequent supervised NLP tasks such as *neural machine translation* (NMT), using pre-trained word vector representations as inputs of a task-specific target model [19]. Such *semi-supervised* approaches are especially suitable for NLP, because unlabeled text data is largely available on the web, whereas labeled data is scarce and generating labels is often expensive. It has in many cases indeed proved true that, by learning from large amounts of data on an unsupervised source task, performance on data-scarce downstream tasks can be improved drastically in NLP [25, 56, 62, 85, 129].

While word vector representations can be regarded as an early form of transfer learning in NLP, compared to later systems these techniques only leverage a very small amount of the information contained in the source data. Current state-of-the-art systems exploit the scalability of large neural networks, in most cases based on the Transformer architecture, to process huge amounts of unlabeled pre-training data. This paradigm is accompanied by an improved generalization of many shared downstream tasks and in particular by a shift from syntactic to more semantic tasks.

In the following, in section 3.1 we first discuss several pre-training objectives used in current NLP systems and subsequently provide a summary of different adaptation techniques in section 3.2. Furthermore, in sections 3.3 and 3.4 we give some information regarding two currently popular concepts related to transfer, *multi-task learning* (MTL) [12, 93] and *meta-learning* [11].

3.1 Unsupervised Pre-Training

We group different pre-training approaches of NLP systems into three categories: *language modeling* (LM) [7], which is autoregressive; *masked language modeling* (MLM) [25], which is closely related to *denoising autoencoding* (DAE) [118]; and *contrastive learning* (CTL) [3], which, in contrast to MLM, is based on discriminating corrupted samples from original inputs rather than reconstructing them.

3.1.1 Language Modeling

Given a sequence $\boldsymbol{x} = [x_1, \ldots, x_n]$ of tokens x_i, LM seeks to infer the probability $p(\boldsymbol{x})$. Forward LM originates from the autoregressive factorization

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i | \boldsymbol{x}_{<i}), \tag{3.1.1}$$

where $\boldsymbol{x}_{<i} := [x_0, x_1, x_2, \ldots, x_{i-1}]$ and x_0 is an added special token indicating the start of the sequence.[1] From this factorization, the problem can be reduced to estimating each conditional factor using a parametric model. In forward LM, the neural encoder produces a sequence of contextualized representations $\boldsymbol{z}_{\mathrm{LM}}(\boldsymbol{x}) = [\boldsymbol{z}_{\mathrm{LM}}(\boldsymbol{x}_{<1}), \ldots, \boldsymbol{z}_{\mathrm{LM}}(\boldsymbol{x}_{<n})]$, where each element may depend only on the previous tokens $\boldsymbol{x}_{<i}$. In Transformer-based NLP systems, the conditional probability of a token x_i is usually obtained by calculating a softmax score over the dot-products of the contextualized representations with the input embeddings:

$$p(x_i|\boldsymbol{x}_{<i};\boldsymbol{\theta}) = \frac{\exp\Big(\boldsymbol{z}_{\mathrm{LM}}(\boldsymbol{x}_{<i})^T \boldsymbol{e}(x_i)\Big)}{\sum_{x' \in \mathcal{V}} \exp\big(\boldsymbol{z}_{\mathrm{LM}}(\boldsymbol{x}_{<i})^T \boldsymbol{e}(x')\big)}. \tag{3.1.2}$$

The loss $\mathcal{L}_{\mathrm{LM}}(\boldsymbol{x}, \boldsymbol{\theta})$ of a sequence $\boldsymbol{x} = [x_1, \ldots, x_n]$ is defined as the cross entropy between the one-hot empirical distribution of training labels and the predicted conditional distribution:[2]

$$\mathcal{L}_{\mathrm{LM}}(\boldsymbol{x}, \boldsymbol{\theta}) = -\sum_{i=1}^{n} \log p(x_i|\boldsymbol{x}_{<i};\boldsymbol{\theta}). \tag{3.1.3}$$

The model parameters are estimated by minimizing the combined loss $\sum_{\boldsymbol{x} \in \mathcal{X}} \mathcal{L}_{\mathrm{LM}}(\boldsymbol{x}, \boldsymbol{\theta})$ over a corpus $\mathcal{X}$ of sequences. Contrary to forward LM, in backward LM the probability of a sequence is factorized into $p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i|\boldsymbol{x}_{>i})$, where similarly to forward LM we define $\boldsymbol{x}_{>i} := [x_{i+1}, x_{i+2}, \ldots, x_n, x_{n+1}]$ and x_{n+1} is added to mark the end of the sequence. In this case, the model is trained to predict the conditional probability of each token x_i when given future tokens $\boldsymbol{x}_{>i}$.

Systems An early approach of estimating discrete distributions over word sequences from large-scale text data was presented by Bengio et al. [7], employing a feed-forward neural network to predict word probabilities. Mikolov et al. [70] followed this general approach, but used a recurrent neural network, allowing for modeling longer dependencies within word sequences. In the context of semi-supervised learning, Dai and Le [23] proposed to use the weights of a recurrent pre-trained LM as an initialization for a supervised *long short-term memory* (LSTM). Howard and Ruder [44] refined this approach, introducing several new fine-tuning techniques. ELMo [81] combines multiple LSTMs, trained with forward and backward LM objectives, allowing for capturing bidirectional representations. Radford et al. [85] introduced GPT, a semi-supervised

[1] As x_0 is deterministic and constant, we have $p(\boldsymbol{x}) = p(\boldsymbol{x}|x_0)$ and thus Eq. (3.1.1) holds.

[2] Note that minimizing Eq. (3.1.3) is equivalent to maximizing the log-likelihood of the sequence.

system based on the Transformer architecture, which is pre-trained with a forward LM objective.[3] Transformer-XL [24] is another Transformer-based language model, which extends the context-size by introducing a recurrence mechanism between consecutive input segments. XLNet [129] employs a generalization of the classical LM objective by factorizing the probabilities of tokenized input sequences using random permutations while maintaining the autoregressive structure. DialoGPT [133] is a large-scale language response generation system with an architecture equivalent to GPT-2. Recently, GPT-3 [11], a 175B parameter system, was released.

Pros and cons First, LM is a *generative* approach. Hence language models can be used for generating new text, which can be directly leveraged in applications such as language response generation and proves useful in meta-learning, as will be demonstrated in section 3.4. It has been demonstrated that LM is a more effective pre-training objective than other objectives, such as translation [132]. LM is considered a difficult task, presumably requiring a system to learn about syntax and semantics [95]. Furthermore, as Howard and Ruder [44] argue, a pre-trained language model is well suited to the characteristics of most downstream tasks. However, this does not apply to all tasks, as a shortcoming of traditional language models is their inability of capturing bidirectional contexts. Devlin et al. [25] demonstrate that, especially in sentence-level tasks, such as QA, the autoregressive structure has a negative impact on model performance. Although there exist several bidirectional language models, such as ELMo [81], these models are based on a shallow concatenation of forward and backward language models, meaning that most parameters are in fact related to unidirectional objectives. A notable exception is XLNet [129], which uses a generalization of the classical LM objective known as *permutation language modeling* (PLM) [129]. XLNet masks weights during pre-training to preserve the autoregressive structure, which means that effectively only a small fraction of input tokens is used during pre-training. This approach increases the required amount of computation significantly.

3.1.2 Masked Language Modeling

MLM can be divided into two stages. In the first stage, a corrupted version of each original input sequence $\boldsymbol{x}$ is produced by randomly masking a fixed proportion of input tokens. The second stage consists of training the model to predict the masked tokens. This approach is similar to DAE, with the difference that DAE involves reconstructing

[3] GPT-2 and GPT-3, which build upon the original GPT, are also trained with a LM objective, but were not established in the classical semi-supervised context.

the entire input, whereas in MLM only the masked tokens are predicted. Specifically, for each sequence $\boldsymbol{x} = [x_1, \ldots, x_n]$ an index set $M = \{m_1, \ldots, m_k\} \subset \{1, \ldots, n\}$ is picked uniformly at random from the set of all possible index sets of length k.[4] The corrupted version of $\boldsymbol{x}$, denoted as $\boldsymbol{x}^{\text{masked}}$, is then obtained by replacing all tokens x_i, $i \in M$, with a `[MASK]` token. Reconstructing the masked tokens corresponds to the task of predicting the joint conditional probability $p(x_{m_1}, \ldots, x_{m_k} | \boldsymbol{x}^{\text{masked}})$. Therefore, in contrast to the generative approach followed in autoregressive LM, no explicit density estimation is performed. The joint conditional probability is approximated with

$$p(x_{m_1}, \ldots, x_{m_k} | \boldsymbol{x}^{\text{masked}}) \approx \prod_{i \in M} p(x_i | \boldsymbol{x}^{\text{masked}}), \tag{3.1.4}$$

assuming that the masked tokens are conditionally independent. A Transformer-based system can then be trained to predict the univariate conditional probabilities as

$$p(x_i | \boldsymbol{x}^{\text{masked}}; \boldsymbol{\theta}) = \frac{\exp\Big(\boldsymbol{z}_{\text{MLM}}(\boldsymbol{x}^{\text{masked}})_i^\top e(x_i)\Big)}{\sum_{x' \in \mathcal{V}} \exp\Big(\boldsymbol{z}_{\text{MLM}}(\boldsymbol{x}^{\text{masked}})_i^\top e(x')\Big)}, \qquad \text{for all } i \in M, \tag{3.1.5}$$

where $\boldsymbol{z}_{\text{MLM}}$ is a neural encoder that maps an input sequence $\boldsymbol{x}$ to a sequence $\boldsymbol{z}_{\text{MLM}}(\boldsymbol{x}) = [\boldsymbol{z}_{\text{MLM}}(\boldsymbol{x})_1, \ldots, \boldsymbol{z}_{\text{MLM}}(\boldsymbol{x})_n]$ of contextualized representations. Note that each element $\boldsymbol{z}_{\text{MLM}}(\boldsymbol{x})_i$ depends on the full input sequence $\boldsymbol{x}$. Therefore, in contrast to LM, the contextualized representation of each token is bidirectional. The loss used in MLM is

$$\mathcal{L}_{\text{MLM}}(\boldsymbol{x}, \boldsymbol{\theta}) = -\sum_{i \in M} \log p(x_i | \boldsymbol{x}^{\text{masked}}), \tag{3.1.6}$$

which is an approximation of the expected cross entropy $\mathbb{E}[-\sum_{i \in \mathcal{M}} \log p(\boldsymbol{x}_i | \boldsymbol{x}^{\text{masked}})]$, where $\mathcal{M}$ denotes a random index set. Note that the independence assumption in (3.1.4) follows implicitly from the choice of this loss function. As in neural LM, the parameters are obtained by minimizing the combined loss $\sum_{\boldsymbol{x} \in \mathcal{X}} \mathcal{L}_{\text{MLM}}(\boldsymbol{x}, \boldsymbol{\theta})$.

Systems MLM, which is based on the *Cloze Task* [111], was introduced as the general pre-training approach of BERT [25], the first bidirectional Transformer. This novel approach led to significantly improved results on a broad spectrum of benchmark tasks. Devlin et al. [25] implemented a slightly more elaborate masking approach than described above and, in addition to MLM, used *next sentence prediction* (NSP) [25] as a second pre-training objective. We illustrate NSP and the masking approach of BERT in

[4] A common choice is $k = \lceil 0.15n \rceil$.

Chapter 5. Several changes to the original model have been proposed since the introduction of BERT. In RoBERTa, Liu et al. [62] removed the NSP objective and used different input formats with a modified masking pattern, as well as considerably larger batch sizes. These modifications improved the performance of BERT on GLUE [120], SQuAD 1.1 & 2.0 [89, 90] and RACE [54] significantly. Furthermore, Liu et al. [62] investigated the impact of the amount of pre-training data and the number of pre-training iterations, showing that increasing these factors leads to further performance gains. ALBERT [56] employs parameter reduction techniques, allowing for scaling the model by increasing the hidden layer size, and replaces NSP with *sentence order prediction* (SOP). These techniques further improved the overall performance compared to RoBERTa. However, as Lan et al. [56] noticed, though the parameter-reduction techniques allow for scaling the model while maintaining a manageable number of parameters, this benefit comes at increased computational costs. DistilBERT [98] leverages a technique called knowledge distillation, which consists in replacing the one-hot empirical training distribution in the MLM loss with the predicted distribution of a so-called teacher model, in this case BERT. Using a training objective that combines distillation with MLM, DistilBERT achieved similar results as BERT on the GLUE benchmark, but is considerably smaller and faster than the original model. In the field of cross-lingual language modeling, XLM [55] leverages different pre-training objectives: autoregressive LM, MLM, as well as a combination of MLM and an extension of MLM called *translation language modeling* (TLM). The results of Lample and Conneau [55] demonstrate that MLM, as well as the combination of MLM and TLM, outperform LM in cross-lingual tasks.

Pros and cons In contrast to shallow bidirectional approaches, such as ELMo, masked language models learn deep bidirectional contexts, because all parameters are related to a bidirectional training objective. As Devlin et al. [25] demonstrated, the ability to learn deep bidirectional contexts is the key driver of the large performance gains that BERT achieved on a wide range of NLP tasks. However, while more effective than unidirectional language models, MLM requires increased computational costs, since only a small percentage of tokens is used per input sequence. A second problem is the mismatch between the altered pre-training data, containing artificial `[MASK]` tokens, and the data used in the subsequent supervised stage. Furthermore, in contrast to the exact factorization used in LM, the conditional probability of the masked tokens is approximated based on a conditional independence assumption. Moreover, in contrast

to forward LM, left-to-right text generation is more complex in the case of MLM.[5]

3.1.3 Contrastive Learning

CTL, formalized by Arora et al. [3], consists in training a model by differentiating observed data points from artificially generated samples. This approach is motivated by the idea that observed data points are semantically more similar than artificial samples, which should enable the model to learn semantically meaningful representations. Specifically, we consider a variant of CTL called *noise-contractive estimation* (NCE) [36], which has proven useful in language representation learning. In NCE, a binary classifier discriminates between the observed data and artificially generated noise. Given an input sequence $\boldsymbol{x} = [x_1, \ldots, x_n]$, the first step of NCE consists in generating a randomly corrupted sample $\boldsymbol{x}^{\text{corrupt}} = [x_1^{\text{corrupt}}, \ldots, x_n^{\text{corrupt}}]$, where for a fixed proportion of the tokens it holds $x_i^{\text{corrupt}} = x_i$. In contrast to MLM, the corrupted examples, which are sampled from a noise distribution, are similar to the real input examples instead of being masked out. Using $\boldsymbol{x}^{\text{corrupt}}$ as input, the discriminator then predicts for each position whether the token is corrupted or not. Specifically, the probability that a token has not been corrupted can be modeled as

$$p(x_i^{\text{corrupt}} = x_i | \boldsymbol{x}^{\text{corrupt}}; \boldsymbol{\theta}) = \frac{\exp\Big(\boldsymbol{z}_{\text{CTL}}(\boldsymbol{x}^{\text{corrupt}})_i^\top \boldsymbol{w}\Big)}{1 + \exp\Big(\boldsymbol{z}_{\text{CTL}}(\boldsymbol{x}^{\text{corrupt}})_i^\top \boldsymbol{w}\Big)}, \tag{3.1.7}$$

where $\boldsymbol{z}_{\text{CTL}}(\boldsymbol{x}) = [\boldsymbol{z}_{\text{CTL}}(\boldsymbol{x})_1, \ldots, \boldsymbol{z}_{\text{CTL}}(\boldsymbol{x})_n]$ is again a sequence of contextualized input representations of a sequence $\boldsymbol{x}$ and $\boldsymbol{w} \in \mathbb{R}^H$ are the weights of a sigmoid output layer. Using the notation $\hat{p}_i := p(x_i^{\text{corrupt}} = x_i | \boldsymbol{x}^{\text{corrupt}}; \boldsymbol{\theta})$, the loss for each input sequence $\boldsymbol{x}$ can be written as

$$\mathcal{L}_{\text{CTL}}(\boldsymbol{x}, \boldsymbol{\theta}) = \sum_{i=1}^{n} -\mathbb{1}(x_i^{\text{corrupt}} = x_i) \log \hat{p}_i - \mathbb{1}(x_i^{\text{corrupt}} \neq x_i) \log(1 - \hat{p}_i),$$

which is an approximation of the expected cross-entropy over the corrupted inputs with a sample $\boldsymbol{x}^{\text{corrupt}}$ from the noise distribution. As before, parameters are obtained by minimizing the combined loss $\sum_{\boldsymbol{x} \in \mathcal{X}} \mathcal{L}_{\text{CTL}}(\boldsymbol{x}, \boldsymbol{\theta})$. The recently introduced model ELECTRA [18] leverages an extension of CTL called *replaced token detection* (RTD) [18]. This approach involves a generator network, in the case of ELECTRA a small

[5] Representing BERT as a Markov Random Field, Wang and Cho [119] show that text generation can be accomplished employing Gibbs sampling.

masked language model, that produces the corrupted inputs and is trained jointly with the discriminator network. ELECTRA employs a training objective resembling the objective of a *generative adversarial network* (GAN) [33], with the main difference that the generator is trained with maximum likelihood rather than an adversarial objective. The discriminator is used on downstream tasks, whereas the generator is solely active during pretraining.

Systems In a seminal paper, Collobert and Weston [19] presented a semi-supervised system, which learns word vector representations by discriminating generated fake text from positive examples. Mnih and Kavukcuoglu [74] learned word vector representations with NCE, where in addition to the generated samples the binary discriminator takes probabilities of the noise distribution as input. To obtain Word2Vec embeddings using the *continuous bag-of-words* (CBOW) architecture, Mikolov et al. [71] introduced a simpler version of NCE called *negative sampling* (NEG). In contrast to the original NCE, in NEG the binary discriminator does not take probabilities of the noise distribution as input, but instead only uses samples from this distribution. ELECTRA extends this approach by replacing the noise distribution with a generator network, as described above. Furthermore, ELECTRA uses a Transformer network, while Mnih and Kavukcuoglu [74] employed a modified *log-bilinear language model* (LBL) [73] and Mikolov et al. [71] a feedforward neural network to differentiate original data points from artificial samples, inputting a window of surrounding context. Finally, the NSP objective from BERT and the SOP objective from ALBERT, which were mentioned in the previous section, are also examples of NCE.

Pros and Cons While CTL is conceptually similar to MLM, the pre-training objective of CTL involves *all* data points. Thus, each input sequence has a much larger effect on the training progress. Furthermore, especially in the case of ELECTRA, which replaces original input tokens with reasonable alternatives, the mismatch between pre-training data and fine-tuning data is alleviated. As Clark et al. [18] demonstrate, compared to MLM, these improvements lead to significantly reduced runtime and improved performance on downstream tasks.

3.2 Transfer to Downstream Tasks

Many different variants have been proposed to transfer pre-trained representations to downstream tasks. Existing approaches can be classified into two broad categories: *fea-*

ture extraction and *fine-tuning* approaches [25, 84].[6] Most state-of-the-art NLP systems employ a fine-tuning approach, which has largely replaced feature extraction approaches.

3.2.1 Feature Extraction

In feature extraction, specific parts of the pre-trained representations are extracted and used as inputs of a task-specific supervised model. Transfer to downstream tasks thus involves additional task-specific architectures. The supervised stage consists to a large degree of training newly initialized parameters of these task-specific architectures from scratch. Apart from the extracted features, other pre-trained parameters are not used during the supervised training.

Systems NLP feature extraction approaches with neural networks have their origins in word vector representations described in Chapter 2, which were introduced to replace methods requiring hand-crafted features. Bengio et al. [7] first trained a neural language model to obtain word vector representations, while other approaches took into account the bidirectional context of a word [19, 20]. While Collobert et al. [20] used a multilayer neural network with convolutions, smaller architectures, such as used in Word2Vec [71] or GloVe [80], were shown to be sufficient for capturing useful semantic and syntactic information with word vectors. To incorporate higher-level semantics, this approach was extended to learning contextualized word vectors [67, 68] and representations of sentences and paragraphs [21, 41, 57, 63, 108]. ELMo [81] obtained contextualized representations of word sequences by extracting layers of pre-trained LSTMs, which were then used as inputs of task-specific RNNs. Notably, feature-based versions of BERT, likewise producing contextualized word representations, achieved similar performance in the *named entity recognition* (NER) task as fine-tuned versions [25].

How to extract If trained with a neural network, word vector representations are simply obtained as the first layer of a network. In contrast, to obtain contextualized representations it is necessary to extract higher layers. More recent systems often extract multiple layers. For instance, ELMo extracts a weighted average of all pre-trained layers. The best performing feature-based version of BERT uses the top four layers of the Transformer encoder. Feature extraction approaches have particularly been applied to token-level tasks, such as NER [20, 25, 80, 81, 115], *part-of-speech tagging* (POS) [20] or QA [81, 112], for which representations of multiple or all tokens of an input sequence are extracted.

[6] Feature extraction was in the past sometimes referred to as fine-tuning [20], because performing additional gradient updates to pre-trained weights in an end-to-end fashion was not common.

Pros and Cons Feature extraction can be useful if many similar tasks have to be performed, because supervised training of a relatively small task-specific architecture without updating the pre-trained weights is usually not very expensive. Furthermore, using a task-specific architecture that takes specific pre-trained features as input can be the most accurate method for certain tasks. On the other hand, compared to fine-tuning approaches, feature extraction contradicts the goal of building a unified architecture to improve generalization of many shared tasks.

