Bachelor Thesis

An Empirical Analysis of Self-built GPT Models for GLUE Task Performance

Simon Ian Richter

February 23, 2024

Reviewer:
Prof. Dr. Stefan Harmeling
M.Sc. Marc Höftmann

Abstract

Since the release of OpenAI's ChatGPT at the end of 2022, the "Generative Pretrained Transformer" (GPT) has gained widespread attention. GPTs are generative language models that are applied to a wide range of tasks spanning in domain of natural language processing (NLP). They are typically pretrained on large datasets and then finetuned to more specific tasks. This work focuses on analyzing the characteristics of this type of artificial neural network, in particular with respect to the performance on the GLUE benchmark.

After presenting the architecture of GPT networks, we use a self-implemented framwork for pretraining GPTs on the dataset OpenWebText as well as finetuning and evaluation them on the GLUE benchmark. This GLUE benchmark is a collection of nine typical tasks in the field of natural language processing. It allows for an quantitative comparison between different models by using various metrics to assign scores between 0 and 100 for each task.

We train and evaluate both a small and a large GPT model on all tasks of the benchmark. Analyzing the trend of the reached scores in dependence on the amount of pretraining concludes, that the use of pretraining and finetuning leads to a significant increase in scores compared to just training the models on the tasks. We notice that this gain quickly saturates with further pretraining. During finetuning, we observe instabilities on some of the tasks that cause fluctuations in the scores and can even cause divergence in some cases.

We also compare our two models to a variety of other models. While the comparison to our self-established baselines provides reference scores to improve upon, the GLUE baselines show performance that was common before GPTs were introduced. OpenAI's well-known GPT1 model improves significantly on these baselines, and performs only slightly worse than our large model. The human performance on the GLUE benchmark ultimately shows the limitations of our models.

Contents

1 Introduction

In this thesis the process of implementing, training, finetuning and evaluating a Generative Pretrained Transformer (GPT) is illustrated. GPTs are often used in the domain of natural language processing (NLP), aiming to learn a general understanding of natural language while acquiring implicit knowledge that goes well beyond grammar rules in order to solve all kinds of language related tasks.

1.1 Motivation and Background

TURING [1950] proposes the question ”can machines think”. As part of this, he introduces his now famous ”imitation game”, a game where two contestants are only allowed to answer via written text and have to persuade an interrogator that they are human. Wondering ”what will happen when a machine takes the part of [one contestant] in this game?”, Turing places high value in a machine's capability to imitate human behaviour in the form of written text. This shows the value he places on the ability of ”thinking machines” to process and generate natural language.
Natural language processing is a large subdomain in the field of machine learning. Generally speaking, it focuses on various tasks involving sequences of characters, usually in the form of written text. Typical NLP-tasks include language translation, text classification, text summarization or text generation. There exist various tests and benchmarks like the ”One Billion Word Benchmark” [Chelba et al., 2013] or the ”Language Modeling Broadened to Account for Discourse Aspects” (LAMBADA) dataset [Paperno et al., 2016], used to measure the performance of NLP models on such tasks.
While many established types of language models, for example recurrent neural networks (RNNs) or long short-term memory (LSTM) networks [Hochreiter and Schmidhuber, 1997], reach good results on benchmarks, they are not convincing enough to solve the imitation game. 66 years after Turing proposed his imitation game, the introduction of the transformer architecture and especially the adaption of it in form of the Generative Pretrained Transformer resembled an inn ovation in the field of natural language processing. Even though the imitation game is rather a thought experiment than an objective test, it can be argued that GPT-based systems come closer to succeeding in it than any previous architecture[1]. They also perform very well on many established benchmarks, often surpassing the state of the art. Beyond that, with only a short fine-tuning to adapt to each task, a single GPT model can be able to outperform an ensemble of models each developed exclusively for a specific task. GPT models not only show state-of-the-art performance, but also versatility and high adaptability.
Companies from all sectors are exploring how GPTs can be incorporated into their products or services. The impressive performance paired with a GPT's capability of generating text make it suitable for all kinds of tasks that involve communication with humans, like for example customer services or personal assistants. In the last years Microsoft invested over $10 billion in OpenAI [Times, 2023] and integrated OpenAI'S systems into many of their products, like Edge, Github or Azure. This sudden growth of the AI sector even lead the european commission to propose regulations on the danger of artificial intelligence, known as the AI act [Council of European Union, 2023].
This thesis aims to give a better understanding of GPT models by explaining the theoretical principles and providing a vivid example in the form of a self-implemented and self-trained GPT model. This includes all training steps as well as all methods used for evaluation. The process of developing a versatile and well-performing language model is presented and its behaviour is analyzed and compared to other models. The trained model is by no means just for demonstration purposes, as it achieves good evaluation scores on the GLUE Benchmark.