3.2.2 Fine-Tuning

In contrast to feature-based approaches, fine-tuning consists of performing additional gradient updates to the pre-trained parameters in an end-to-end fashion. That is, the pre-trained parameters together with the architecture are used as an initialization point for the supervised training. Only a minimal set of additional parameters is introduced during fine-tuning, leveraging the existing pre-trained structure instead of introducing new task-specific architectures.

Systems Fine-tuning approaches in NLP were first applied in *sequence to sequence* (Seq2Seq) learning, especially NMT. Dai and Le [23] presented the first fine-tuning approach in NLP, proposing to pre-train a Seq2Seq system with an autoencoding objective and initialize the network with the pre-trained weights for subsequent supervised training. Ramachandran et al. [91] pre-trained the encoder and the decoder of a Seq2Seq system separately with an LM objective, initializing multiple layers of both parts of the network with pre-trained weights. Howard and Ruder [44] trained a regular LSTM-RNN with an LM objective and introduced several fine-tuning tricks. Another line of work used fine-tuning to transfer between similar supervised tasks [72, 101]. Since the introduction of the Transformer, most state-of-the-art systems have leveraged a fine-tuning transfer of pre-trained contextualized representations. The first fine-tuned Transformer-based systems were GPT and BERT, followed by many others, such as XLNet, RoBERTa, ALBERT and ELECTRA.

How to fine-tune Usually the pre-trained representation of the top layer is simply fed through an added dense layer to obtain the predicted labels. Subsequently the full network is updated based on the labeled data and a supervised objective [25, 56, 62, 85, 129]. There also exist fine-tuning techniques that update only a subset of the pre-trained weights, such as *adapter layers* [6, 43] or *gradual unfreezing* [44], which were studied in T5 [88]. These techniques allow for more efficient fine-tuning, since

significantly less parameters are updated for a new fine-tuning task, which can be useful if a large number of fine-tuning tasks are performed. In some cases supervised objectives are trained jointly with unsupervised objectives during fine-tuning to avoid overfitting [85, 91].

Pros and Cons Fine-tuning is an efficient approach if the goal is to perform tasks from a large and diverse range, because it allows for using a unified end-to-end system. Furthermore, updating a large set of parameters starting from a good pre-trained initialization can improve performance significantly. In some cases, however, if the goal is to build an expert system geared towards a specific task, approaches that involve a task-specific architecture can be more accurate.

3.3 Multi-Task Learning

In MTL, a single system learns to perform several tasks at once. This is usually achieved by optimizing more than one loss function [93], but there exist also instances which simply mix different task-specific datasets together during training [88]. As Ruder [93] argues, MTL can be regarded as a form of inductive transfer: auxiliary tasks introduce an *inductive bias*, which causes the system to choose certain hypotheses over others, ideally allowing for better generalization. In neural networks, MTL can be implemented either via *soft parameter sharing* or via *hard parameter sharing* [93]. In hard parameter sharing, lower layers of a network are shared across tasks, whereas top layers are task-specific. In contrast, soft parameter sharing implies that a different network is used for each task, but parameters between the different networks are regularized, e.g. via the L_1 norm.

MTL has also been combined with pre-train/fine-tune approaches in NLP. Liu et al. [61] pre-trained contextualized representations using a Transformer encoder as in BERT, followed by a second stage in which the parameters were updated by training on multiple supervised tasks at once with task-specific output layers. This approach, which is an instance of hard parameter sharing, led to an absolute improvement of 2.2% on the GLUE benchmark compared to the original version of BERT. In T5, Raffel et al. [88] implemented MTL by mixing different datasets together while using the same loss function for all tasks. Consistent inputs and outputs, specified in natural language across all tasks, allow for using a homogeneous maximum likelihood objective. Adding multiple supervised tasks to the unsupervised objective during pre-training results in comparable performance to unsupervised pre-training. However, removing the unsupervised

objective and solely pre-training on multiple supervised tasks prior to fine-tuning on a single task results in significantly worse performance, which underlines the importance of unsupervised pre-training in NLP.

3.4 Meta-Learning

While avoiding task-specific architectures, fine-tuning approaches still require large task-specific labeled data sets. This requirement limits the practicability of fine-tuning approaches, since high-quality labeled data does not always exist and constructing such data is often infeasible. Furthermore, while pre-training reduces the risk of overfitting there remains a strong dependence of the final model on the fine-tuning data, which is much smaller than the pre-training data [11, 44].

In NLP, *meta-learning* or *in-context learning* [11] describes a set of methods enabling a pre-trained system to adapt to different tasks without performing additional gradient updates. Furthermore, no or only a minimal amount of task-specific data is provided. *Meta-learning* can be implemented by specifying the task to be performed as a natural language instruction added to the task-specific input [11, 86]. As stated above, this approach was also used in T5 to enable MTL. The idea is that, by having processed naturally occurring instances of such instructions together with a surrounding context in the training data, the system automatically infers the demanded task from the instruction. In addition to the natural language instruction, the system can be conditioned on one or more examples, known as *one-shot* and *few-shot* learning, respectively. The *zero-shot* setting, in which no examples are provided, has the advantage that there is no risk of overfitting to the provided examples. At the same time, however, the zero-shot setting is more challenging and instructions can potentially be ambiguous without demonstrations. In general, the suitability of the different settings is task-dependent, particularly regarding the goal of mimicking the way instructions are communicated to humans [11].

GPT-2 [86] was the first prominent system using Meta-Learning in NLP. While architecture and pre-training procedure of GPT-2 follow the original GPT, fine-tuning is replaced by giving the system a natural language instruction together with zero or more examples. The same approach was used in GPT-3 [11], however, with an increased model size of 175B parameters compared to 1.5B parameters. GPT-3 achieved an impressive performance across a large range of tasks. As Radford et al. [86] state, in effect such a system performs unsupervised MTL.

While Meta-Learning, as outlined above, has several advantages, the performance

is worse compared to the performance of a fine-tuned system with similar capacity. A notable weakness of GPT-3 was especially observed on NLI tasks [11]. This raises the question whether such systems simply learn templates and syntactic patterns that are unrelated to logic and inference.

Chapter 4

Architectures

This chapter gives an overview of the most important architectures that have been used in NLP in recent years. Specifically, we discuss different neural networks. The goal of this section is to provide a concise high-level overview, presented in a chronological order correspondent to the evolution of neural networks in NLP. We attempt to trace this evolution by discussing the most important properties that result from the different architectures, mainly from a computational perspective. As most state-of-the-art NLP systems use the now ubiquitous Transformer, we conduct a detailed description of this architecture in section 4.3.

4.1 Recurrent Neural Network

A *recurrent neural network* (RNN) [96, 122] is a class of neural networks developed to model variable-length sequences. An RNN processes elements sequentially by using an internal memory, consisting of so-called *hidden states* $\boldsymbol{h}_i$, where *recurrent connections* between the hidden states allow for retaining information about past elements. Each hidden state is a function f of the previous hidden state $\boldsymbol{h}_{i-1}$ and the current input embedding $\boldsymbol{e}(x_i)$:

$$\boldsymbol{h}_i = f(\boldsymbol{e}(x_i), \boldsymbol{h}_{i-1}). \tag{4.1.1}$$

The so-called *state transition function* f is usually of the form

$$f(\boldsymbol{e}(x_i), \boldsymbol{h}_{i-1}) = \phi(\boldsymbol{W}_1^T \boldsymbol{e}(x_i) + \boldsymbol{W}_2^T \boldsymbol{h}_{i-1}), \tag{4.1.2}$$

where ϕ is an activation function, such as the sigmoid function. In order to enable the network to process variable-length sequences, at each step, the same sets of input connections $\boldsymbol{W}_1$ and hidden-to-hidden connections $\boldsymbol{W}_2$ are used, which is commonly referred to as *parameter sharing*. An RNN can also have multiple layers. In this case, the input embedding $\boldsymbol{e}(x_i)$ is replaced with the output embedding from the previous layer. The hidden states of the final layer are a contextualized representation of the input sequence.

4.1.1 Language Modeling

As described in Chapter 3, LM is motivated by the autoregressive factorization

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i|\boldsymbol{x}_{<i}), \tag{4.1.3}$$

where $\boldsymbol{x}_{<i}$ is defined as in Chapter 3. In a *simple recurrent neural network*, also known as *Elman network* [28], the hidden states are directly used to predict the probabilities

$$p(x_i|\boldsymbol{x}_{<i}; \boldsymbol{\theta}) = g(\boldsymbol{h}_{i-1}), \tag{4.1.4}$$

where g is a nonlinear function that outputs the probability of x_i, which is usually implemented with a softmax normalization. Such a network has for instance been used in speech recognition [70].

4.1.2 Sequence to Sequence Learning

Another application of RNNs in NLP is Seq2Seq learning, where an input sequence $\boldsymbol{x} = [x_1, \ldots, x_n]$ is used to predict an output sequence $\boldsymbol{y} = [y_1, \ldots, y_m]$ of different length. Similar to LM, this supervised task can be reduced to predicting the individual factors in the autoregressive factorization

$$p(\boldsymbol{y}|\boldsymbol{x}) = \prod_{i=1}^{m} p(y_i|\boldsymbol{y}_{<i}, \boldsymbol{x}), \tag{4.1.5}$$

which is accomplished with an encoder-decoder structure [109]. An encoder-RNN first compresses the input sequence $\boldsymbol{x} = [x_1, \ldots, x_n]$ into a so-called *context-vector*

$$\boldsymbol{c} = q(\{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n\}), \tag{4.1.6}$$

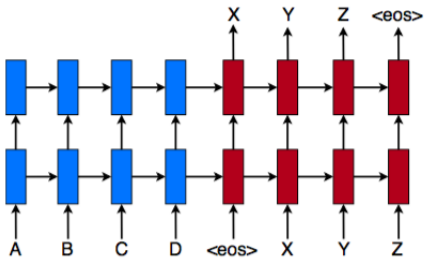

Figure 4.1: An encoder-RNN (blue) maps the input sequence to a context vector, in this case the last hidden state. A decoder-RNN (red) then predicts output elements by accessing context vector and previous outputs.

where $\boldsymbol{h}_i$ are the hidden states of the encoder-RNN.[1] The context vector is subsequently accessed by a decoder-RNN, which models the individual factors as

$$p(y_i|\boldsymbol{y}_{<i}, \boldsymbol{x}; \boldsymbol{\theta}) = g(y_{i-1}, \boldsymbol{s}_i, \boldsymbol{c}), \tag{4.1.7}$$

where $\boldsymbol{s}_i$ are the hidden states of the decoder-RNN and g is again a non-linear function that outputs probabilities. Encoder-decoder RNNs have been used especially in machine translation [109]. As proposed by Dai and Le [23], the encoder-decoder structure can also be leveraged to pre-train contextual representations in a semi-supervised setting. This can be achieved, for example, by replacing the output sequence with the input sequence so that the decoder reconstructs the input sequence from the context vector.

Attention in Encoder-Decoder RNNs

The structure of encoder-decoder RNNs involves a memorization step, in which the encoded sequence is compressed into a single, low resolution context vector, before decoding. Within this process, much information about the input sequence is lost, which prevents the network from learning long-range dependencies. To remedy this effect, Bahdanau et al. [5] proposed an attention mechanism to dynamically access the hidden states of the encoder.

To enable the decoder to dynamically focus on different hidden states of the encoder at each step, Bahdanau et al. [5] replaced the fixed context vector $\boldsymbol{c}$ with an individual context vector $\boldsymbol{c}_i$, accessed by the decoder when predicting the i-th output element. A context vector $\boldsymbol{c}_i$ is obtained as the weighted sum of the encoder's hidden states $\boldsymbol{h}_j$,

[1] A common choice is $q(\{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n\}) = \boldsymbol{h}_n$.

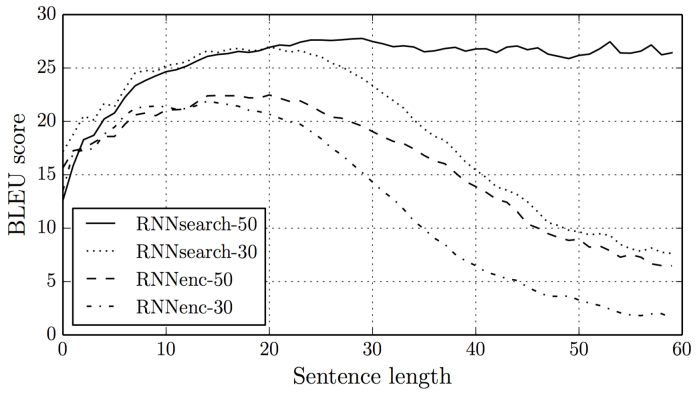

Figure 4.2: The RNN with attention (RNNsearch-50) performs much better on long sentences than a comparable RNN without attention (RNNenc-50) [5].

$j = 1, \ldots, n$:

$$\boldsymbol{c}_i = \sum_{j=1}^{n} \beta_{ij} \boldsymbol{h}_j. \tag{4.1.8}$$

The weights of the i-th context vector indicate which parts of the input sequence are most relevant for the i-th output element. The calculation of these weights is based on evaluating how well the hidden states of the encoder match the hidden states of the decoder associated with the ith output element. Bahdanau et al. [5] first calculate so-called *alignment scores*

$$a_{ij} = f(\boldsymbol{s}_{i-1}, \boldsymbol{h}_j), \tag{4.1.9}$$

were f is a *compatibility function* given by a feed-forward neural network with a single hidden layer.[2] Weights are computed by normalizing these scores with the softmax function:

$$\beta_{ij} = \frac{\exp\big(a_{ij}\big)}{\sum_{k=1}^{n} \exp\big(a_{ij}\big)}. \tag{4.1.10}$$

This class of attention mechanisms is known as *additive attention*.

Luong et al. [64] proposed several alternative functions for generating alignment scores, one of which is based on the dot-products between hidden states of the encoder and decoder. In this case, the weights β_{ij}, $j = 1, \ldots, n$, of the i-th context vector are

[2]Specifically, Bahdanau et al. [5] use $f(\boldsymbol{s}_{i-1}, \boldsymbol{h}_j) = \boldsymbol{W}_3^T(\tanh(\boldsymbol{W}_1^T s_{i-1} + \boldsymbol{W}_2^T h_j))$ for suitable weight matrices.

computed as

$$\beta_{ij} = \frac{\exp\left(\boldsymbol{s}_i^T \boldsymbol{h}_j\right)}{\sum_{k=1}^{n} \exp\left(\boldsymbol{s}_i^T \boldsymbol{h}_k\right)}, \tag{4.1.11}$$

where $\boldsymbol{s}_i$ denotes the i-th hidden state of the decoder.[3] As will be shown in section 4.3, this mechanism, known as *dot-product attention*, is closely related to attention as applied in the Transformer.

4.1.3 Computational Perspective

RNNs are powerful sequence models applicable to a wide range of tasks. For instance, when encoder-decoder RNNs were introduced to the field of machine translation, they quickly achieved state-of-the-art performance, marking the beginning of *neural machine translation* (NMT). However, the inherent serial structure of RNNs has severe limitations.

Parallel computation First, the sequential order prohibits parallelization. The greater the number of units that must be computed sequentially across all layers, the lower the degree of parallelization. As shown in Figure 4.3, since each hidden state $\boldsymbol{h}_i$ depends on the previous hidden state $\boldsymbol{h}_{i-1}$, this number is proportional to the length of the input sequence in RNNs. Thus, the number of operations that need to be computed sequentially is $\mathcal{O}(n)$. This sequentiality results in increased runtimes, exacerbated by memory constraints that limit the batch size, as large batch sizes mitigate the computational costs of parameter updates [102].

Path length Second, during training, relating two elements of an input sequence involves backpropagation along the full network path spanned between the two elements. Due to the serial structure of RNNs, this path length is linear in the distance between the input elements. This feature prevents the network from learning long-range dependencies, as long paths typically result in vanishing and sometimes in exploding gradients [42]. Most encoder-decoder RNNs employ *long short-term memory* (LSTM) [42] architectures [109] or *gated recurrent units* (GRUs) [17] to mitigate this effect. Furthermore, attention does also reduce the path length, since the decoder can directly access individual states of the encoder. However, none of these variants resolves the inherent

[3] Note that, in contrast to Bahdanau et al. [5], Luong et al. [64] hence use the i-th decoder state $\boldsymbol{s}_i$ (instead of $\boldsymbol{s}_{i-1}$) to compute the i-th contect vector $\boldsymbol{c}_i$. For details, see Luong et al. [64], section 3.1, and Bahdanau et al. [5], section A.2.2.

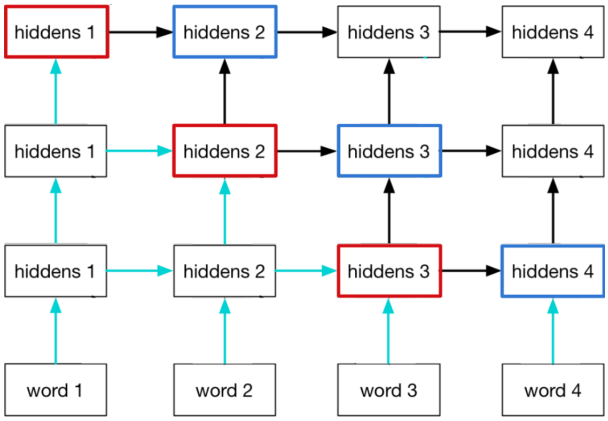

Figure 4.3: The blue units can be computed in parallel, but only given the red units. In RNNs, the number of sequential operations is thus proportional to the input length [35].

sequentiality of RNNs. Thus, difficulties with relating signals in long sequences persist. In NLP, however, long-range dependency is predominant, not least because of the division into subwords [24].

Layer complexity Each matrix multiplication in Eq. (4.1.2) requires $\mathcal{O}(H^2)$ operations. Since there are n hidden states per layer, the number of operations which have to be performed per layer hence is $\mathcal{O}(n \cdot H^2)$. Note that the attention mechanism comes at the expense of increased complexity. Since the context vector is calculated for each position, complexity increases to $\mathcal{O}(n^2 \cdot H^2)$ [46].

4.2 Convolutional Neural Network

Aiming at resolving the problems associated with RNNs, another class of models that has been used in sequence modeling is the *convolutional neural network* (CNN) [30, 46, 77]. When applied to sequence modeling, these models simply consist of a stack of convolutional layers. The final hidden layer again yields a series of contextualized representations of the input. As recurrent layers, a convolutional layer shares the same set of parameters across input positions, allowing for processing variable-length sequences.

Let the outputs of the l-th convolutional layer be denoted as $\boldsymbol{h}^l = (\boldsymbol{h}_1^l, \ldots, \boldsymbol{h}_n^l)$, where $\boldsymbol{h}_i^l \in \mathbb{R}^H$. The embedding dimension H is referred to as the number of channels in a CNN. A convolutional layer obtains each output by applying a filter, which is shared across all positions, to the outputs of the preceding layer. A filter with kernel size $K = 2R+1$ connects K elements to compute each output. For a suitable activation

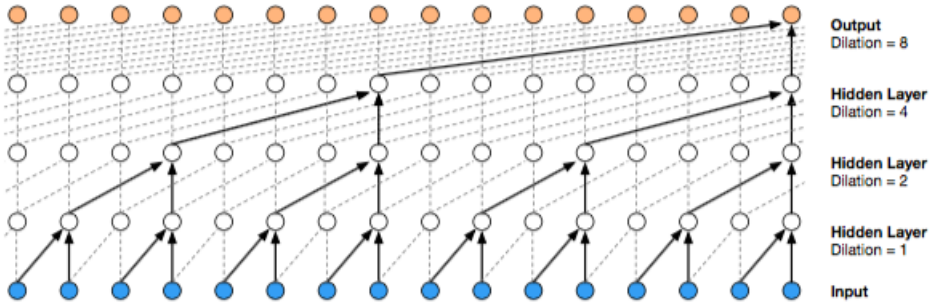

Figure 4.4: WaveNet uses causal dilated convolutions [77].

function ϕ, a one-dimensional convolutional layer computes the m-th channel of the i-th output as

$$h_{i,m}^{l+1} = \phi\Big(\sum_{j=1}^{H}\sum_{\tau=-R}^{R} w_{m,j,\tau}^{l+1} h_{i+\tau,j}^{l}\Big), \tag{4.2.1}$$

where on the RHS we assume that $h_{i+\tau,j}^{l} = 0$, if $i + \tau \notin \{1, \ldots, n\}$.[4] Note that the weights are not indexed by the position of the output element and thus are shared across all positions, as mentioned.

4.2.1 Language Modeling

Like recurrent networks, one-dimensional CNNs have been applied to language modeling. A prominent example is *WaveNet* [77], which is trained on raw audio waveforms and can be used in text-to-speech applications. While in RNNs the serial structure naturally prevents that the prediction $p(x_i|\boldsymbol{x}_{<i};\boldsymbol{\theta})$ depends on future elements, in ordinary CNNs outputs can attend to future elements. To make sure that future elements cannot affect the predicted output, Oord et al. [77] proposed so-called *causal convolutions*, masking all weights that connect elements from right to left (see Figure 4.4). Furthermore, Oord et al. [77] used *dilated convolutions*, which is a technique that allows for modeling long-range dependencies. Formally, a convolution with dilation factor s can be written

[4] This approach is known as *same padding*, which is used in sequence modeling. Same padding allows for a constant number of elements per layer, as otherwise the filter could not be applied to all elements, which would reduce the number of elements after each layer.

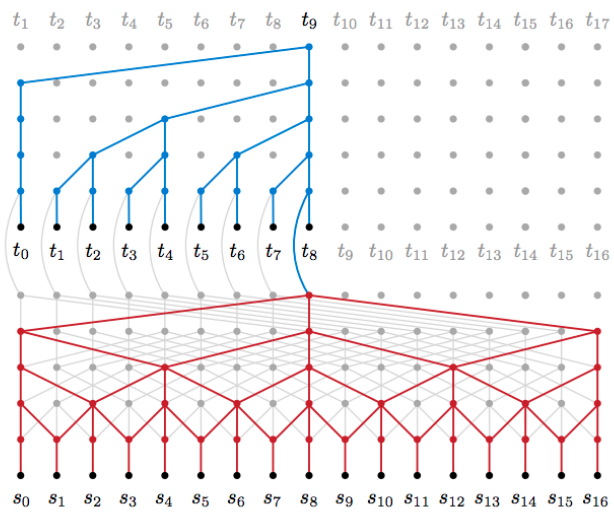

Figure 4.5: The encoder-decoder architecture of ByteNet [46].

as

$$h_{i,m}^{l+1} = \phi\Big(\sum_{j=1}^{H}\sum_{\tau=-R}^{R} w_{m,j,\tau}^{l+1} h_{i+s\tau,j}^{l}\Big). \tag{4.2.2}$$

That is, only every s-th element is accessed. By inserting zeros between filter elements, dilated convolutions increase the area covered by each output neuron (as shown in Figure 4.4), known also as the *receptive field* of a network. Note that Oord et al. [77] use increasing factors of dilation.

4.2.2 Sequence to Sequence Learning

To apply CNNs to input and output sequences of different lengths, as in RNNs, an encoder-decoder architecture can be used [30, 46]. *ByteNet* [46], developed for machine translation, is an instance of such a system and a generalization of WaveNet. Similar to WaveNet, dilated convolutions in ByteNet allow for learning long-range dependencies. Furthermore, as dictated by Eq. (4.1.5), the decoder of ByteNet adopts the autoregressive masking pattern used in WaveNet, which prevents future elements from predicting the current output. In contrast, the encoder applies no mask. As will be shown in section 4.3, this approach is also used in the Transformer. In contrast to standard RNNs without attention, the encoder of ByteNet computes n contextualized representations,

which also referred to as *resolution preserving*.

The attention mechanism, initially developed for RNNs, has also been applied to encoder-decoder CNNs. Using a CNN with attention and GRUs, *ConvS2S* [30] achieved state-of-the-art performance in machine translation, surpassing Google's machine translation system at the time, an encoder-decoder RNN with attention [127].

4.2.3 Computational Perspective

Parallel computation Since CNNs, unlike RNNs, do not have recurrent connections, they allow a high degree of parallelization. This is reflected in a constant number of sequential operations in the length of the input sequence, since the number of layers is constant and parallelization is performed layer-wise. Thus, when applied to long sequences, CNNs usually have shorter runtimes than RNNs [77].

Path length By using dilation, compared to RNNs, WaveNet and ByteNet reduce the path traversed to relate two elements of an input sequence to $\mathcal{O}(\log_K(n))$. This facilitates modeling long-range dependencies with CNNs.

Layer complexity Eq. (4.2.1) shows that computation of each channel of an element requires $\mathcal{O}(H \cdot K)$ operations (assuming that successive layers have the same number of channels). Since there are H channels per element and n elements per layer, the number of operations which have to be performed per layer thus is $\mathcal{O}(H^2 \cdot K \cdot n)$. While CNNs thus have a higher complexity than standard encoder-decoder RNNs, the complexity is lower compared to encoder-decoder RNNs with added attention. Thus, CNNs provide the resolution-preserving property at a lower computational cost than RNNs with attention.

4.3 Transformer

The Transformer [117] is a sequence modeling architecture originally proposed as an end-to-end system for MT. The application of this novel architecture in modern NLP systems is a key driver of the success of these systems. The central component of the Transformer is the so-called *self-attention mechanism* [14, 79], which is inspired by the attention mechanism originally proposed for recurrent networks. Compared to previous sequence modeling approaches, this mechanism allows the Transformer to relate signals within an input sequence in a more direct and efficient manner.

4.3.1 More Attention

Prior to the introduction of the Transformer, attention mechanisms had already become an integral part of many models. As outlined in the previous chapter, in encoder-decoder RNNs the attention mechanism was introduced to automatically search for sections of an input sequence that are relevant to predicting an output element [5, 64]. Aside from its application in MT, attention had for instance also been used in CV [34, 128, 130]. In particular, self-attention [14, 79], originally called *intra-attention*, had been introduced.

While attention had become a widely adopted mechanism, it had in all cases been applied in combination with other mechanisms, such as recurrence. In contrast, Vaswani et al. [117] demonstrated that "Attention Is All You Need". Relying entirely on attention to relate different elements of a sequence, Vaswani et al. [117] achieved state-of-the-art performance in MT.

Universal definition The Transformer applies different variants of attention. In order to describe these variants and to formalize attention within a broader framework, we adopt the general definition of attention used by Vaswani et al. [117]. This definition is closely related to attention as introduced in section 4.1.2. Following Vaswani et al. [117], attention can be defined as a mechanism that takes as input a sequence of *queries* $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n \in \mathbb{R}^{H_k}$, *keys* $\boldsymbol{k}_1, \ldots, \boldsymbol{k}_n \in \mathbb{R}^{H_k}$ and *values* $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n \in \mathbb{R}^{H_v}$. As in RNNs, the similarity a_{ij} between query $\boldsymbol{q}_i$ and key $\boldsymbol{k}_j$ is measured by a compatibility function:

$$a_{ij} = f(\boldsymbol{q}_i, \boldsymbol{k}_j). \tag{4.3.1}$$

In Table 4.1 we provide a list of compatibility functions found in the literature. The output consists of a sequence of vectors $\boldsymbol{o}_1, \ldots, \boldsymbol{o}_n \in \mathbb{R}^{H_v}$, where each vector is computed as the sum of the values, weighted with the similarity between queries and keys:

$$\boldsymbol{o}_i = \sum_{j=1}^{n} \beta_{ij} \boldsymbol{v}_j, \qquad \text{where } \beta_{ij} = \frac{\exp(a_{ij})}{\sum_{k=1}^{n} \exp(a_{ik})}. \tag{4.3.2}$$

As can be observed, in the case of RNNs queries correspond to the hidden states of the decoder, while the hidden states of the encoder play the role of both keys and values. Note that the attention mechanism addresses elements based on content. That is, the attention weights are computed dynamically depending on the content of each input sequence. This is in contrast to other sequence architectures, such as dilated CNNs, where input elements are referenced statically by position.

Compatibility Functions		
Name	Function	Reference
additive	$f(\boldsymbol{q}_i, \boldsymbol{k}_j) = \boldsymbol{w}_{\text{out}}^T \phi(\boldsymbol{W}_1^T \boldsymbol{k}_j + \boldsymbol{W}_2^T \boldsymbol{q}_i)$	Bahdanau et al. [5]
general bilinear	$f(\boldsymbol{q}_i, \boldsymbol{k}_j) = \boldsymbol{q}_i^T \boldsymbol{W}^T \boldsymbol{k}_j$	Luong et al. [64]
dot-product/multiplicative	$f(\boldsymbol{q}_i, \boldsymbol{k}_j) = \boldsymbol{q}_i^T \boldsymbol{k}_j$	Luong et al. [64]
concat	$f(\boldsymbol{q}_i, \boldsymbol{k}_j) = \boldsymbol{w}_{\text{out}}^T \phi(\boldsymbol{W}^T [\boldsymbol{k}_j \parallel \boldsymbol{q}_i])$	Luong et al. [64]
location-based	$f(\boldsymbol{q}_i, \boldsymbol{k}_j) = f(\boldsymbol{W}^T \boldsymbol{q}_i)$	Luong et al. [64]
biased general	$f(\boldsymbol{q}_i, \boldsymbol{k}_j) = \boldsymbol{k}_j^T (\boldsymbol{W}^T \boldsymbol{q}_i + \boldsymbol{b})$	Sordoni et al. [107]
activated general	$f(\boldsymbol{q}_i, \boldsymbol{k}_j) = \phi(\boldsymbol{q}_i^T \boldsymbol{W}^T \boldsymbol{k}_j)$	Ma et al. [65]
scaled dot-product	$f(\boldsymbol{q}_i, \boldsymbol{k}_j) = \frac{\boldsymbol{q}_i^T \boldsymbol{k}_j}{\sqrt{H_k}}$	Vaswani et al. [117]

Table 4.1: Overview of compatibility functions found in the literature.

Scaled dot-product attention The Transformer uses a variant of attention known as *scaled dot-product attention* [117]. In this case, the compatibility function is given by the scaled dot-product between keys and queries:

$$f(\boldsymbol{q}_i, \boldsymbol{k}_j) = \frac{1}{\sqrt{H_k}} \boldsymbol{q}_i^T \boldsymbol{k}_j. \tag{4.3.3}$$

Dot-products are scaled by the factor $\frac{1}{\sqrt{H_k}}$ to avoid large inputs of the softmax function. For large inputs the gradient of the softmax becomes very small, which slows down learning [117]. In gradient-based learning methods, such as neural networks, this problem is generally known as the *vanishing gradient problem* [42]. By stacking queries, keys and values row-wise into $\boldsymbol{Q} \in \mathbb{R}^{n \times H_k}$, $\boldsymbol{K} \in \mathbb{R}^{n \times H_k}$ and $\boldsymbol{V} \in \mathbb{R}^{n \times H_v}$, scaled dot-product attention can be compactly written as

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\Big(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{H_k}}\Big)\boldsymbol{V}, \tag{4.3.4}$$

where the softmax function is applied row-wise.

Self attention In *self-attention* [14, 79], the attention mechanism is applied to a single

input sequence. Queries, keys and values are obtained from a sequence of H-dimensional embeddings with three learnable linear transformations $\boldsymbol{W}^Q \in \mathbb{R}^{H \times H_k}$, $\boldsymbol{W}^K \in \mathbb{R}^{H \times H_k}$ and $\boldsymbol{W}^V \in \mathbb{R}^{H \times H_v}$. The linear transformations give self-attention a set of controllable parameters to differentiate between the three roles of queries, keys and values. At a high-level, a self-attention layer takes a sequence as input and outputs a sequence of equal length, representing the context in which each input element appears.

Multi-head attention In order to give attention more power to discriminate between different regions of the input vectors, several attention functions can be applied in parallel. This is known as *multi-head attention* [117]. For multi-head attention, Vaswani et al. [117] consider queries, keys and values of equal dimension H. In a first step, queries, keys and values are linearly transformed to dimensions H_k, H_k and H_v, respectively. Each input is transformed A times, which is achieved with A linear transformations $\boldsymbol{W}_j^Q \in \mathbb{R}^{H \times H_k}$, $\boldsymbol{W}_j^K \in \mathbb{R}^{H \times H_k}$ and $\boldsymbol{W}_j^V \in \mathbb{R}^{H \times H_v}$, $j = 1, \ldots, A$. In most versions of the Transformer the dimensions of the attention components are chosen such that $H_k = H_v = H/A$. Each of the A sets of transformed queries, keys and values is then separately used as input of the attention function defined in Eq. (4.3.4). Finally, in order to produce a single output sequence of the same embedding dimension as the input sequence, the resulting A outputs of the attention function, known as *attention heads* [117], are concatenated and transformed back to dimension H with a linear mapping $\boldsymbol{W}^O \in \mathbb{R}^{H \times AH_v}$, which is also learned during training. Following Vaswani et al. [117], multi-head attention can thus be written as

$$\begin{aligned} \text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_A)\boldsymbol{W}^O, \\ \text{where head}_j &= \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_j^Q, \boldsymbol{K}\boldsymbol{W}_j^K, \boldsymbol{V}\boldsymbol{W}_j^V). \end{aligned} \tag{4.3.5}$$

Multi-head self attention Furthermore, the Transformer uses a combination of self-attention and multi-head attention, called *multi-head self-attention*. In this case, $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ in Eq. (4.3.5) are replaced with a single matrix, containing the embeddings of a single input sequence.

Masked self-attention As a machine translation model, the original Transformer builds on the autoregressive factorization given by Eq. (4.1.5). Thus, during training elements of the output sequence must not depend on future output elements. However, in the above version of self-attention each output element can attend to all the elements of the input sequence. To prevent output elements from being generated by future elements a so-called *attention mask* can be applied to the input of the softmax. Specifically,

the upper triangular part of $\boldsymbol{Q}\boldsymbol{K}^T$ is set to $-\infty$, resulting in weights $\beta_{ij} = 0$, if $i < j$. Apart from preserving the autoregressive structure, the attention mask has another purpose. Note that the Transformer is a fixed-length model. That is, in contrast to RNNs, the Transformer takes as input sequences of fixed length. Therefore, each system has a maximum sequence length N_{ctx}, which is also known as *context size* [86]. If an input sequence is shorter than the context size, so-called *padding tokens* are added until the sequence is of length N_{ctx}. The positions of the padded tokens are then masked, such that these tokens require no additional computation.

4.3.2 Additional Components

Feed-forward network Each (self)-attention layer in the Transformer is followed by a fully-connected feed-forward network, which is applied position-wise and identically across positions. That is, each of the H-dimensional output elements of an attention layer is processed separately but with identical weights by the feed-forward network. The feed-forward network consists of one hidden layer with ReLU activation:

$$\text{FFN}(\boldsymbol{x}) = \boldsymbol{W}_2^T \max(0, \boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2.$$

Vaswani et al. [117] use dimension $H_{ff} = 4H = 2048$ for the hidden layer. The outputs of the feed-forward network are of dimension H, such that they can be processed by the attention sub-layer of the next layer. It is important to note that, since the feed-forward network is applied separately to each position, connections between different positions only exist in the attention sub-layers.

Residual connections All sub-layers in the Transformer are implemented with *residual connections*, also known as *shortcut connections* [39]. Residual connections facilitate optimization, allowing for increased network depth, which often results in enhanced model performance. In particular, due to the non-linear activation functions it is difficult for the layers of a neural network to learn an identity mapping. When stacking many layers, however, there are often some layers that do not improve model performance, which would be best expressed by such an identity mapping. Instead of directly attempting to learn an underlying mapping $H(\boldsymbol{x})$ with a given stack of layers, He et al. [39] implement residual connections by adding the input $\boldsymbol{x}$ to the output of the stacked layers. Instead of learning $H(\boldsymbol{x})$, this mechanism lets the layers learn the residual function $F(\boldsymbol{x}) := H(\boldsymbol{x}) - \boldsymbol{x}$.[5] This residual mapping is easier to optimize than the original

[5] If input and output dimensions differ, a linear transformation of $\boldsymbol{x}$ is used.

mapping [39]. In particular, in case of an underlying identity mapping, learning the relation $F(\boldsymbol{x}) = 0$ is easier than learning the identity mapping $H(\boldsymbol{x}) = \boldsymbol{x}$.

Layer normalization Another important component in the Transformer is *layer normalization* [4], which is computed as $\text{LayerNorm}(\boldsymbol{v}) = \gamma \frac{\boldsymbol{v}-\mu}{\sigma} + \beta$, where $\mu = \frac{1}{H}\sum_{k=1}^{H} v_k$, $\sigma^2 = \frac{1}{H}\sum_{k=1}^{H}(v_k - \mu)^2$, and γ, β are scale and bias parameters, respectively. Layer normalization is computed position-wise for the outputs of the attention and feed-forward sub-layers, after application of the residual connections. The normalized outputs are less dependent on outputs of previous layers, which stabilizes the optimization process [4]. Layer normalization is inspired by *batch normalization* [45], which has been the standard normalization method in neural networks. While layer normalization estimates mean and standard deviation by individually summing over all hidden units for each training example, batch normalization sums over batches of training examples, but individually for each hidden unit. Layer normalization has empirically led to better performances than batch normalization in NLP applications [104]. It has been argued that the relatively worse performance of batch normalization is caused by a high variance of batch-wise computed statistics of NLP data [104].

Positional encoding The attention mechanism itself is invariant to sequence ordering. That is, if the inputs are rearranged with a permutation σ, the attention mechanism produces a permutation of the initial output. However, the order of tokens entails important information about the meaning of an input sequence. Thus, information about the positions has to be used as input of the attention function to provide a mechanism that captures the order of a sequence. To allow for a meaningful extrapolation to sequences of arbitrary length, Vaswani et al. [117] implement a technique called *positional encoding*. Instead of learning an H-dimensional embedding for a position *pos*, a fixed mapping

$$\begin{aligned} \text{PE} : \mathbb{N} &\to \mathbb{N}^H \\ pos &\mapsto \text{PE}(pos) \end{aligned}$$

is used to represent positions in an H-dimensional vector space. Specifically, Vaswani et al. [117] obtain the i-th element of the encoding as

$$\text{PE}_i(pos) = \begin{cases} \sin\left(\frac{pos}{10000^{\frac{i}{H}}}\right), & \text{if } \ i \bmod 2 \equiv 0, \\ \cos\left(\frac{pos}{10000^{\frac{i-1}{H}}}\right), & \text{else.} \end{cases}$$

That is, each element of the vector is a sinusoid (as a function of the position) with different frequency for each dimension. The vector-valued positions are added to the embeddings of the tokens and subsequently used as input of the first attention layer. The idea that underlies this approach is that the network learns how to interpret the positional encodings. If a position is not encountered during training, the network should ideally infer the encoding of that position from other large positions observed during training.

4.3.3 Full Architecture

As mentioned, the original Transformer [117] is an encoder-decoder network. Therefore, the objective is to predict the individual factors in Eq. (4.1.5) given, as in the other Seq2Seq models that were considered in the previous sections. In the Transformer, the encoder maps the input sequence $\boldsymbol{x} = [x_1, \ldots, x_n]$ to a contextualized representation $\boldsymbol{z}(\boldsymbol{x}) = [\boldsymbol{z}_1(\boldsymbol{x}), \ldots, \boldsymbol{z}_n(\boldsymbol{x})]$ of this sequence. The decoder g uses this representation to generate an output sequence $\boldsymbol{y} = [y_1, \ldots, y_m]$ by iteratively predicting the probabilities $p(y_i |\ \boldsymbol{y}_{<i}, \boldsymbol{x}; \boldsymbol{\theta}) = g(\boldsymbol{y}_{<i}, \boldsymbol{z}(\boldsymbol{x}))$.

Encoder In the original Transformer, the encoder is composed of $L = 6$ identical layers. Each of these layers is split into two sub-layers. The first sub-layer applies multi-head self-attention to the outputs from the previous layer. The second sub-layer is a fully connected feed-forward network. As mentioned, both sub-layers are implemented with residual connections and followed by layer normalization.

Decoder The decoder is also composed of $L = 6$ identical layers in the original Transformer. The decoder takes two types of sequences as input: the output elements[6] and the sequence of contextualized representations. The former sequence is consumed by multi-head self-attention sub-layer with an autoregressive mask. The first sub-layer of the decoder is followed by a multi-head attention sub-layer. The queries of this attention sub-layer are the output elements of the self-attention sub-layer, while keys and values correspond to the output elements of the encoder (i.e., the contextualized representation). In contrast to the other two attention layers of the Transformer, this second sub-layer of the decoder is thus not a self-attention layer. Furthermore, no attention mask is applied for this sub-layer, such that the entire input sequence is used for prediction. Similarly to the encoder, the final sub-layer of the decoder is a fully

[6] During training, these correspond to the true output elements, whereas at test time the previously generated output elements are used.

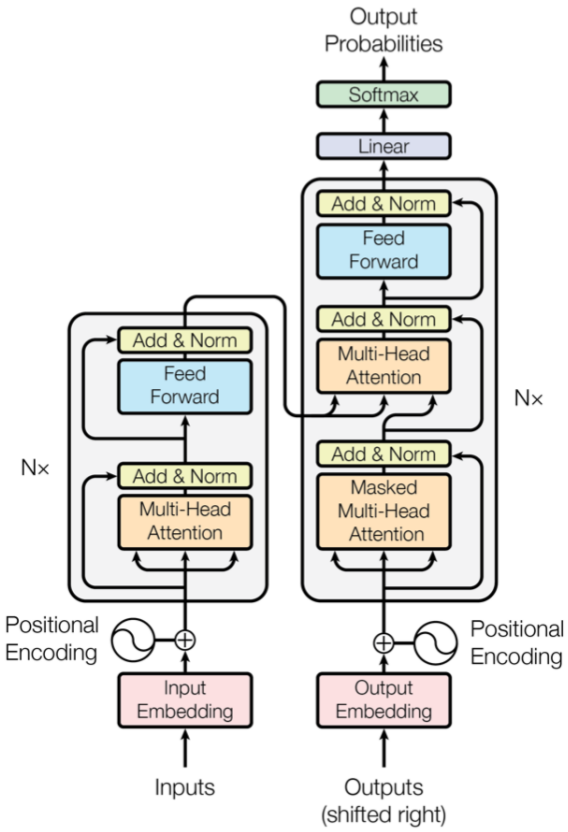

Figure 4.6: The original Transformer [117].

connected feed-forward network. All sub-layers employ residual connections followed by layer normalization.

4.3.4 Computational Perspective

For comparison, we now examine the computational characteristics of self-attention in light of the alternative approaches discussed in the previous section.

Parallel computation Transformers allow for parallel computation, since the attention mechanism relates elements in a non-sequential manner. This is reflected in a constant number of sequential operations in the input length. Thus, as CNNs, Transformers overcome computational inefficiencies of RNNs with regard to serial processing.

Path length Transformers further reduce the logarithmic maximum path length of ByteNet to a constant maximum path length in the distance of two input elements

[117]. The reduced path length is a consequence of the content-based addressing (rather than positional addressing) of the attention mechanism.

Layer complexity The scaled dot-product has a cost of $\mathcal{O}(H)$. For each output element, n compatibility functions are computed, resulting in a cost of $\mathcal{O}(H \cdot n)$ per output element. Since there are n elements, the computational cost per layer amounts to $\mathcal{O}(H \cdot n^2)$.

In total, attention sub-layers thus allow for greater training parallelization and reduced path length, at the cost of quadratic complexity in the input. While the reduced path length facilitates modeling long-range dependencies, the quadratic complexity contradicts this goal. However, several improvements have been proposed to reduce the quadratic complexity, as briefly outlined in the next section.

Note that the above considerations are only with respect to the attention operation, while projections and feed-forward layers are not taken into account. As will be shown in Chapter 7, however, the feed-forward layers in fact require the vast majority of compute in the Transformer, unless the context size N_{ctx} is very large compared to the embedding dimension H. On the other hand, the properties of constant maximum path length and parallel computation apply also when taking all components into account.

4.3.5 Modifications

Most systems that are based on the Transformer make modifications to the original version or use additional mechanisms. Some important variants are summarized in the following.

Position embeddings Instead of applying the positional encoding mechanism that was proposed by Vaswani et al. [117], most Transformer-based systems use *position embeddings*. Each position is embedded with an H-dimensional vector from a position embedding matrix $\boldsymbol{W}_p \in \mathbb{R}^{N_{ctx} \times H}$. As in the case of positional encodings, the first Transformer layer takes as input the sum of position and token embeddings.

Transformer decoder Many Transformer-based systems do not use the original encoder-decoder architecture. For instance, most autoregressive language models remove the entire encoder module and the second attention layer in the decoder. Such a system is referred to as *Transformer decoder* [60]. Since the decoder consists of an attention mask, future tokens are prevented from predicting the output at each position.

Transformer encoder Bidirectional systems, such as BERT, instead remove the entire decoder and train only with the encoder of the Transformer. Such a system is accordingly called *Transformer encoder* [25]. Since the encoder uses no attention masking, each token can attend to any other token of an input sequence.

Modified attention Several modifications of attention have been proposed. These methods reduce the quadratic time of attention in the context size by exploiting that the attention matrix $\boldsymbol{QK}^T$ is typically sparse. This allows for extending the context size significantly. We briefly describe the following two important examples:

- **Factorized attention**: Child et al. [16] introduced a factorized sparse version of attention in which every attention head only attends to an individual subset of all positions. The subsets are chosen such that the heads complement each other. This reduces the amount of compute while preserving the constant maximum path length. Factorized attention has for instance been applied in GPT-3 [11].
- **Locality-sensitive hashing attention**: Kitaev et al. [50] noted that it is inefficient to compute the matrix product $\boldsymbol{QK}^T$, since only the softmax of this product needs to be known. Attempting to calculate only the largest products for each query, Kitaev et al. [50] apply a variant of *locality-sensitive hashing* (LSH) [38]. This reduces the complexity in the context size from quadratic to logarithmic time with only marginal decreases in performance.

There exist many other extensions, such as using a recurrence mechanism between consecutive sequences [24], and numerous minor modifications, such as applying layer normalization to the inputs instead of the outputs of each sub-layer [85, 86]. Unfortunately, a more comprehensive discussion of those variations is out of the scope of this paper.

Chapter 5

State-of-the-Art Systems

In the previous chapter, we described the full architecture of the Transformer, which was originally proposed as an end-to-end system for machine translation. In this section, we discuss several NLP systems that make use of this architecture. The systems that are presented are part of our experiments, which will be discussed in Chapter 7. Building on the methods introduced in the previous chapters, the idea of this chapter is to make the reader familiar with the most important idiosyncratic features of these systems.

5.1 OpenAI GPT

The Generative Pre-trained Transformer (OpenAI GPT) [85] was the first large-scale NLP system that combined the Transformer architecture with transfer learning.

Input Each input sequence consists of a span of contiguous text of at most $N_{ctx} = 512$ tokens, obtained with a BPE tokenizer with a vocabulary size of $40,478$ (478 base characters and $40,000$ merges). Pre-BPE tokenization is performed with the spaCy[1] tokenizer. Each token embedding is summed with a position embedding, dispensing with the positional encoding of the original Transformer. OpenAI GPT uses no special tokens during pre-training,[2] but additional tokens are initialized during fine-tuning.

Pre-training objective Pre-training is performed with the standard unidirectional LM objective described in Chapter 3. Because the Transformer is a fixed-length model,

[1] `https://spacy.io/`

[2] Therefore, the first token of a sequence is not predicted and hence the corresponding term does not enter the loss.

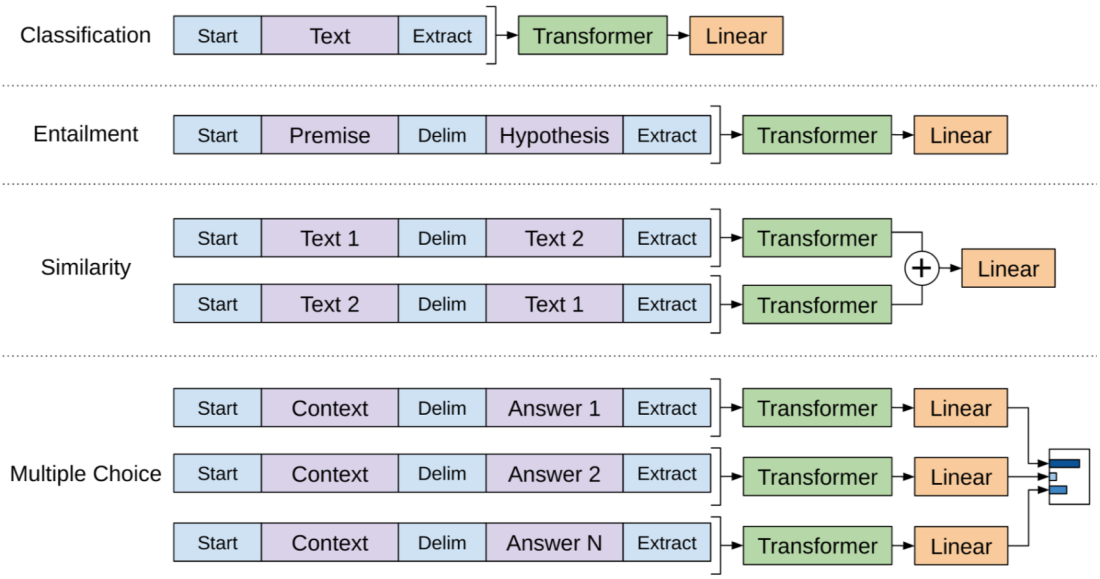

Figure 5.1: Fine-tuning GPT on different tasks [85].

only contexts of length ≤ 512 tokens are used to minimize the LM loss summed over the training corpus.[3]

Architecture Since OpenAI GPT is pre-trained with the LM objective, future tokens have to be masked, such that the output is not determined by these tokens. Therefore, OpenAI GPT uses the Transformer decoder, where each layer consists of masked multi-head self-attention followed by a feed-forward network. The activation function is the *Gaussian Error Linear Unit* (GELU) [40]. Radford et al. [85] trained a network with an embedding dimension of $H = 768$, $L = 12$ layers and $A = 12$ attention heads. As in the original Transformer, the feed-forward dimension was set to $4H$.

Pre-training procedure Radford et al. [85] trained for 100 epochs with a batch size of 64 on the BooksCorpus [134], which contains approximately 800M words.

Fine-tuning OpenAI GPT was fine-tuned on a variety of different supervised tasks. Some of these tasks are part of the General Language Understanding Evaluation (GLUE) benchmark [120], which is discussed in Chapter 6.4. As stated, OpenAI GPT depends on special tokens that are added to each input sequence during fine-tuning. In particular,

[3]Note that the objective given in Eq. (1) by Radford et al. [85] is not correct, because each sequence is factorized separately. The number of tokens used for the prediction of the next token is therefore much smaller than the context size, except for the last token of each sequence.

as shown in Figure 5.1, Radford et al. [85] apply a task-specific transformation to each input to make the pre-trained model more adaptable to the idiosyncrasies of each task:

- For all tasks, a start token <s> is added to the beginning and an end token <e> to the end of each input sequence. Due to the autoregressive input factorization, the contextualized representation of the last token contains information about the whole sequence. Therefore, Radford et al. [85] use the hidden state of the added end token to represent each sequence.
- For single-sentence classification tasks, apart from start and end tokens no additional tokens are added. The hidden state of the end token is fed through an additional linear layer during fine-tuning and the output is normalized with the softmax function.
- For entailment tasks, premise and hypothesis are concatenated and separated with a delimiter token <$>. The resulting sequence is processed by the pre-trained Transformer decoder with an additional linear layer that takes as input the embedding of the added end token. The output over the entailment classes is normalized with the softmax function.
- For similarity tasks, in addition to the separation with the delimiter token, each sentence pair is duplicated and arranged in reverse order. The pre-trained Transformer decoder is then applied separately to each of the resulting two sequences. This modification is implemented because, unlike in entailment tasks, the order of sentences is irrelevant in similarity tasks. Finally, the embeddings of the end tokens of both sequences are added element-wise and fed through a linear output layer. If the similarity task is formulated as a classification task, a softmax normalization is applied to the outputs of the linear layer.
- For multiple choice tasks, such as QA and commonsense reasoning, a new input sequence is created for each possible answer by using the delimiter token to concatenate the question and the context this question appears in to the answer. The resulting input sequences are processed independently by the pre-trained Transformer decoder and fed through separate linear output layers. The probabilities for each answer are then obtained as a softmax score over the outputs.

Note that both the added tokens as well as the weights of the additional linear layer are randomly initialized for each task. The pre-trained weights together with the additional linear layer are then updated by minimizing the combined loss over the labeled dataset,

where the loss function is usually the cross-entropy loss. Radford et al. [85] additionally add an auxiliary language modeling loss (weighted with a factor λ) to this objective.

5.2 BERT

The introduction of BERT [25] marked a major breakthrough in the development of Transformer-based NLP systems. The main innovation of BERT compared to OpenAI GPT was to replace the unidirectional pre-training objective with a bidirectional objective.

Input Tokenization of the inputs is performed with the WordPiece tokenizer with a vocabulary of $30,522$ tokens. Each input example consists of two "sentences", where sentence refers to an arbitrary span of contiguous text. The combined maximum length of the sentences is $N_{ctx} = 512$ tokens. In 50% of the time the two sentences are consecutive, while in the other 50% of the time the second sentence is randomly drawn from the corpus. The first token of every input example is a special `[CLS]` token. The contextualized representation corresponding to this symbol is used to represent an input sequence in classification tasks. In order two differentiate between the two sentences they are furthermore separated by a `[SEP]` token. Additionally, as will be discussed below, some input elements of each sequence are randomly replaced with `[MASK]` tokens. Each input element is embedded as the sum of a token embedding, a position embedding and a so-called segment embedding. The segment embedding indicates whether a token is located in the first or second sentence.

Pre-training objective BERT is pre-trained with the sum of the masked language modeling (MLM) loss and the next sentence prediction (NSP) loss.

- **MLM**: BERT applies a modified version of the basic MLM objective described in Chapter 3 to reduce the mentioned pre-train/fine-tune discrepancy arising in MLM. Specifically, BERT randomly chooses 15% of all tokens for prediction, but only replaces these tokens with a `[MASK]` symbol in 80% of the time. In 10% of the time the tokens are replaced with a random token and in 10% of the time the initial token is kept.
- **NSP**: Devlin et al. [25] added the NSP objective to increase the performance of BERT in tasks that require an understanding of the relationship between two sentences, such as entailment or similarity tasks. Given an input, NSP consists of predicting whether the second sentence follows the first sentence, or whether it

was randomly drawn from the corpus. The binary output of this task (with labels `IsNext`/`NotNext`) is predicted by feeding the contextualized representation of the `[CLS]` token through a linear output layer followed by a softmax normalization. The normalized outputs are then evaluated against the true labels using the cross-entropy loss. Note that NSP is an instance of CTL.

Architecture Due to the bidirectional pre-training objective, BERT does not rely on an attention mask and thus uses the Transformer encoder. The activation function is again the GELU [40]. Devlin et al. [25] trained two different versions: $\text{BERT}_{\text{BASE}}$, with $H = 768$, $L = 12$, $A = 12$, and $\text{BERT}_{\text{LARGE}}$, with $H = 1024$, $L = 24$, $A = 16$. For both variants, the feed-forward size was set to $4H$.

Pre-training procedure Devlin et al. [25] trained for 1M steps, corresponding to approximately 40 epochs, with a batch size of 256 over a combination of the BooksCorpus (800M words) [134] and English Wikipedia (2,500M words) [1]. The first 90% of the steps were performed on short inputs with sequence length 128, while only the last 10% of the steps consisted of inputs with sequence length 512. Note that Devlin et al. [25] furthermore in all cases randomly inject short sequences[4] with a probability of 10% to additionally alleviate the pre-train/fine-tune discrepancy, since most sequences during fine-tuning are rather short. Masking was performed once prior to training, which is also known as static masking [62]. The corpus was duplicated 10 times, such that each sequence is processed in different ways when training for multiple epochs. However, since it was trained for 40 epochs, each sequence was processed with exactly the same masking pattern four times during training.

Fine-tuning For sequence classification tasks, BERT is fine-tuned by feeding the contextualized representation of the `[CLS]` token through a sequence classifier, consisting of a linear layer and a softmax normalization. Since BERT is pre-trained with pairs of sentences that are separated by the `[SEP]` token, in contrast to GPT, sentence pairs are directly processed by a single Transformer during fine-tuning. In addition, Devlin et al. [25] fine-tuned on token-level tasks, in which the hidden states of other tokens of a sequence are fed though a linear output layer. For instance, to fine-tune BERT on SQuAD 1.1 [89], Devlin et al. [25] predicted the start and end tokens of an answer. Furthermore, for the multiple choice task SWAG [131], Devlin et al. [25] applied a similar approach as Radford et al. [85]. That is, a "question" was concatenated to each choice,

[4]The length is picked uniformly at random between 1 and the maximum sequence length.

and the resulting sequences were encoded separately with the pre-trained Transformer. The contextualized representations for each choice were then converted to scores.

5.3 OpenAI GPT-2

Radford et al. [86] obtain OpenAI GPT-2 by scaling-up OpenAI GPT, making only minor changes to the original model. OpenAI GPT-2 achieved state-of-the-art results on several language modeling datasets in a zero-shot setting.

Input Radford et al. [86] introduced a byte-level BPE tokenizer with a vocabulary size of $50,257$ to tokenize the inputs. The fixed-length context size was increased from 512 to 1024 tokens compared to OpenAI GPT. Furthermore, a single special token was added to indicate the start and end of each input sequence during pre-training. As in OpenAI GPT, a position embedding is a added to each token embedding.

Pre-training objective Radford et al. [86] used the standard unidirectional LM objective. Thus, OpenAI GPT-2 is pre-trained with the same objective as OpenAI GPT.

Architecture Radford et al. [86] scale-up the model size compared to OpenAI GPT by training four different architectures. The largest of these architectures, which is used for the final model that is referred to as GPT-2, is a Transformer decoder with an embedding size of $H = 1600$, $L = 48$ layers, $A = 25$ heads and feed-forward dimension of $H_{ff} = 4H$. Minor modifications were made, such as applying layer normalization to the inputs instead of the outputs of each sub-layer and using a different weight initialization method.

Pre-training procedure Radford et al. [86] trained with a batch size of 512 for on WebText [86], which has an approximate size of 40GB.[5] Neither the number of training steps nor the number of epochs is reported.

Fine-tuning OpenAI GPT-2 was not fine-tuned. Instead, several tasks were performed in a zero-shot setting.

5.4 RoBERTa

RoBERTa [62] is based largely on BERT, but achieves a better performance than BERT by making several important changes to the original model.

[5] The word count is not public.

Input RoBERTa uses a byte-level BPE tokenizer with a vocabulary consisting of $50,265$ tokens. Liu et al. [62] initially experiment with different input formats. For the final version they use contiguous sentences of length $N_{ctx} = 512$. However, if a document is contains less than 512 tokens, Liu et al. [62] add sentences from the next document and separate the documents with a special separator token. Each final input sequence is furthermore enclosed by special start and end tokens. As in BERT, some tokens of each sequence are randomly replaced with a `[MASK]` token. Each token embedding is summed with a positional embedding.

Pre-training objective Liu et al. [62] removed the NSP objective of BERT and trained only with the MLM objective. The masking procedure itself is equivalent to the procedure of Devlin et al. [25], although static masking is replaced by a dynamic approach, as will be described below.

Architecture Liu et al. [62] performed several ablation studies using the $\text{BERT}_{\text{BASE}}$ architecture, but the final results were obtained with the $\text{BERT}_{\text{LARGE}}$ architecture, which is a Transformer encoder with $H = 1024$, $L = 24$, $A = 16$ and $H_{ff} = 4H$.

Pre-training procedure The final version of RoBERTa was trained on the combination of the following corpora: BooksCorpus [134], English Wikipedia [1], CC-News [62], OpenWebText [32] and Stories [114]. The total number of words is not public, but the combined size of the dataset is 160GB. For comparison, BERT was trained on a 13GB dataset. The best performing model was trained for $500,000$ steps with a batch size of $8,000$. In contrast to Devlin et al. [25], Liu et al. [62] implement dynamic masking, where a new mask is selected for the input sequences at each step during training.

Fine-tuning Though RoBERTa is not pre-trained with pairs of sentences, fine-tuning on sentence-pair tasks follows the same approach as in BERT. That is, sentences are separated by a special token and processed by a single Transformer. For classification tasks, fine-tuning is achieved by feeding the contextualized representation of the special start token of a sequence through a linear output layer with softmax. RoBERTa was also fine-tuned on token-level tasks. For instance, on SQuAD 1.1. Liu et al. [62] applied the same fine-tuning procedure as Devlin et al. [25]. Moreover, for multiple choice tasks, such as RACE [54], Liu et al. [62] also encoded concatenated question-answer sequences separately with the pre-trained Transformer to obtain a softmax score for each answer.

Part II

Benchmarking

Chapter 6

Background

The summary of the different systems in the previous chapter shows that different systems are often trained under very diverse conditions. For example, GPT and BERT were trained for much more steps but with a significantly lower batch size than RoBERTa. At the same time these systems employ different pre-training objectives. As stated, to disentangle such factors, in this part we conduct a systematic benchmarking study, in which we train and evaluate systems with a variety of different configurations.

6.1 Related Work

There exist several recent attempts to conduct such a systematic investigation. One line of research empirically derives generalization results for large neural NLP systems. Rosenfeld et al. [92] study how the generalization error of language models depends on model and dataset size. Regarding model size, they provide an approximation of the test loss, assuming that a language model is scaled with respect to a predefined scheme, such as increasing solely the embedding dimension. A related but much more comprehensive study was conducted recently by Kaplan et al. [47], examining power laws of the test loss when scaling large neural language models with respect to a broad variety of different dimensions. These dimensions include architectural hyperparameters, model size, dataset size, the number of training steps and the batch size. A central question in the work of Kaplan et al. [47] is how these factors can be combined to attain an optimal performance given a fixed amount of compute.

Compute efficient training is also investigated by Li et al. [59], recognizing that an optimal allocation of computational resources is crucial for improving model performance. Considering MLM pre-training, Li et al. [59] examine specifically how the

number of training steps and the batch size should be chosen in the relation to the model size.

In a large-scale study, Raffel et al. [88] cover an even broader variety of modeling scenarios than Kaplan et al. [47], but train a much smaller number of systems for each scenario. For instance, Raffel et al. [88] include several variants of the Transformer, different pre-training objectives and various fine-tuning strategies in their analysis. Finally, based on their observations, Raffel et al. [88] also scale-up a system to 11B parameters.

6.2 Objectives of the Study

The overall aim of our study is to evaluate the effect of different modeling choices in Transformer-based NLP systems on model performance.[1] Because an automated search over the full models is too costly, the idea is to use *down-scaled* systems, such that a larger number of systems can be evaluated. Ideally, the insights obtained from benchmarking such down-scaled versions can then be leveraged to train larger networks.

We categorize the different modeling choices that we consider into three broad categories, which are outlined in the following. For each category, we examine only a subset of all possible scenarios, but we attempt to focus on the most important choices that have to be made when pre-training a Transformer.

6.2.1 Comparison of Pre-Training Tasks

As mentioned in Chapter 3, the choice of the unsupervised pre-training objective is of central importance. Many different pre-training objectives have been proposed in recent years. Arguably the most prevalent ones include denoising objectives, such as MLM, as well as the traditional LM objective. In this study, we concentrate on the following objectives:

MLM & NSP The combination of MLM and NSP was used as the pre-training objective of BERT [25]. Devlin et al. [25] evaluated the effect of removing the NSP task, thus pre-training solely with the MLM task, finding a significant performance decline of the fine-tuned versions on MNLI and QNLI, both of which are sentence-pair tasks. Devlin et al. [25] mentioned that exactly the same data was used for pre-training both variants. However, the input format of the regular version of BERT (MLM & NSP) consists of segment pairs, either sampled contiguously from the same document (positive examples)

[1] Performance can refer to both the validation loss and the performance on downstream tasks in this work.

or from different documents (negative examples) with equal probability. If the randomly generated segment pairs were also used when pre-training without the NSP task, then the model learned from incorrect data without being able to distinguish the positive from the negative examples and thus adapt during training. The possibility that Devlin et al. [25] might have mistakenly used the same input format for both versions was also mentioned by Liu et al. [62] as an explanation for the observed performance gap.

MLM In RoBERTa, Liu et al. [62] removed the NSP task and pre-trained only with the MLM task, after showing in an ablation study that pre-training without the NSP task does not hurt the performance on several downstream tasks. However, Liu et al. [62] did not report results for QNLI, the downstream task on which Devlin et al. [25] observed the largest performance gap. Liu et al. [62] did also not state whether original segments that were replaced with samples were kept and used in other input sequences, or whether these original segments were discarded. In the latter case, the size of the dataset is effectively reduced by a significant portion, while the number of epochs is effectively increased.

LM Although it has been shown that LM is inferior to other unsupervised pre-training tasks [25], LM is still widely used, for instance in the GPT-n series [11, 85, 86]. We include LM in our analysis because it has traditionally been one of the most important training objectives in neural NLP, as outlined in Chapters 3 and 4.

In the remainder of this study, we will label systems pre-trained with MLM & NSP as BERT-style, systems pre-trained with MLM as RoBERTa-style, and systems pre-trained with LM as GPT-2-style.[2]

6.2.2 Comparison of Different Shapes

In some domains, such as CV, it has been observed that the performance of a neural network depends significantly on the choice of architectural hyperparameters, such as width or depth [110]. In contrast, Kaplan et al. [47] observed a similar language modeling test loss over a wide range of shape parameters, such as depth, width or the number of attention heads. Similarly, for masked language models, Li et al. [59] found that the validation loss does not depend strongly on the model shape. This holds true also for

[2]We refer to GPT-2-style instead of GPT-style systems, because we use the class *GPT2Model* from the Hugging Face transformers library [126]. The main reason for this choice is that we use a byte-level BPE tokenizer. Furthermore, the class GPT2Model allows for a more flexible modeling than the class GPTModel [126] does, since several components are fixed in the latter.

the MNLI validation accuracy of fine-tuned systems.

To investigate the effect of different shapes, we consider the following three architectural hyperparemters of Transformer-based NLP systems:

Depth In neural networks, depth is given by the number of layers L. It has been stated that stacking many layers in a Transformer-based system can be somewhat inefficient and does not always lead to a considerable increase in performance [56].

Width In neural NLP systems, width corresponds to the embedding dimension H. Increasing the embedding dimension has in general produced slightly better results than increasing the number of layers in Transformer-based systems [56, 59, 88].

Attention Heads As described in Chapter 4, attention heads are used to discriminate between different regions of the embedding space. In most applications of the Transformer, the number of attention heads A is set in fixed relation to the embedding dimension, such that the latter is 64 times larger than A. It has been reported that the performance decreases for larger ratios [11, 117].

Note that there exist several other choices, such as the feed-forward dimension H_{ff}, but examining the entire spectrum of possible shapes is unfortunately out of the scope of this study.

6.2.3 Effect of Model Size, Training Steps and Batch Size

As stated, several recent studies have investigated the problem of compute efficient training of Transformer-based systems [47, 59, 88]. The consensus among these studies is that, under a restricted compute budget, optimal performance is achieved by training very large models and stopping training well before convergence. Furthermore, additional compute should rather be used to increase the batch size instead of training for more steps.

To examine convergence characteristics, we monitor the pre-training validation loss of several systems and test how this loss responds to different model sizes and shapes. Additionally, we conduct experiments regarding the effect of the batch size and the number of training steps. In particular, we evaluate how the training time and the model performance depend on both factors.

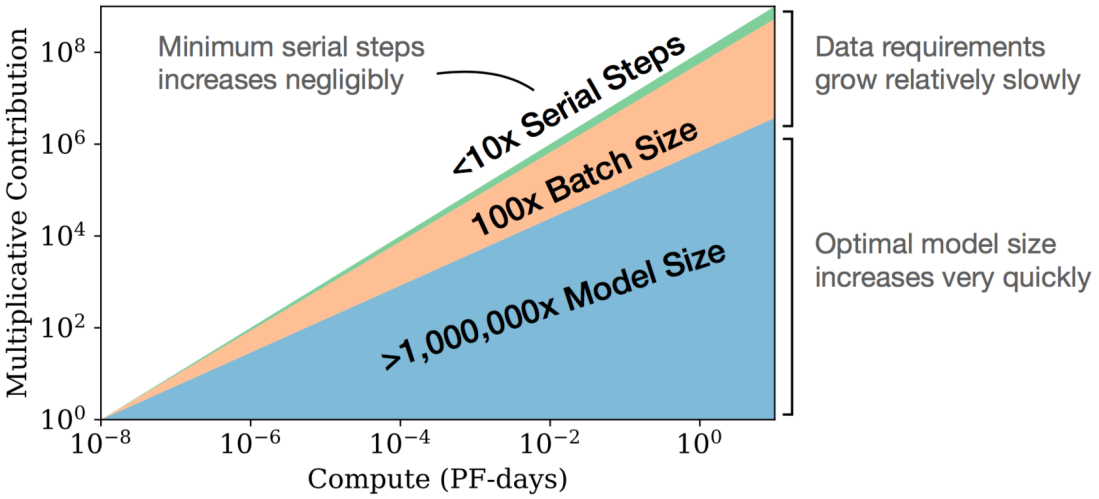

Figure 6.1: Efficient allocation of compute for Transformer-based LMs according to Kaplan et al. [47].

6.3 WikiText-103

We pre-train all systems on WikiText-103 [69], which is a large-scale text corpus introduced by Merity et al. [69] with the purpose of training and evaluating language models on long-range contexts. WikiText-103 has mainly been used as an evaluation dataset for different systems [24, 86, 105]. A prominent system which was pre-trained on WikiText-103 is ULMFiT [44].

In the following we provide information regarding content and size of WikiText-103, as well as describe the pre-processing steps that were undertaken when creating WikiText-103. Furthermore, we compare WikiText-103 to the datasets on which the original systems of our down-scaled versions were pre-trained.

Content WikiText-103 is composed of 23,805 *Good* and 4,790 *Featured* Wikipedia articles. Articles from these two categories, classified based on reviews by human editors, are considered of the highest quality and make up less than 1% of all Wikipedia articles. Amongst other criteria, the articles are selected with regard to neutrality, factual accuracy and breadth of coverage.[3]

Size The training set of WikiText-103 has a total size of 103,227,021 words (and other symbols) contained in 28,475 articles. The average article thus has a length of 3,625

[3]For details see `https://en.wikipedia.org/wiki/Wikipedia:Good_articles`.

Pre-processed Text	Original Text
Nebraska Highway 88 (N-88) is a highway in northwestern Nebraska.	Nebraska Highway 88 (N @-@ 88) is a highway in northwestern Nebraska .
These numbers may be computed by the recurrence relation <formula> Eric W. Weisstein conjectured , and McKay et al . (2004) proved , that the same numbers count the (0 @,@ 1) matrices for which all eigenvalues are positive real numbers .	These numbers may be computed by the recurrence relation $\sum_{k=1}^{n}(-1)^{k-1}\binom{n}{k}2^{k(n-k)}a_{n-k}$. Eric W. Weisstein conjectured, and McKay et al. (2004) proved, that the same numbers count the (0,1) matrices for which all eigenvalues are positive real numbers.

Table 6.1: Original examples from Wikipedia [123, 124] and modified WikiText-103 training data.

words, which allows for learning long-range dependencies [87]. In addition to the training set, when comparing different pre-trained systems by their validation loss, we make use of the validation set of WikiText-103. The validation set consists of 60 articles and $217,646$ words. Furthermore, there exists a test set, which contains 60 articles and $245,569$ words. However, we do not make use of the test set. The vocabulary of the combined train, test and validation sets is made up of $267,735$ unique terms. We recognize that state-of-the-art NLP systems are usually pre-trained on datasets with considerably longer contexts and larger sizes. However, due to limited compute power and because we benchmark *down-scaled* systems, we chose to pre-train on WikiText-103.

Pre-processing details Merity et al. [69] performed several pre-processing steps, such as removing sections that are primarily made up of lists, or replacing LaTeX code with a special <formula> symbol. In particular, the text was pre-processed with the tokenizer from the *Moses* toolkit [51]. This tokenizer separates punctuation marks from text, preserving special sequences such as dates or URLs, and performs several normalization steps [27]. Specifically, hyphenated words are separated with two at signs (e.g., *non-hyphenated* → *non @-@ hyphenated*). Merity et al. [69] additionally modified numbers by inserting at signs before and after punctuation marks (e.g., *1,200* → *1 @,@ 200*). Note that, apart from punctuation mark separation, no tokenization (i.e., in the strict sense of splitting a text into smaller pieces) was performed. In the version used by Merity et al. [69], as a final step, rare words were replaced with an <unk> token. However, we have downloaded the raw version of WikiText-103, which contains all words.[4]

[4]The data was obtained from https://www.salesforce.com/products/einstein/ai-research/the-wikitext-dependency-language-modeling-dataset/.

System	Corpora	Size	Word Count
GPT	BooksCorpus [134]	4GB	800M
BERT	BooksCorpus [134], English Wikipedia [1]	13GB	3,300M
GPT-2	WebText [86]	40GB	Not public
RoBERTa	BooksCorpus [134], English Wikipedia [1], CC-News [62], OpenWebText [32], Stories [114]	160GB	Not public

Table 6.2: Pre-training corpora used by the original systems of the down-scaled versions.

Comparison with other corpora As mentioned, WikiText-103 is significantly smaller than most of the pre-training corpora of modern NLP systems. For instance, Devlin et al. [25] report that BERT was trained on $3,300$M words, which is approximately 32x the size of WikiText-103. Table 6.2 lists the corpora on which the original systems of our down-scaled versions were trained.[5] Aside from the generally large amounts of data compared to WikiText-103, it can be observed that the pre-training datasets vary considerably in size across the different systems, which makes a fair comparison difficult. In contrast, pre-training on the same corpus allows us to systematically evaluate different components of the down-scaled systems, since we can largely exclude factors such as the amount of pre-training data as the cause of different performances.

6.4 The GLUE Benchmark

We fine-tune and evaluate our systems on the *General Language Understanding Evaluation* (GLUE) benchmark [120], which is a collection of natural language understanding tasks introduced with the goal of evaluating NLP systems on a broad spectrum of tasks and datasets. It is one of the most prominently featured NLP benchmarks [25, 56, 62, 85, 129] and has in particular been used to evaluate the performance of BERT, RoBERTa and GPT. Therefore, fine-tuning on the GLUE benchmark may provide a good indication of the performance of down-scaled version compared to the original systems.

The GLUE benchmark consists of nine tasks from different genres accompanied by datasets of varying size, which encourages sample-efficient multi-task learning. While

[5]Sizes and word counts were obtained from the articles describing the respective original systems.

Corpus	|Train|	|Test|	Task	Metrics	Domain
			Single-Sentence Tasks		
CoLA	8.5k	**1k**	acceptability	Matthews corr.	misc.
SST-2	67k	1.8k	sentiment	acc.	movie reviews
			Similarity and Paraphrase Tasks		
MRPC	3.7k	1.7k	paraphrase	acc./F1	news
STS-B	7k	1.4k	sentence similarity	Pearson/Spearman corr.	misc.
QQP	364k	**391k**	paraphrase	acc./F1	social QA questions
			Inference Tasks		
MNLI	393k	**20k**	NLI	matched acc./mismatched acc.	misc.
QNLI	105k	5.4k	QA/NLI	acc.	Wikipedia
RTE	2.5k	3k	NLI	acc.	news, Wikipedia
WNLI	634	**146**	coreference/NLI	acc.	fiction books

Table 6.3: Summary of the GLUE tasks [120].

the GLUE benchmark is model-agnostic, due to the diverse spectrum and the inclusion of data-scarce tasks it is especially useful for evaluating pre-trained systems [120]. In the following we provide a short description of the different tasks, which were originally summarized by Wang et al. [120]. If not explicitly mentioned, the evaluation metric is accuracy. For each task, the corresponding data is split into a training, a test and a validation set.

CoLA The *Corpus of Linguistic Acceptability* (CoLA) [121] is a dataset consisting of example sentences from the linguistics literature, labeled as either grammatically acceptable or unacceptable, which is known as an *acceptability judgment* in linguistics.[6] CoLA was introduced to assess the linguistic competence of a system by testing its ability to identify grammatically acceptable sentences without formal training in grammar. The input consists of single sentences evaluated by a binary classifier. Performance is usually measured with the *Matthew's correlation coefficient* (MCC) [66]. As described in Chapter 2, this metric is especially suitable for evaluating unbalanced binary classification tasks.

SST-2 The *Stanford Sentiment Treebank* (SST-2) [106] is a collection of sentences from movie reviews, classified into positive and negative sentiment by human annotators. The task is to predict the sentiment of a given sentence. Note that, as CoLA, SST-2 is a single-sentence task.

[6] In linguistics, an *acceptability judgment* is a subjective report of a native speaker regarding the grammatical wellformedness, nativeness, or naturalness of a text [76].

MRPC The *Microsoft Research Paraphrase Corpus* (MRPC) [26] consists of sentence pairs from online news sources. Each pair comes with a binary judgment indicating whether human raters labeled the two sentences as semantically equivalent. Note that, since classes are imbalanced (68% positive), in addition to accuracy the F1 score is used as an evaluation metric.

QQP The *Quora Question Pairs* (QQP)[7] are question pairs obtained from Quora, which is a question-and-answer website. As in MRPC, the task is to determine whether a pair of questions have the same meaning. Again, both F1 and accuracy are reported, since the classes are imbalanced (63% negative). As stated by Wang et al. [120], the label distribution of the test set differs from the distribution of the training set.

STS-B The *Semantic Textual Similarity Benchmark* (STS-B) [13] consists of sentence pairs extracted from news headlines, video and image captions and other sources. In contrast to the other two similarity tasks, in STS-B sentence pairs are assessed according to their *degree* of semantic similarity. Each pair is accompanied by a similarity score ranging from 1 to 5, assigned by human annotators. Performance is usually measured by either the Pearson or the Spearman correlation of predicted scores with human annotations.

MNLI The *Multi-Genre Natural Language Inference* (MNLI) Corpus [125] is a large-scale collection of crowd-sourced sentence pairs. For each sentence pair, the task is to assess whether the second sentence (the hypothesis) is an *entailment*, a *contradiction* or *neutral* with respect to the first sentence (the premise). The MNLI corpus was motivated by the lack of a multi-genre NLI corpus. Therefore, premises are drawn from ten highly diverse sources, such as government reports, travel guides and telephone conversations. For each premise, human annotators created a hypothesis from each of the three classes (*entailment*, *contradiction*, *neutral*), i.e., three hypotheses for each premise, which ensures that the labels are equally distributed. Note that MNLI has two test and validation sets, respectively: one that matches the genres of the training data and one that contains data from different genres, allowing for cross-genre generalization evaluation.

QNLI *Question-answering NLI* (QNLI) [120] is a modified version of SQuAD 1.1 [89], a collection of crowd-sourced question-answer pairs. In the original version each question, written by a human annotator, has an answer which is found in a specific text span of a

[7] `data.quora.com/First-Quora-Dataset-Release-Question-Pairs`

corresponding Wikipedia paragraph. The original task is to predict the text span that contains the answer. Wang et al. [120] reformulated this task as a binary sentence-pair classification by construing multiple question-sentence pairs from each paragraph, where sentences in positive examples contain the correct answer, while conversely sentences in negative examples do not contain the correct answer.

RTE *Recognizing Textual Entailment* (RTE) is a collection of multiple smaller datasets from a series of annual textual entailment challenges: RTE-1 [22], RTE-2 [37], RTE-3 [31] and RTE-5 [9]. The data consists of sentence pairs drawn from news and Wikipedia articles. The task is to predict whether the second sentence is either *entailed* or *not entailed* in the first. While RTE is similar to MNLI (although MNLI consists of three classes), the amount of training and test data is much smaller compared to MNLI.

WNLI *Winograd NLI* (WNLI) [120] is a modified version of the Winograd Schema Challenge [58]. In the original task, given a statement (extracted from a collection of fiction books), a system must identify the referent of an ambiguous pronoun from a list of choices. This task is especially difficult, because it is designed such that the structure of the sentence does not help to disambiguate the sentence. Wang et al. [120] transformed the original task into a binary textual entailment task by presenting a hypothesis that implies that a specific referent from the list of choices is correct. Given the pronoun in the corresponding premise, the task is to predict whether this implication is correct.

Chapter 7

Experiments

7.1 Definition of the Model Size

We follow Kaplan et al. [47] and use the approximate number of non-embedding parameters to define the model size, which we denote as N_{model}. The embedding parameters consist of all token, position and (if present) segment embeddings. The number of embedding parameters does not depend on the network depth, and when scaling width and/or depth, it is a sub-leading term of the total number of parameters. Furthermore, the number of FLOPs related to embedding (and de-embedding) is also sub-leading term of the total number of FLOPs. Consistent with this is the observation of Kaplan et al. [47] that discarding the number of embedding parameters when calculating model size and amount of compute results in significantly cleaner scaling laws. Since the share of embedding parameters decreases significantly for larger models, similarly to Kaplan et al. [47] we expect that discarding the number of embedding parameters allows for a better generalization of our results to large models. Another advantage of defining the model size as the number of non-embedding parameters is that this number is closely linked to the number of (non-embedding related) FLOPs. This enables us to design benchmarking scenarios by training different models of comparable size, which at the same time require roughly similar amounts of computation.

7.1.1 Number of Non-Embedding Parameters

Omitting biases and other sub-leading terms, the number of non-embedding parameters, which is our definition of the model size, is given by

$$N_{\text{model}} := 12LH^2, \tag{7.1.1}$$

where we have assumed that $H_k = H_v = \frac{H}{A}$ and $H_{ff} = 4H$. Therefore, per layer there are approximately $12H^2$ non-embedding parameters. This number can be derived from the following three steps performed in each layer of a Transformer:

1. Input projection For each attention head, the queries, keys and values of dimension $\frac{H}{A}$ are obtained with the three matrices $\boldsymbol{W}_i^Q$, $\boldsymbol{W}_i^K$, and $\boldsymbol{W}_i^V$, which are each of size $H \times \frac{H}{A}$. In total, the input projection thus consists of $3 \cdot A \cdot \frac{H^2}{A} = 3H^2$ parameters.

2. Output projection First, note that performing attention on the projected inputs of dimension $\frac{H}{A}$ involves no additional parameters. The concatenated attention results are projected back to dimension H with the $H \times H$ matrix $\boldsymbol{W}^O$. Therefore, the output projection involves an additional set of H^2 parameters.

3. Feed-forward network The last sub-layer of each layer consists of applying a feed-forward network to the output projections. There exist $H \cdot 4H$ connections between the output projections and the neurons of the inner-layer, and another $4H \cdot H$ connections from the inner-layer to the final output neurons. This step hence involves $8H^2$ parameters.

Note that the feed-forward network accounts for the majority of non-embedding parameters, followed by the input and output projections, respectively.

7.1.2 Relation to FLOPs

As stated, the number of non-embedding parameters is closely linked to the number of non-embedding related FLOPs. We start by deriving the number of FLOPs per token and forward pass for GPT-2-style systems, where sub-leading terms such as biases and layer normalization are again omitted. The FLOPs for the matrix-vector operations that are performed in the following steps can be found in Appendix B.1.

1. Input projection The matrix-vector products of each per-layer input with $\boldsymbol{W}_i^Q$, $\boldsymbol{W}_i^K$, and $\boldsymbol{W}_i^V$ involve approximately $3 \cdot 2 \cdot H \cdot \frac{H}{A}$ FLOPs per attention head. Considering all attention heads, the input projection thus requires approximately $6H^2$ FLOPs per token.

2. Attention The computation of the attention operation can be divided into two sub-components:

- **Computation of the weights**: On average, $\frac{N_{ctx}}{2}$ attention weights have to be computed per input token, since on average half of the tokens are masked for each

input token. Computation of a dot-product attention weight requires approximately $2\frac{H}{A}$ FLOPs per head. In total, the computation of the attention weights hence involves approximately $N_{ctx}H$ FLOPs per token.

- **Computation of the weighted sum**: Since only half of the tokens are summed on average, given the attention weights, calculation of the weighted sum of the values has an average cost of approximately $N_{ctx}H$ FLOPs for each token.

3. Output projection The vector matrix product of the attention outputs with $\boldsymbol{W}^O$ requires approximately $2H^2$ FLOPs for each token.

4. Feed-forward network The feed-forward network consists of two consecutive matrix multiplications, where each matrix contains $4H^2$ parameters. Thus, the feed-forward network requires approximately $2 \cdot 2 \cdot 4H^2 = 16H^2$ FLOPs per token.

The number of FLOPs per token and forward pass in GPT-2-style systems, which we denote by $C_{forward}$, can hence be approximated as

$$\begin{aligned} C_{forward} &\approx L(6H^2 + N_{ctx}H + N_{ctx}H + 2H^2 + 16H^2) \\ &= 24LH^2 + 2LN_{ctx}H \\ &= 2N_{\text{model}} + 2LN_{ctx}H. \end{aligned} \tag{7.1.2}$$

BERT-style and RoBERTa-style systems require slightly more FLOPs than GPT-2-style systems, because these systems have no autoregressive attention mask. Hence, in both steps of the attention operation above, the computational cost is approximately twice as much, i.e., $2N_{ctx}H$ in each step. Therefore, BERT-style and RoBERTa-style systems require approximately $2N_{\text{model}} + 4LN_{ctx}H$ FLOPs per token and forward pass. As mentioned by Kaplan et al. [47], if $H > N_{ctx}/12$, the context-dependent term in Eq. (7.1.2) only accounts for a relatively small fraction of the compute of GPT-style systems. In particular, when increasing H, the importance of the context-dependent term diminishes. For BERT-style and RoBERTa-style systems the context-dependent term becomes small if $H > N_{ctx}/6$. Both constraints are satisfied by a large margin for all our systems, especially since we mainly train on rather short sequences. The backward pass requires approximately twice as much compute as the forward pass [47], such that the total amount of non-embedding related compute per token and training step can be approximated as

$$C := 6N_{\text{model}}. \tag{7.1.3}$$

System	Tokenizer	Partition	Number of Tokens	
			Total	Average
BERT-Style	Wordpiece (BertTokenizer)	Short	110,888,186	110.04
		Long	43,274,856	375.52
RoBERTa-Style	Byte-level BPE (RobertaTokenizer)	Short	70,025,709	110.31
		Long	27,692,351	457.04
GPT-2-Style	Byte-level BPE (GPT2Tokenizer)	Short	70,564,106	111.16
		Long	27,729,551	457.65

Table 7.1: Total corpus length and average sequence length of the tokenized inputs.

7.2 Data Preparation and Input Format

We train on the training set of WikiText-103. Each Wikipedia section is a separate document in our setting. In all cases, we train for the first 90% of the inputs on short sequences with a maximum length of 128 tokens. For the remaining 10% of inputs, we train on sequences with a maximum length of 512 tokens. This approach speeds up training, which is essential in our scenario, and was also implemented by Devlin et al. [25]. Training on the long sequences is mainly intended for learning the position embeddings, because some GLUE tasks contain long-range inputs. When inspecting the pre-training validation loss, we adjust the evaluation sequence lengths to the lengths of the training sequences, so that the validation data follows the same distribution as the training data. This causes the validation loss on the long sequences to start at a slightly higher point than the final validation loss from the short sequences. Further, note that we lower the batch size by a factor of four when training on the long sequences, because for the long sequences this is the maximum batch size that fits on a single NVIDIA 16GB V100 GPU.[1]

Input format The NSP task requires modification of the input data. We therefore distinguish between two basic input formats: `DOC-SENTENCES` [62], which is used for pre-training of RoBERTa-style and GPT-2-style systems, and `SEGMENT-PAIR+NSP` [62] when training BERT-style systems.

- `DOC-SENTENCES`: For RoBERTa-style and GPT-2-style systems, we follow Liu et al. [62] and use the `DOC-SENTENCES` input format. That is, we string together contiguous sentences until we reach the end of a document (in this case, the resulting input sequence is smaller than the maximum sequence length), or until we reach the maximum sequence length. In the latter case, the remaining

[1] In principle one could, however, perform gradient accumulation over four batches to have the same batch size on both partitions.

part of the document is packed into the next input sequence, beginning at the start of the first successive natural sentence. Each input sequence may contain sentences from only one document. For RoBERTa-style, following Liu et al. [62], each sequence is enclosed by special start and end tokens, `<s>` and `</s>`, respectively. The token vocabulary of size 30,000 is constructed with a byte-level BPE tokenizer, based on the WikiText-103 training set. To make the inputs compatible with the class *RobertaModel* from the Hugging Face transformers library [126], we use a *RobertaTokenizer* from the same library, which we initialize with the byte-level BPE tokens. For GPT-2-style systems, since we implement the same fine-tuning approach as Radford et al. [85], start and end tokens are only added during fine-tuning. Thus, no additional tokens are added to the input sequences during pre-training. To customize the inputs to the class *GPT2Model* [126], we initialize a *GPT2Tokenizer* [126] with a vocabulary of 30,000 byte-level BPE tokens generated from the WikiText-103 training data.

- `SEGMENT-PAIR+NSP`: For BERT-style, following Devlin et al. [25], each input sequence consists of two segments, labeled as `A` and `B`. Segment `A` consists of a randomly chosen number of contiguous natural sentences from one document. In 50%, we fill segment `B` with contiguous natural sentences from a second document, which is chosen uniformly at random. In the remaining 50%, we fill the second segment with the actual natural sentences that follow the sentences from segment `A`. If the document contains more sentences, which were not used due to the maximum sequence length, the same process is reiterated. Furthermore, as Devlin et al. [25], in 10% we use a shorter sequence length, chosen uniformly at random between two and the maximum sequence length, in order to alleviate the mismatch of the input format between pre-training and fine-tuning. Following Devlin et al. [25], all segments are separated with a special `[SEP]` token and the first token of each sequence is a special `[CLS]` token. The token vocabulary of size 30,000 is constructed by applying WordPiece subword tokenization to the WikiText-103 training data. Subsequently, a *BertTokenizer* [126] is initialized with the WordPiece tokens to make the inputs compatible with the *BertModel* [126] class.

Masking During pre-training of BERT-style and RoBERTa-style systems, we mask input sequences as proposed by Devlin et al. [25]. That is, 15% of all tokens are chosen for prediction, and these tokens are replaced with a `[MASK]` symbol in 80% of the time. In the remaining cases, the tokens are either replaced with a random token, or the initial token is kept, with equal probability. In contrast to Devlin et al. [25], following Liu et al.

[62] we *dynamically* mask each input sequence during training, as explained in Chaper 5.

Amount of input data The average and total lengths of the final input sequences for each pre-training task are summarized in Table 7.1. As can be observed, a large part of the data is duplicated when preparing the inputs for the NSP task. Note that, during construction of the NSP inputs, if the second segment of a sequence is replaced with a random sample, the original content of the document is retained and used in the next sequence. For the first segment of this sequence, as before, a random number of contiguous sentences is selected, but this time from the retained content. Then, with a probability of 50%, the second segment is again a random sample. This process is reiterated until the complete document is used up. A randomly sampled segment appears at least twice, one time as a sample and another time in its original location. This explains the large amount of duplicated data. Adding both partitions, the pre-training data of RoBERTa-style systems has an approximate size of 64% of the pre-training data of BERT-style systems. The datasets for GPT-2-style and RoBERTa-style systems have approximately the same lengths. Note that, in all cases, when preparing the inputs each document is filled with contiguous *natural* sentences until either the full document is used up, or the combined length exceeds the maximum sequence length. In the latter case, the exceeding tokens are cut off and the next sequence starts with the first successive natural sentence from the same document. Cutting off the remaining part of a natural sentence when reaching the maximum sequence length results in a slight reduction of the dataset.

7.3 Pre-Training Details

Training duration To ensure a fair comparison of the different pre-training objectives, we pre-train RoBERTa-style and GPT-2-style systems for 10 epochs, and BERT-style systems for 6 epochs, which in all cases equates to approximately $137,000$ total training steps combined over both partitions.[2] Since the data is duplicated when training with MLM & NSP, it is natural to simply lower the number of epochs in relation to the amount of pre-training data. While the amount of pre-training data of RoBERTa-style and GPT-2-style systems amounts to more than 60% of the data of BERT-style systems, we found that, on the other hand, the average WordPiece token contains slightly more information than the average byte-level BPE token.

[2] In sections where we do not compare the different objectives the number of epochs may differ.

Optimization Apart from the experiments in section 7.5.4, we use a batch size of 64 when training on the short sequences and a batch size of 16 for the long sequences. We optimize all systems with Adam [49] using the following parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon =$ 1e-6 and L_2 weight decay of 0.01. For BERT-style and RoBERTa-style systems we use a maximum learning rate of 1e-4, and for GPT2-style systems the maximum learning rate is 2.5e-4. In all cases we use a linear warmup for the first 1000 steps, which corresponds to approximately 1% of the total steps. Furthermore, for all systems we employ dropout with a rate of 0.1 on all layers. The activation function of all systems is the GELU [40]. The hyperparameters are in general chosen as in the original systems, except for RoBERTa-style systems, because RoBERTa was trained with significantly larger batches, which requires different hyperparameters. For RoBERTa-style systems we therefore choose the same hyperparameters as for BERT-style systems.

Implementation We pre-train all systems on a single NVIDIA 16GB V100 GPU, making use of the Hugging Face transformers library [126]. The same also holds true for fine-tuning.

7.4 Fine-Tuning Details

We follow Devlin et al. [25] and train for three epochs on all GLUE tasks. We use a batch size of 16 and a learning rate of 2e-5 for each task. Apart from these hyperparameter configurations, we apply the same fine-tuning procedures that were used by the original systems, as described in Chapter 5. For GPT-2-style systems, we implemented the fine-tuning approach of GPT (because GPT-2 was not fine-tuned).

However, we do make one small modification to the original implementations. In contrast to BERT-style systems, the pre-training objective of RoBERTa-style and GPT-2-style systems does not contain a classification task. When performing the NSP task, in the original BERT the contextualized representation of the `CLS` token is obtained by feeding the corresponding final hidden state through a linear layer with dropout and *tanh* activation. Subsequently, the contextualized representation is fed through another linear layer with dropout, which is the output layer mapping the contextualized representation to the class probabilites. Consequently, when fine-tuning BERT-style systems on a classification task, there are in fact two linear layers between the final hidden state and the output classes. However, RoBERTa and GPT in their original implementation use only one linear layer. In order to be as consistent as possible, in

contrast, we use two linear output layers for all systems. The first linear layer is followed by a *tanh* activation and both layers are implemented with a dropout rate of 0.1.[3]

7.5 Results

In the following we discuss our results. In section 7.5.1, we start by evaluating how varying single shape dimensions affects the performance on different GLUE tasks for the three different pre-training objectives. In this stage we want to investigate whether the performance gain diminishes after a certain level, compare how the performance changes when scaling different dimensions, and examine whether models with different pre-training objectives respond differently to single-dimension scaling. Subsequently, in section 7.5.2 we change multiple shape dimensions simultaneously to investigate whether the different dimensions depend on each other. In sections 7.5.3 and 7.5.4 we study how to train efficiently by varying the model size, the number of training steps and the batch size. In section 7.5.5 we put together our observations from the previous sections and scale networks to different sizes.

We will in general evaluate different systems mainly on MNLI, QQP and QNLI, which are the three largest GLUE tasks, since the results on these tasks are the most reliable, as will be shown. In particular, we therefore calculate the average score over the validation set performances of the three tasks, which we denote by **GLUE-Large**. For MNLI, we consider only the matched validation set when calculating this score.

7.5.1 Scaling Single Shape Dimensions

In this section, we separately scale the number of layers and the embedding dimension, while holding all other dimensions constant. We start by analyzing the performance on the three largest GLUE tasks, which are all sentence-pair tasks. As shown in Figure 7.1, BERT-style systems perform significantly better than GPT-2-style and RoBERTa-style systems on the large sentence-pair tasks, contrary to the results of Liu et al. [62] and in line with the original findings of Devlin et al. [25].

Observation 1. *The pre-training objective has a large impact on the performance of a fine-tuned system. Pre-training with the combination of NSP & MLM achieves the best results on sentence-pair tasks, while training with the unidirectional LM objective shows in general the worst performance.*

[3] For more information regarding this issue see `https://discuss.huggingface.co/t/what-is-the-purpose-of-the-additional-dense-layer-in-classification-heads/526`.

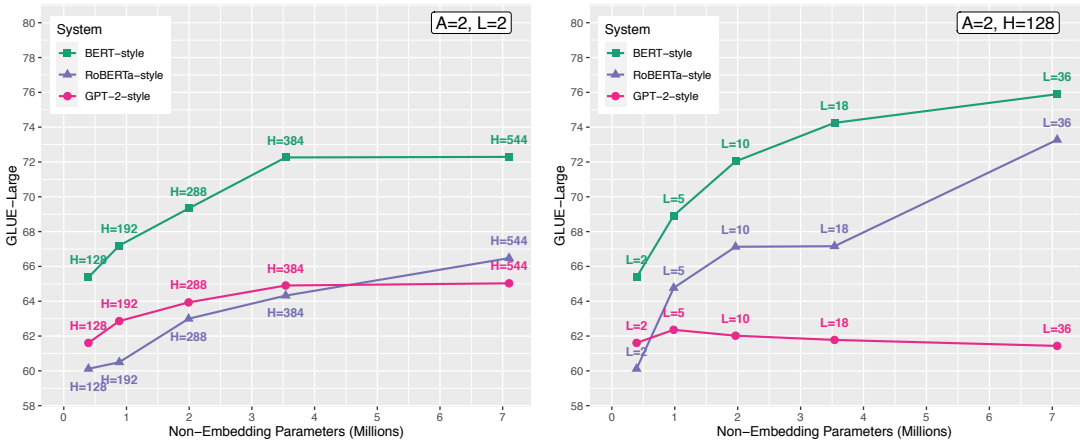

Figure 7.1: Average score on the three largest GLUE tasks, when varying the embedding size (left plot) vs. when varying the number of layers (right plot).

Furthermore, for BERT-style systems the average performance is a relatively smooth function of the model size. Scaling up either the number of layers or the embedding size results in an initial increase in performance, which then saturates at approximately 75% and 72%, respectively. For RoBERTa-style systems, the difference between scaling the number of layers and increasing the embedding size is much larger. Furthermore, although the performance seems to saturate at a certain level, the trend is less apparent than for BERT-style systems.[4] For GPT-2-style systems, the average score slightly increases when scaling the embedding size, but interestingly, stacking more layers shows no positive effect at all. This suggests that GPT-2-style systems require more pre-training data compared to BERT-style and RoBERTa-style systems.

Observation 2. *In most cases, the performance of a fine-tuned system increases up to a certain level when scaling either width or depth, but the progression depends strongly on the pre-training objective.*

Next, we consider SST-2, which is a single-sentence task and the fourth largest task from the GLUE benchmark. In general, in contrast to the results on the sentence-pair tasks, the performance of the three different model types on SST-2 is much closer together. This is an indication that the NSP pre-training task does especially help to learn sentence-pair relationships. However, we observe that shape and model size also seem to have a rather low impact on the SST-2 validation accuracy, suggesting that this task generally shows a stable performance across a wide range of systems.

[4]Note that the relatively low average score for the 18-layer RoBERTa system, shown in the right plot of Figure 7.1, is due to a weak performance on the QNLI task (see Table 7.3).

BERT-Style				**Validation Set Performance**					
A	H	L	N_{model}	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
2	128	2	393,216	65.4	59.0/60.2	72.3	64.8	78.0	0.0
2	192	2	884,736	67.2	62.1/62.8	74.0	65.4	82.6	0.0
2	288	2	1,990,656	69.3	63.7/65.2	76.0	68.3	82.0	0.0
2	384	2	3,538,944	72.3	65.7/66.6	77.8	73.2	81.1	0.0
2	544	2	7,102,464	72.3	66.8/68.1	78.0	72.0	83.3	5.9
GPT-2-Style				**Validation Set Performance**					
A	H	L	N_{model}	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
2	128	2	393,216	61.6	56.3/56.2	66.1	62.3	79.8	0.0
2	192	2	884,736	62.9	58.0/58.4	68.7	61.9	79.7	0.0
2	288	2	1,990,656	63.9	58.7/58.7	70.9	62.2	81.7	0.0
2	384	2	3,538,944	64.9	59.8/59.6	71.9	63.0	81.2	0.0
2	544	2	7,102,464	65.0	59.8/59.7	72.4	62.9	82.5	6.9
RoBERTa-Style				**Validation Set Performance**					
A	H	L	N_{model}	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
2	128	2	393,216	60.1	53.7/55.1	64.7	61.9	79.2	0.0
2	192	2	884,736	60.5	54.4/55.4	65.0	62.0	80.8	0.0
2	288	2	1,990,656	63.0	57.5/58.0	68.1	63.4	80.3	0.0
2	384	2	3,538,944	64.3	59.4/59.8	69.0	64.6	81.9	7.1
2	544	2	7,102,464	66.5	60.2/60.7	72.7	66.5	81.8	17.0

Table 7.2: Performance on GLUE when increasing only the embedding dimension.

CoLA is also a single-sentence task and the fifth largest of the GLUE tasks. Interestingly, for all systems we observe that the validation performance on CoLA is only increasing when scaling the embedding size. This result suggests that a certain embedding size is necessary to learn linguistic skills. Furthermore, RoBERTa-style systems perform better than BERT-style and GPT-2-style systems on CoLA.

The results for the remaining four GLUE tasks are listed in Table B.2 and Table B.1 in the Appendix. The performance on these tasks is very unstable and in many cases decreases for larger model sizes. In most cases we find that, while the training loss decreases, the validation loss is not decreasing during fine-tuning. This is especially true for the three smallest tasks, which all contain less than 4k training examples. We conclude that, most likely, the amount of pre-training data is not large enough to prevent overfitting on these small datasets. We therefore do not further analyze the results on the four smallest GLUE tasks in the following sections. However, the corresponding results on these tasks can be found in the Appendix.

BERT-Style				**Validation Set Performance**					
A	H	L	N_{model}	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
2	128	2	393,216	65.4	59.0/60.2	72.3	64.8	78.0	0.0
2	128	5	983,040	68.9	62.1/64.2	75.0	68.6	79.8	0.0
2	128	10	1,966,080	72.0	65.3/66.9	76.7	74.1	81.8	0.0
2	128	18	3,538,944	74.2	67.2/68.6	77.8	77.7	82.2	0.0
2	128	36	7,077,888	75.9	69.7/70.4	79.7	78.3	83.3	0.0
GPT-2-Style				**Validation Set Performance**					
A	H	L	N_{model}	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
2	128	2	393,216	61.6	56.3/56.2	66.1	62.3	79.8	0.0
2	128	5	983,040	62.4	57.6/56.1	67.4	62.0	80.5	0.0
2	128	10	1,966,080	62.0	56.9/57.0	67.7	61.5	81.4	0.0
2	128	18	3,538,944	61.8	56.1/56.4	66.8	62.4	80.6	0.0
2	128	36	7,077,888	61.4	56.6/56.7	66.6	61.1	80.7	0.0
RoBERTa-Style				**Validation Set Performance**					
A	H	L	N_{model}	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
2	128	2	393,216	60.1	53.7/55.1	64.7	61.9	79.2	0.0
2	128	5	983,040	64.8	59.5/60.6	70.4	64.4	80.2	0.0
2	128	10	1,966,080	67.1	60.9/61.9	72.0	68.5	81.7	0.0
2	128	18	3,538,944	67.2	62.9/64.3	74.3	64.3	80.0	0.0
2	128	36	7,077,888	73.3	67.6/69.1	77.3	75.0	82.6	0.0

Table 7.3: Performance on GLUE when increasing only the number of layers.

7.5.2 Scaling Multiple Shape Dimensions

We next examine whether the performance on GLUE can be improved by scaling multiple dimensions at the same time. First, we increase both H and L and compare the performances with the results from the previous section. Figure 7.2 shows that for RoBERTa-style and BERT-style systems, scaling both dimensions improves the performance on the three largest GLUE tasks significantly.

Observation 3. *Scaling multiple shape dimensions can lead to a better performance than scaling single dimensions, while keeping the model size constant.*

Therefore, we conclude that the shape dimensions are not independent of each other. For GPT-2-style systems, however, we do not observe a performance increase, as shown in Table 7.4.

So far, we did not increase the number of attention heads when scaling the embedding dimension. We observed that, without using more attention heads, wide systems perform worse than deep systems. To evaluate whether a larger number of attention heads can boost the performance of wide systems, we re-implement our widest systems

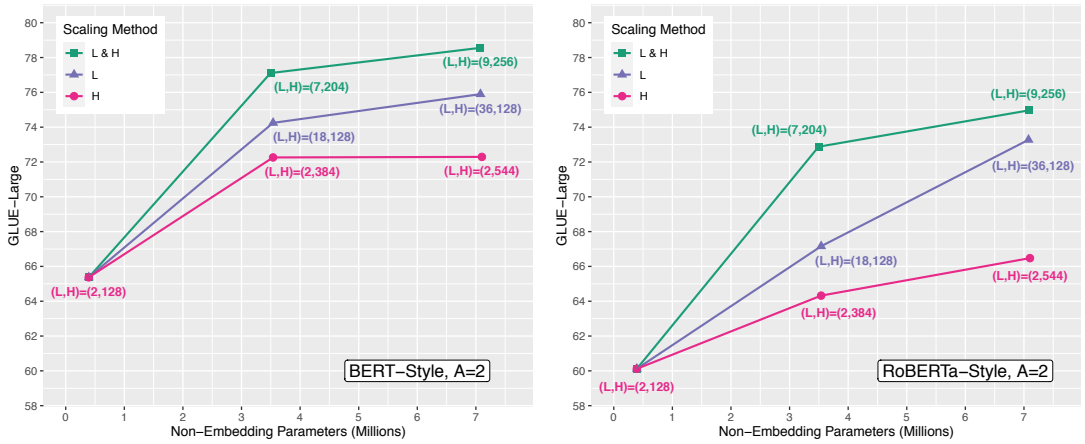

Figure 7.2: Performance on GLUE when increasing single vs. multiple shape dimensions.

with $A = 8$ attention heads, which corresponds to an attention head dimension of 68. The results are listed in Table 7.4. It can be observed that the scores on the large GLUE tasks are improved substantially by using more attention heads. In particular, when using 8 instead of 2 attention heads the wide BERT-style system (A=8, H=544, L=2; Table 7.4) performs better than the deep BERT-style system of comparable size (A=2, H=128, L=36; Table 7.3). Furthermore, as shown in Table in Table 7.4, the wide BERT-style system also performs close to the more balanced network ($A = 2$, $H = 256$, $L = 9$).

Observation 4. *The performance of a fine-tuned system can be similar over a wide range of shapes. For BERT-style systems we observe that wide systems perform slightly better than deep systems, provided that the number of attention heads is adapted to the embedding dimension.*

In contrast to BERT-style systems, deep RoBERTa-style systems still perform better than wide systems with an increased number of attentions heads. For GPT-2-style systems, adding more attention heads hardly increases the performance.

7.5.3 Monitoring the Validation Loss

In the previous sections, different models were made comparable by their number of non-embedding parameters. We have also shown in section that this number is related to the computational cost when evaluated as the number of FLOPs. Reporting the computational cost in FLOPs neglects, however, that some operations can be run in

BERT-Style				**Validation Set Performance**					
A	H	L	N_{model}	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
2	204	7	3,495,744	77.1	69.7/70.6	80.4	81.2	83.5	0.0
2	256	9	7,0778,88	78.6	72.0/72.7	81.2	82.5	83.4	5.2
8	544	2	7,102,464	78.4	71.7/71.9	81.9	81.6	83.3	3.7
GPT-2-Style				**Validation Set Performance**					
A	H	L	N_{model}	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
2	204	7	3,495,744	63.6	58.5/58.9	69.9	62.3	81.1	2.2
2	256	9	7,0778,88	63.8	59.1/59.1	69.7	62.6	81.8	5.3
8	544	2	7,102,464	66.0	60.7/60.4	73.8	63.3	82.6	7.9
RoBERTa-Style				**Validation Set Performance**					
A	H	L	N_{model}	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
2	204	7	3,495,744	72.9	66.0/68.5	76.5	76.2	72.9	10.6
2	256	9	7,0778,88	75.0	68.4/70.9	78.2	78.3	75.0	19.5
8	544	2	7,102,464	70.9	66.1/67.2	77.4	69.1	83.4	14.2

Table 7.4: Performance on GLUE when increasing multiple shape dimensions.

parallel, while others cannot. In order to assess the speed of convergence, following Li et al. [59], we therefore directly report the wall-clock time in seconds.

Figure 7.3 shows the WikiText-103 validation loss for BERT-Style systems of different shape, when trained on the short sequences.[5] The left plot depicts several pre-training loss curves corresponding to the single-dimension scaling experiments from section 7.5.1. Interestingly, when comparing the validation loss with the GLUE results listed in Table 7.2 and Table 7.3, we find that, although increasing the embedding size (while holding fixed the number of attention heads) results in a lower validation loss than increasing the number of layers, the GLUE performance increases more in the latter case. This leads us to the following observation:

Observation 5. *The pre-training validation loss is not necessarily a good indicator for the performance of a fine-tuned system.*

Dependent on the downstream task some architectures presumably favor fine-tuning more than others, which can offset a relatively worse initialization point. This finding suggests that, although Kaplan et al. [47] observe similar test losses for different shapes, benchmarking the corresponding fine-tuned versions may present a different picture.

In the left plot of Figure 7.3 we furthermore observe that shape has a significant effect on the pre-training time. In particular, stacking many layers requires much longer

[5]The validation loss of the subsequent training on the long sequences can be found in Appendix B.

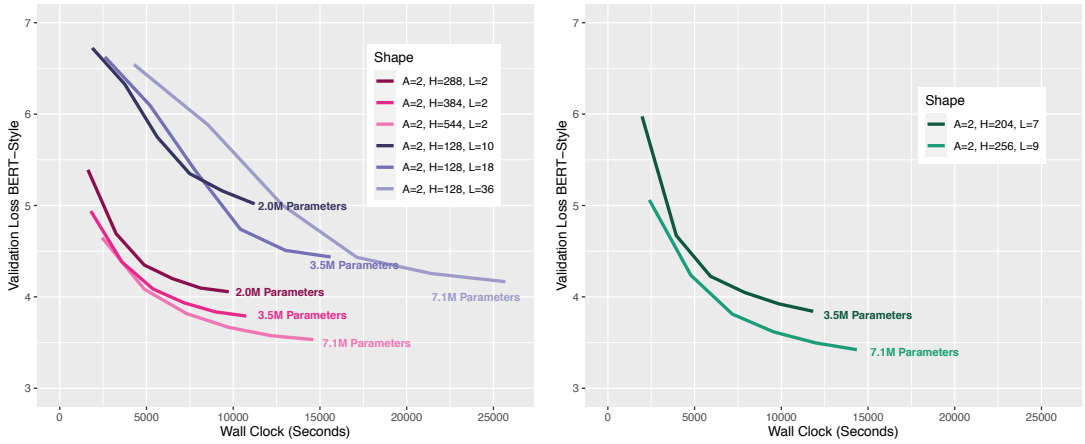

Figure 7.3: Validation loss of BERT-Style systems of different shape. All loss curves are associated with the first stage of pre-training, where we train on short sequences with a maximum length of 128 tokens. The depicted parameter counts refer to the model size N_{model}.

pre-training. It is also evident that increasing the size does not lead to a proportionate increase in the pre-training time. This holds true especially when scaling multiple dimensions, as depicted in the right plot of Figure 7.3. When doubling the number of pre-training parameters, the training time only increases from approximately $11,800$ seconds to approximately $14,400$ seconds. In particular, the loss of the larger system is smaller at any measured point in time.

Observation 6. *Given a fixed time budget, training large systems for a relatively small number of steps is more efficient than training small systems for a large number of steps.*

For instance, the 9-layer system in the right plot of Figure 7.3 achieves a significantly lower validation loss than the 7-layer system after $10,000$ seconds, which corresponds to approximately $65,800$ and $79,800$ steps, respectively. Li et al. [59] made a similar observation by showing that larger Transformer-based systems generally reach a lower pre-training validation perplexity in shorter time. A point of concern might be that larger systems overfit more easily during fine-tuning. However, Li et al. [59] showed that, when stopping models of different size at the same pre-training validation perplexity, large systems achieve in general comparable downstream task performances to small systems, which contradicts the argument of overfitting.

BERT-Style		**Validation Set Performance**					
Training Strategy	Total Time	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
Baseline	21,358s	78.6	72.0/72.7	81.2	82.5	83.4	5.2
$\frac{1}{2}$x steps, 1x batch	10,736s	77.4	70.2/71.2	80.5	81.5	82.5	0.0
1x steps, $\frac{1}{2}$x batch	14,575s	78.2	71.5/71.9	80.9	82.3	83.9	2.5
RoBERTa-Style		**Validation Set Performance**					
Training Strategy	Total Time	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
Baseline	19,760s	75.0	68.4/70.9	78.2	78.3	75.0	19.5
$\frac{1}{2}$x steps, 1x batch	9,906s	73.7	67.0/69.0	76.7	77.4	83.5	−0.03
1x steps, $\frac{1}{2}$x batch	13,101s	75.6	68.2/70.0	79.5	78.9	84.4	11.6

Table 7.5: GLUE results and total pre-training time when varying the batch size vs. the number of training steps.

7.5.4 Number of Training Steps and Batch Size

The amount of training data that is processed by a model, potentially for a repeated of number times, can be increased by either training for more steps or by training with a larger batch size. To evaluate the effect of both variants, we compare how the performance of a baseline system changes when training for 50% of steps vs. when training with a reduced batch size of 50%. As the baseline we use our best performing system thus far ($A = 2, H = 256, L = 9$). We conduct our experiments for RoBERTa-style and BERT-style systems.

The results are listed in Table 7.5. In both cases we find that reducing the number of training steps is more detrimental to the performance than reducing the batch size. Conversely, it follows that when scaling up a system, a better model performance can be achieved when doubling the amount of training steps than when doubling the batch size, which is consistent with the results of Raffel et al. [88]. On the other hand, we observe that the systems with the smaller batch size were trained for a significantly longer time than the systems with the reduced number of training steps. Therefore, increasing the batch size may result in a more favorable training duration than increasing the number of training steps. This is expected, because in the latter case the number of gradient updates is larger. In the above experiment, note that when reducing the number of training steps by 50% the performance on the large GLUE tasks only decreases by about 1-2%. This is consistent with our findings from the previous section and provides additional evidence that training for a large number of steps is inefficient.

Observation 7. *Doubling the number of training steps marginally increases the downstream task performance, whereas doubling the batch size significantly reduces the average*

BERT-Style				Validation Loss (WikiText-103)	Validation Performance (GLUE)
A	H	L	N_{model}	BERT-Style Loss	GLUE-Large
2	128	2	393,216	**5.66**	66.6
2	104	3	389,376	**6.34**	**68.2**
2	90	4	388,800	**6.41**	67.1
2	74	6	394,272	6.47	-
2	64	8	393,216	6.50	-
2	58	10	403,680	6.54	-
2	52	12	389,376	6.58	-
2	48	14	387,072	6.62	-
2	46	16	406,272	6.62	-

Table 7.6: Grid search over nine small BERT-style systems.

training time of an input sequence.

As stated, empirically other studies have also shown that using a larger batch size is much more efficient than training for more steps [47]. This means that the reduction of training time by using larger batches dominates the marginal performance gains resulting from an increased number of training steps.

7.5.5 Systematic Scaling

Based on our observations we now attempt to perform model scaling in a more systematic way. In particular, we apply a modified version of the compound scaling method [110]. This method was used to scale up EfficientNet [110], which achieved a significantly better accuracy on ImageNet [97] than previous approaches, while using less computational resources. Due to Observation 4 we do not expect that compound scaling can provide such large improvements in the case of Transformer-based systems. However, at the same time we observed that the different shape dimensions are not independent of each other, which was the main motivation of Tan and Le [110] for using compound scaling. For scaling we only consider BERT-style systems, because these systems showed the strongest performance in the previous experiments.

We propose the following compound scaling method for Transformer-based systems:

$$\begin{aligned} & L = \alpha^{\phi} \\ & H = \beta^{\phi} \\ & A \approx H/64 \\ & \text{s.t. } \alpha\beta^2 \approx 2 \\ & \alpha \geq 1, \beta \geq 1. \end{aligned} \tag{7.5.1}$$

BERT-Style						Validation Set Performance	
A	H	L	N_{model}	Total Time	Epochs	GLUE-Large	Final Loss
2	256	9	7,0778,88	21,358s	6	78.56	3.24
7	469	4	10,558,128	**20,873s**	5	**79.41**	**3.13**

Table 7.7: Verification of the scaling method: the proposed modifications lead to a better GLUE score and a lower validation loss, while requiring less training time.

For suitable values of α and β, a system is scaled up by increasing the *compound coefficient* ϕ. Doubling the number of layers L doubles the model size N_{model}, but doubling the embedding dimension H will increase the model size by four times. Since the model size also dominates the amount of compute in a Transformer (see section 7.1), the constraint $\alpha\beta^2 \approx 2$ thus ensures that when scaling the network from ϕ_{old} to ϕ_{new}, the amount of compute approximately increases by the factor $2^{\phi_{\text{new}}-\phi_{\text{old}}}$. The amount of compute is approximately the same for any number A of attention heads [117]. Following existing approaches and based on Observation 4, we will therefore set the number of attention heads to $A \approx H/64$.

Grid search To determine α and β, we follow Tan and Le [110] and perform a grid search over a set of nine small networks of comparable size. We train these networks only on the short sequences. Subsequently, we select the three systems with the lowest validation loss. Because of Observation 5, we then fine-tune these three systems on GLUE and evaluate the systems based on the average score on the three largest GLUE tasks. Table 7.6 shows that the best performing system has $L = 3$ number of layers and an embedding dimension of $H = 104$. From the constraint in Eq. (7.5.1) it follows that the size of this system corresponds to a compound coefficient of $\phi = \log_2(LH^2) = 14.99 \approx 15$, such that we obtain $\alpha = 3^{\frac{1}{15}} \approx 1.076$, $\beta = 104^{\frac{1}{15}} \approx 1.363$. Note that the resulting coefficients favor scaling width over depth. In general, we believe that this is reasonable, especially in light of the much longer training times of deep networks compared to wide networks (see Figure 7.3). However, we also want to emphasize that further research is needed, whether these scaling coefficients are suitable for BERT-style systems. For GPT-2-style systems, Kaplan et al. [47] proposed to scale such that width/depth remains fixed. Importantly, however, Kaplan et al. [47] did not study the effect of shape parameters on the GLUE performance, but instead only monitored the LM test loss. In MT, on the other hand, Transformer-based systems are scaled preferably by increasing the width [59, 103]. There also exist approaches that focus on increasing depth, while making modifications to the Transformer to allow for more

BERT-Style				Validation Set Performance					
A	H	L	N_{model}	GLUE-Large	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA
9	585	5	20,553,500	80.7	75.3/75.5	83.5	83.4	85.1	16.5
13	832	5	41,553,440	**81.4**	**75.6/75.9**	**84.1**	**84.4**	**85.8**	**21.3**

Table 7.8: GLUE results of BERT-style systems, scaled up based on the observations made in the previous sections.

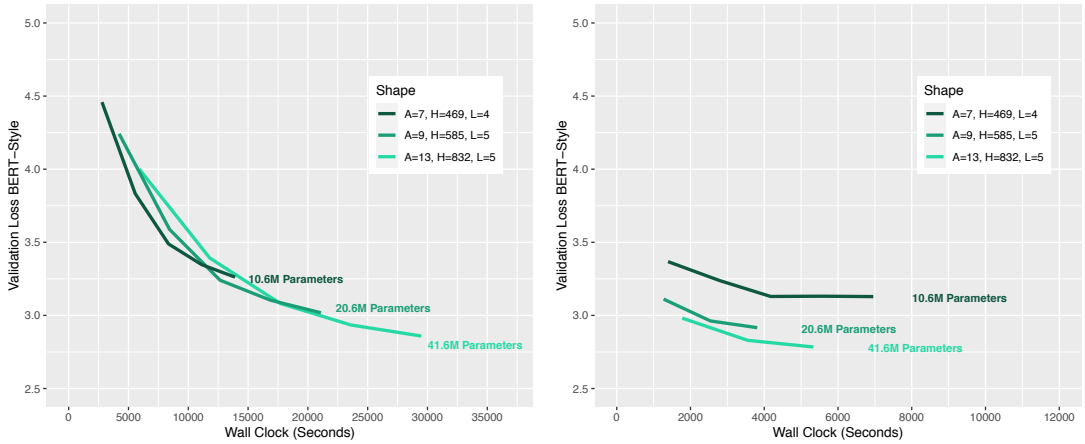

Figure 7.4: Validation loss of large scaled-up BERT-style systems when pre-training on the short sequences (left plot) and for subsequent pre-training on the long sequences (right plot). The depicted parameter counts refer to the model size N_{model}.

efficient training [2].

Scaling Based on Observation 6, we successively increase the compound coefficient to scale three systems to larger sizes than all previously trained systems, but train for less steps. In general, we believe that increasing the model size instead of training for more steps is much more important than the exact shape resulting from the compound scaling method. For our smallest system, we train for 5 epochs on both the long and the short sequences. However, because we observe that the validation loss on the long sequences is not decreasing after the third epoch, for the two larger systems we then further reduce the number of epochs on the long sequences to 3 epochs. The GLUE results of the corresponding systems are listed in Table 7.8. Furthermore, Table 7.7 shows a comparison of the smallest of the three systems to the best performing system so far. As can be observed, both the performance on the large GLUE tasks and the final validation loss are improved, while requiring less training time. For the two larger systems, which are each obtained by approximately doubling the model size, GLUE

performance and validation loss are further improved, as shown in Table 7.8. Note that these systems are rather large compared to the relatively small amount of pre-training data. This demonstrates that neural NLP systems are remarkably robust to overfitting on the pre-training data, which is in line with the results of Kaplan et al. [47].

Chapter 8

Discussion

8.1 Limitations

There are several limitations to our benchmarking study, all resulting from the rather limited amount of computational power used to run our experiments.

In our view, the most severe limitation is the small pre-training dataset. Based on the observations of Kaplan et al. [47], we would expect systems to train faster if more training examples were used. Furthermore, the small size of the pre-training dataset is presumably the main cause of overfitting on smaller tasks. Therefore, for further experiments, we suggest to expand the amount of pre-training data.

Furthermore, we did not tune any pre-training hyperparameters, but instead adopted the configurations from the original systems of our down-scaled versions. However, especially since we used different batch sizes as the original systems, it would be advisable to adjust the hyperparameters accordingly [59].

Ideally, fine-tuning hyperparameters should also be adapted, which could potentially alleviate overfitting. For instance, one could perform early stopping with regard to the validation loss, or modify the learning rates separately for each task. Furthermore, more reliable results may be achieved on small datasets by fine-tuning multiple times with different seeds and selecting the average or the best performance of the runs.

8.2 Directions for Further Research

Kaplan et al. [47] studied the effect of the amount of pre-training data, however, not with regard to downstream task performance. Due to the fact that current NLP systems are trained on vastly different amounts of pre-training data, we believe that this relationship

should be explored further.

Although recent attempts have been made to study the relationship between different unsupervised pre-training objectives and the performance on downstream tasks [3], this relation is yet not well understood. Empirically, contrastive pre-training objectives, such as the replaced token detection method from ELECTRA, have shown very promising results. It would be interesting to extend the study to such contrastive objectives. Since we have observed that the NSP task can be beneficial for learning sentence-pair relationships, it would be furthermore interesting to evaluate how the NOP task used by ALBERT compares.

Finally, by fine-tuning on a larger variety of tasks we could break down in more detail how different modeling choices affect the performances on different tasks. We have, for instance, observed that the embedding size is related to the linguistic skills of a system. We believe that further investigation of such relationships will open many opportunities for future research.

8.3 Conclusion

In our experiments, BERT-style systems consistently outperform RoBERTa-style and GPT-2-style systems. We therefore conclude that, at least in case of a relatively small pre-training dataset, the combination of MLM & NSP is preferable to MLM or LM.

Although our experiments were conducted on a much smaller scale than other studies, we were able to reproduce many previous findings. For instance, we observed that, provided multiple dimensions are scaled, systems with very different shapes can achieve similar performances.

Consistent with the results of Kaplan et al. [47] and Li et al. [59], we found that it is in general inefficient to train until convergence. While training for more steps can improve the performance, this increase is often rather marginal. Instead, in accordance with Kaplan et al. [47], we believe that increasing the batch size is more beneficial than training for more steps.

More importantly, also consistent with the results of Kaplan et al. [47] and Li et al. [59], we conclude that the model size is the key factor in Transformer-based systems. We observed that even for rather large systems, both the final pre-training validation loss and the GLUE performance benefit from further increasing the size. At the same time, the total pre-training time increases at a rather low rate. In particular, given a fixed time budget, large systems reach a lower loss than small systems. Therefore, we believe that additional compute should be allocated mainly to increase the model size.

Appendix A

Part I: Appendix

A.1 Subword Tokenization Algorithms

A.1.1 Byte Pair Encoding

Byte pair encoding (BPE) was initially introduced as a data compression algorithm [29] and established by Sennrich et al. [100] in the context of language modeling.

The version proposed by Sennrich et al. [100] consists of an initial *pre-tokenization* step, which involves segmenting the text into a set of distinct words by spaces. Each word is represented as a tuple of single character symbols. For instance, the word `hello` is represented as (`h`,`e`,`l`,`l`,`o`). At this stage the so-called *base vocabulary* is initialized with all single character symbols contained in the text. In the first iteration, it is then evaluated which pair of adjacent symbols occurs most often in the text; the symbols of the most frequent pair are merged and replaced with this new sequence. For instance, if the most frequent pair is (`e`, `l`), these two symbols are replaced with `el`. In the second iteration, pairs of the adjacent updated symbols are again ordered by frequency and the most frequent symbols are replaced by the respective merged sequence. For example, if the most frequent pair is (`el`, `l`), the two symbols are replaced with `ell`. This greedy approach is repeated until the total number of iterations reaches a limit, which is a hyperparameter specified by the user.

Starting from the base vocabulary, at the end of each iteration the most frequent merged symbol is added to the vocabulary. The resulting vocabulary $\mathcal{V}$ thus corresponds to the union of the base vocabulary and all merged symbols. In practice, instead of initializing the base vocabulary with all symbols contained in the text, a predefined set of available symbols is used, since the number of distinct symbols can be very large (there

is a repertoire of around 140,000 Unicode characters). For instance, GPT [85] employs a base vocabulary of 478 characters and performs 40,000 merge operations, which consequently yields a vocabulary of 40,478 tokens. Since there are many Unicode symbols which are not in the base vocabulary (and therefore also not in the final vocabulary), these symbols and all combinations thereof cannot be encoded. For this reason there is a special `[UNK]` token contained in the vocabulary; any single symbol that is unknown is encoded with this token.

Note that when a new text is encoded with the generated subword vocabulary $\mathcal{V}$, there often exist different possibilities to break a word into subwords. The BPE algorithm therefore stores the order in which the vocabulary was generated. A new text is then encoded by applying these steps in the exact same order.

Furthermore, note that there exist many different possibilities to segment words prior to performing the merge operations. For instance, it seems more plausible to split (`d`, `o`, `n`, `'`, `t`) into (`d`, `o`) and (`n`, `'`, `t`). Such segmentation criteria are in practice implemented by using a rule-based pre-tokenization method. Moreover, in case of a language without spaces between words, word segmentation at the start of the algorithm is infeasible. However, in principle BPE can also be applied without segmenting the text into words in the beginning. In particular, WordPiece [99], which we discuss next, is typically regarded as a variant of BPE and was initially developed with the specific purpose of segmenting Asian texts without spaces into subwords.

A.1.2 WordPiece

WordPiece [99] is a greedy approach that is similar to the original BPE algorithm, with the difference that instead of merging the most frequent pairs, symbols are merged based on the likelihood within a probabilistic model. For instance, assume that at the start of an iteration the two symbols (`e`, `l`) have not been merged yet. The probabilistic model then assigns a likelihood to this current state. Subsequently, the WordPiece algorithm computes the increase in the likelihood for all possible merges and merges the two symbols that result in the strongest increase. Thus, in contrast to the original BPE algorithm, at each step a "lookahead" is used, taking into account what consequences a merge has. Schuster and Nakajima [99] state that WordPiece is based on a LM likelihood, estimated with 3 to 5-grams with Katz back-off [48].

BERT [25] uses a variant of WordPiece to encode its inputs. In this variant, all generated subwords start with two hash signs, except for the first subword in a word. For instance, a possible tokenization of the word `embedding` could be (`em`, `##bed`,

`##ding`). In contrast, (`bed`) could be a possible tokenization of the word `bed` (i.e., in this case without hash signs). This approach adds an additional granularity to the subword vocabulary, as it allows for differentiating between different meanings of a subword depdendent on its position in a word. Furthermore, BERT introduces two special tokens, `[CLS]` and `[SEP]`, which are discussed in Chapter 5.

A.1.3 Byte-level BPE

Although the original BPE data compression algorithm [29] operates on bytes (hence the name *byte pair encoding*), both BPE in the context of language modeling as well as WordPiece use a base vocabulary comprised of Unicode characters. As mentioned, there exist around $140,000$ Unicode characters. In order to keep the size of the base vocabulary small, BPE and WordPiece use a language specific set of the most important characters, e.g., 400 English characters in case of GPT. However, as described, this approach induces `[UNK]` tokens when performing a task on new text data.

Instead of using Unicode characters, Radford et al. [86] introduced a new version of BPE that operates directly on bytes, called *Byte-level BPE*, to encode the inputs of GPT-2. The base vocabulary of Byte-level BPE consequently consists of all $2^8 = 256$ bytes. Apart from using bytes instead of Unicode characters, Byte-level BPE is identical to the BPE algorithm proposed by Sennrich et al. [100]. Because any Unicode symbol can be represented using (at most four) UTF-8-byte units, there are no more `[UNK]` tokens. Furthermore, due to its universal base vocabulary, byte-level BPE is language agnostic.

Appendix B

Part II: Appendix

B.1 Floating Point Operations

A FLOP is assumed to be either a multiplication or a summation.

Scalar-vector multiplication For a vector $\mathbf{b} \in \mathbb{R}^H$, the scalar-vector multiplication $\alpha\mathbf{b}$ requires H FLOPs.

Dot product The dot product $\boldsymbol{a}^T\boldsymbol{b}$ of two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^H$ consists of $2H - 1 \approx 2H$ FLOPs, since it requires H multiplications and $H - 1$ summations.

Weighted sum The weighted sum $\sum_{i=1}^{N} \alpha_i \mathbf{b}_i$ of N vectors $\boldsymbol{b}_i \in \mathbb{R}^H$ consists of N scalar-vector multiplications and $(N-1)H$ summations. Hence the computational cost amounts to $2HN - H \approx 2HN$ FLOPs.

Matrix-vector product The computation of a matrix-vector product $\mathbf{Ab}$, $\mathbf{A} \in \mathbb{R}^{H_1 \times H_2}$, $\mathbf{b} \in \mathbb{R}^{H_2}$, consists of H_1 inner products and therefore requires $2H_1H_2 - H_1 \approx 2H_1H_2$ FLOPs.

B.2 Results

B.2.1 Scaling Single Shape Dimensions

BERT-Style				**Validation Set Performance**			
A	H	L	N_{model}	STS-B	MRPC	RTE	WNLI
2	128	2	393,216	35.6	82.1	50.9	59.2
2	192	2	884,736	44.4	81.3	57.0	54.9
2	288	2	1,990,656	59.7	81.3	54.5	40.8
2	384	2	3,538,944	65.5	81.6	59.2	45.1
2	544	2	7,102,464	64.2	81.9	54.5	38.0
GPT-2-Style				**Validation Set Performance**			
A	H	L	N_{model}	STS-B	MRPC	RTE	WNLI
2	128	2	393,216	3.4	82.1	57.0	46.5
2	192	2	884,736	10.4	80.1	49.1	43.7
2	288	2	1,990,656	17.4	78.7	49.8	39.4
2	384	2	3,538,944	19.8	79.7	49.5	45.1
2	544	2	7,102,464	24.9	77.1	53.1	26.8
RoBERTa-Style				**Validation Set Performance**			
A	H	L	N_{model}	STS-B	MRPC	RTE	WNLI
2	128	2	393,216	-6.0	81.2	57.4	54.9
2	192	2	884,736	12.2	81.7	52.7	45.1
2	288	2	1,990,656	13.2	81.9	53.4	42.3
2	384	2	3,538,944	14.8	80.0	51.3	38.0
2	544	2	7,102,464	17.2	81.0	53.1	39.4

Table B.1: Performance on GLUE (small tasks) when increasing only the embedding dimension.

BERT-Style				**Validation Set Performance**			
A	H	L	N_{model}	STS-B	MRPC	RTE	WNLI
2	128	2	393,216	35.6	82.1	50.9	59.2
2	128	5	983,040	41.1	81.4	57.0	52.1
2	128	10	1,966,080	3.1	81.7	60.3	46.5
2	128	18	3,538,944	12.0	80.7	58.5	46.5
2	128	36	7,077,888	60.5	81.2	61.4	56.3
GPT-2-Style				**Validation Set Performance**			
A	H	L	N_{model}	STS-B	MRPC	RTE	WNLI
2	128	2	393,216	3.4	82.1	57.0	46.5
2	128	5	983,040	8.3	80.1	54.2	49.3
2	128	10	1,966,080	7.8	78.8	51.6	52.1
2	128	18	3,538,944	9.1	79.0	49.8	56.3
2	128	36	7,077,888	6.8	79.4	53.1	56.3
RoBERTa-Style				**Validation Set Performance**			
A	H	L	N_{model}	STS-B	MRPC	RTE	WNLI
2	128	2	393,216	-6.0	81.2	57.4	54.9
2	128	5	983,040	4.2	81.4	48.4	54.9
2	128	10	1,966,080	4.4	80.1	59.5	52.1
2	128	18	3,538,944	7.1	81.4	50.2	50.7
2	128	36	7,077,888	9.8	81.1	54.2	50.7

Table B.2: Performance on GLUE (small tasks) when increasing only the number of layers.

B.2.2 Scaling Multiple Shape Dimensions

BERT-Style			Validation Set Performance			
A	H	L	STS-B	MRPC	RTE	WNLI
2	204	7	64.5	80.7	57.4	54.9
2	256	9	77.3	81.4	58.8	42.3
8	544	2	79.0	82.2	61.7	29.6
GPT-2-Style			**Validation Set Performance**			
A	H	L	STS-B	MRPC	RTE	WNLI
2	204	7	14.8	78.4	49.5	40.8
2	256	9	16.6	78.4	50.2	42.3
8	544	2	23.7	77.2	53.8	33.8
RoBERTa-Style			**Validation Set Performance**			
A	H	L	STS-B	MRPC	RTE	WNLI
2	204	7	18.0	80.4	51.6	49.3
2	256	9	23.1	81.0	50.2	49.3
8	544	2	23.5	82.2	52.0	36.6

Table B.3: Performance on GLUE (small tasks) when increasing multiple shape dimensions.

B.2.3 Monitoring the Validation Loss

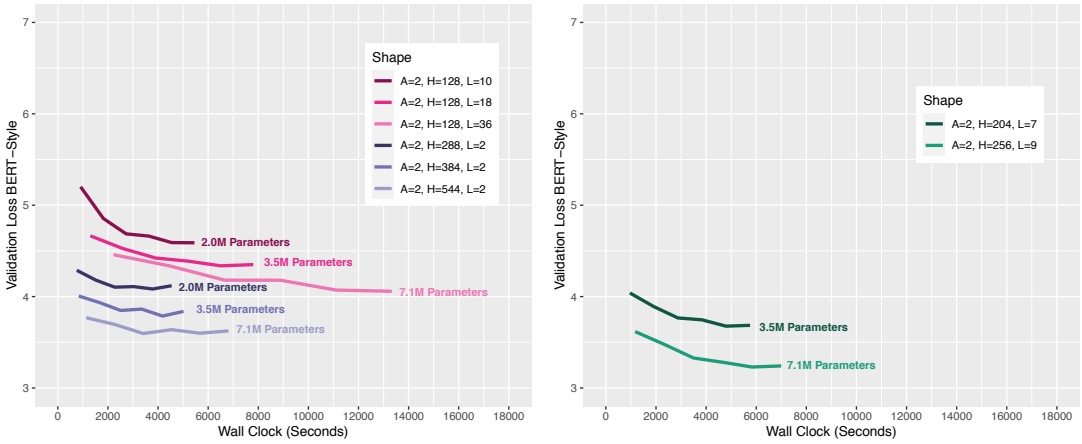

Figure B.1: Validation loss of BERT-Style systems of different shape. All loss curves are associated with the second stage of pre-training, where we train on long sequences with a maximum length of 512 tokens. The depicted parameter counts refer to the model size N_{model}.

B.2.4 Number of Training Steps and Batch Size

BERT-Style		**Validation Set Performance**			
Training Strategy	Total Time	STS-B	MRPC	RTE	WNLI
Baseline	21,358s	77.3	81.4	58.8	42.3
$\frac{1}{2}$x steps, 1x batch	10,736s	74.1	81.7	60.3	45.1
1x steps, $\frac{1}{2}$x batch	14,575s	76.2	83.0	56.3	50.7
RoBERTa-Style		**Validation Set Performance**			
Training Strategy	Total Time	STS-B	MRPC	RTE	WNLI
Baseline	19,760s	23.1	81.0	50.2	49.3
$\frac{1}{2}$x steps, 1x batch	9,906s	15.4	80.0	47.3	42.2
1x steps, $\frac{1}{2}$x batch	13,101s	30.5	80.6	51.6	39.4

Table B.4: GLUE results (small tasks) and total pre-training time when varying the batch size vs. the number of training steps.

B.2.5 Systematic Scaling

BERT			**Validation Set Performance**			
A	H	L	STS-B	MRPC	RTE	WNLI
7	469	4	78.6	81.8	60.3	36.6
9	585	5	80.9	83.4	61.4	36.6
13	832	5	81.8	82.8	57.8	25.4

Table B.5: GLUE results (small tasks) of BERT-style systems, scaled up based on the observations made in the previous sections.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. Character-level language modeling with deeper self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3159–3166, 2019.

[3] Sanjeev Arora, Hrishikesh Khandeparkar, Mikhail Khodak, Orestis Plevrakis, and Nikunj Saunshi. A theoretical analysis of contrastive unsupervised representation learning. *arXiv preprint arXiv:1902.09229*, 2019.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[6] Ankur Bapna, Naveen Arivazhagan, and Orhan Firat. Simple, scalable adaptation for neural machine translation. *arXiv preprint arXiv:1909.08478*, 2019.

[7] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[8] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[9] Luisa Bentivogli, Peter Clark, Ido Dagan, and Danilo Giampiccolo. The fifth pascal recognizing textual entailment challenge. In *TAC*, 2009.

[10] Zsolt Bitvai and Trevor Cohn. Non-linear text regression with a deep convolutional neural network. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 180–185, 2015.

[11] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[12] Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[13] Daniel Cer, Mona Diab, Eneko Agirre, Inigo Lopez-Gazpio, and Lucia Specia. Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation. *arXiv preprint arXiv:1708.00055*, 2017.

[14] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.

[15] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21(1):6, 2020.

[16] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[17] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[18] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[19] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, 2008.

[20] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537, 2011.

[21] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loic Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. *arXiv preprint arXiv:1705.02364*, 2017.

[22] Ido Dagan, Oren Glickman, and Bernardo Magnini. The pascal recognising textual entailment challenge. In *Machine Learning Challenges Workshop*, pages 177–190. Springer, 2005.

[23] Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. In *Advances in neural information processing systems*, pages 3079–3087, 2015.

[24] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[26] William B Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005.

[27] Miguel Domingo, Mercedes Garcıa-Martınez, Alexandre Helle, Francisco Casacuberta, and Manuel Herranz. How much does tokenization affect neural machine translation? *arXiv preprint arXiv:1812.08621*, 2018.

[28] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[29] Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2): 23–38, 1994.

[30] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*, 2017.

[31] Danilo Giampiccolo, Bernardo Magnini, Ido Dagan, and Bill Dolan. The third pascal recognizing textual entailment challenge. In *Proceedings of the ACL-PASCAL workshop on textual entailment and paraphrasing*, pages 1–9, 2007.

[32] Aaron Gokaslan and Vanya Cohen. Openwebtext corpus, 2019.

[33] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[34] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[35] Roger Grosse and Jimmy Ba. Lecture notes on attention, February 2019. URL `http://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec16.pdf`. Last visited 2020-12-12.

[36] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 297–304, 2010.

[37] R Bar Haim, Ido Dagan, Bill Dolan, Lisa Ferro, Danilo Giampiccolo, Bernardo Magnini, and Idan Szpektor. The second pascal recognising textual entailment challenge. In *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*, 2006.

[38] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of computing*, 8(1): 321–350, 2012.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[40] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. 2016.

[41] Felix Hill, Kyunghyun Cho, and Anna Korhonen. Learning distributed representations of sentences from unlabelled data. *arXiv preprint arXiv:1602.03483*, 2016.

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

[43] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. *arXiv preprint arXiv:1902.00751*, 2019.

[44] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[45] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[46] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.

[47] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[48] Slava Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing*, 35(3):400–401, 1987.

[49] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[50] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[51] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings*

of the 45th annual meeting of the ACL on interactive poster and demonstration sessions, pages 177–180. Association for Computational Linguistics, 2007.

[52] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959*, 2018.

[53] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[54] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. Race: Large-scale reading comprehension dataset from examinations. *arXiv preprint arXiv:1704.04683*, 2017.

[55] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

[56] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[57] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[58] Hector Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*. Citeseer, 2012.

[59] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers. *arXiv preprint arXiv:2002.11794*, 2020.

[60] Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. *arXiv preprint arXiv:1801.10198*, 2018.

[61] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding. *arXiv preprint arXiv:1901.11504*, 2019.

[62] 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. *arXiv preprint arXiv:1907.11692*, 2019.

[63] Lajanugen Logeswaran and Honglak Lee. An efficient framework for learning sentence representations. *arXiv preprint arXiv:1803.02893*, 2018.

[64] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[65] Dehong Ma, Sujian Li, Xiaodong Zhang, and Houfeng Wang. Interactive attention networks for aspect-level sentiment classification. *arXiv preprint arXiv:1709.00893*, 2017.

[66] Brian W Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[67] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. In *Advances in Neural Information Processing Systems*, pages 6294–6305, 2017.

[68] Oren Melamud, Jacob Goldberger, and Ido Dagan. context2vec: Learning generic context embedding with bidirectional lstm. In *Proceedings of the 20th SIGNLL conference on computational natural language learning*, pages 51–61, 2016.

[69] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[70] Tomas Mikolov, Martin KarafiÃ¡t, Lukas Burget, Jan CernockÃ$\frac{1}{2}$, and Sanjeev Khudanpur. Recurrent neural network based language model. volume 2, pages 1045–1048, 01 2010.

[71] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[72] Sewon Min, Minjoon Seo, and Hannaneh Hajishirzi. Question answering through transfer learning from large fine-grained supervision data. *arXiv preprint arXiv:1702.02171*, 2017.

[73] Andriy Mnih and Geoffrey Hinton. Three new graphical models for statistical language modelling. In *Proceedings of the 24th international conference on Machine learning*, pages 641–648, 2007.

[74] Andriy Mnih and Koray Kavukcuoglu. Learning word embeddings efficiently with noise-contrastive estimation. In *Advances in neural information processing systems*, pages 2265–2273, 2013.

[75] Lili Mou, Zhao Meng, Rui Yan, Ge Li, Yan Xu, Lu Zhang, and Zhi Jin. How transferable are neural networks in nlp applications? *arXiv preprint arXiv:1603.06111*, 2016.

[76] James Myers. Acceptability judgments. In *Oxford Research Encyclopedia of Linguistics*. 2017.

[77] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[78] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[79] Ankur P Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*, 2016.

[80] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[81] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[82] David MW Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*, 2020.

[83] Ofir Press and Lior Wolf. Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859*, 2016.

[84] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained models for natural language processing: A survey. *arXiv preprint arXiv:2003.08271*, 2020.

[85] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.

[86] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1 (8):9, 2019.

[87] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*, 2019.

[88] 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. *arXiv preprint arXiv:1910.10683*, 2019.

[89] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[90] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *arXiv preprint arXiv:1806.03822*, 2018.

[91] Prajit Ramachandran, Peter J Liu, and Quoc V Le. Unsupervised pretraining for sequence to sequence learning. *arXiv preprint arXiv:1611.02683*, 2016.

[92] Jonathan S Rosenfeld, Amir Rosenfeld, Yonatan Belinkov, and Nir Shavit. A constructive prediction of the generalization error across scales. *arXiv preprint arXiv:1909.12673*, 2019.

[93] Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

[94] Sebastian Ruder. *Neural transfer learning for natural language processing*. PhD thesis, NUI Galway, 2019.

[95] Sebastian Ruder, Matthew E Peters, Swabha Swayamdipta, and Thomas Wolf. Transfer learning in natural language processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, pages 15–18, 2019.

[96] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[97] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[98] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[99] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152. IEEE, 2012.

[100] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[101] Aliaksei Severyn and Alessandro Moschitti. Unitn: Training deep convolutional neural network for twitter sentiment classification. In *Proceedings of the 9th international workshop on semantic evaluation (SemEval 2015)*, pages 464–469, 2015.

[102] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[103] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.

[104] Sheng Shen, Zhewei Yao, Amir Gholami, Michael Mahoney, and Kurt Keutzer. Powernorm: Rethinking batch normalization in transformers. In *International Conference on Machine Learning*, pages 8741–8751. PMLR, 2020.

[105] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[106] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[107] Alessandro Sordoni, Philip Bachman, Adam Trischler, and Yoshua Bengio. Iterative alternating neural attention for machine reading. *arXiv preprint arXiv:1606.02245*, 2016.

[108] Sandeep Subramanian, Adam Trischler, Yoshua Bengio, and Christopher J Pal. Learning general purpose distributed sentence representations via large scale multi-task learning. *arXiv preprint arXiv:1804.00079*, 2018.

[109] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[110] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[111] Wilson L Taylor. cloze procedure: A new tool for measuring readability. *Journalism quarterly*, 30(4):415–433, 1953.

[112] Stefanie Tellex, Boris Katz, Jimmy Lin, Aaron Fernandes, and Gregory Marton. Quantitative evaluation of passage retrieval algorithms for question answering. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 41–47, 2003.

[113] Sebastian Thrun and Lorien Pratt. Learning to learn: Introduction and overview. In *Learning to learn*, pages 3–17. Springer, 1998.

[114] Trieu H Trinh and Quoc V Le. A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847*, 2018.

[115] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394, 2010.

[116] Joaquin Vanschoren. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548*, 2018.

[117] 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*, pages 5998–6008, 2017.

[118] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[119] Alex Wang and Kyunghyun Cho. Bert has a mouth, and it must speak: Bert as a markov random field language model. *arXiv preprint arXiv:1902.04094*, 2019.

[120] 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. *arXiv preprint arXiv:1804.07461*, 2018.

[121] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 2019.

[122] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[123] Wikipedia contributors. Directed acyclic graph — Wikipedia, the free encyclopedia, 2020. URL `https://en.wikipedia.org/wiki/Directed_acyclic_graph#Combinatorial_enumeration`. [Online; accessed 7-December-2020].

[124] Wikipedia contributors. Nebraska highway 88 — Wikipedia, the free encyclopedia, 2020. URL `https://en.wikipedia.org/w/index.php?title=`

Nebraska_Highway_88&oldid=951097426. [Online; accessed 3-October-2020].

[125] Adina Williams, Nikita Nangia, and Samuel R Bowman. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426*, 2017.

[126] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, RÃ©mi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[127] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[128] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057, 2015.

[129] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pages 5753–5763, 2019.

[130] Zichao Yang, Xiaodong He, Jianfeng Gao, Li Deng, and Alex Smola. Stacked attention networks for image question answering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 21–29, 2016.

[131] Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. Swag: A large-scale adversarial dataset for grounded commonsense inference. *arXiv preprint arXiv:1808.05326*, 2018.

[132] Kelly W Zhang and Samuel R Bowman. Language modeling teaches you more syntax than translation does: Lessons learned through auxiliary task analysis. *arXiv preprint arXiv:1809.10040*, 2018.

[133] Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and Bill Dolan. Dialogpt: Large-scale generative pre-training for conversational response generation. *arXiv preprint arXiv:1911.00536*, 2019.

[134] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.