[1] Brown et al. [2020] find that human evaluators have difficulty distinguishing articles written by humans from articles written by their largest GPT-3 model.

1.2 What is a Generative Pretrained Transformer?

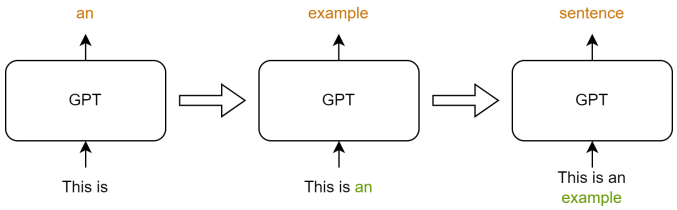


Figure 1.1: An example of how a GPT can generate texts of any length. After a prediction is made, input and output are concatenated and used to make another prediction.

A Generative Pretrained Transformer (GPT) is a type of neural network and falls into the category of language models. It relies heavily on the use of so-called attention layers, namely causal multi-head self-attention. This mechanism allows a GPT to process given texts up to a certain maximum length in a single forward-pass, without the use of recurrence. The attention mechanism puts the individual components of a sentence in relation to each other and helps to distinguish important information from irrelevant elements. Through the use of a special causal masking in the attention modules, GPTs can be efficiently trained to continue a given input, similar to auto-completion used in mobile phones.
Such an input may vary in length and does not need to be padded. In general, it should not exceed the maximum length that was used in the training process. Some model variants have a structural upper limit while others are theoretically capable of extrapolating to infinite lengths. Like most language models, GPTs do not work on normal text data but require a preprocessing technique called tokenization. Tokenization converts a string into a sequence of class labels called token. How classes are decided and what a single token may represent varies widely. A natural approach would to break a sentence down into individual characters and treat each one of them as a token. These token are treated as classes, assigning each of them a unique class label, usually an id. With these labels a text sequence can be represented as a sequence of IDs. For example, when using ASCII to assign IDs to the characters, the word ”token” would be encoded by the sequence 116-111-107-101-110.
Based on a sequence of token given as input, the output of a GPT is a prediction of which token has the highest probability to come next. To extend a sequence by more than one token, the predicted token is appended to the end of the input and fed back into the model. By repeating this process, the output will grow step by step until it reaches the desired length. This generative behaviour distinguishes GPTs from other transformer-based language models. Figure 1.1 shows a visualization of this procedure.
As impressive as the generative capabilities of language models are, many NLP tasks focus only on the ability to understand a given text. Often the output is not expected to be text. Especially when trying to objectively compare different NLP models, a natural language output is usually not desired, as this would itself require another NLP model to analyse the outputs. GPTs are still suited for such tasks, as they have been observed to be good at transferring the pretrained language understandings to new types of tasks, requiring only a short finetuning [Radford et al., 2019]. With only minimal architectural changes and a small amount of additional training, an already trained GPT model can be efficiently adapted to new tasks.

1.3 Thesis Structure

This thesis is divided into 6 chapters.

1. The first chapter introduces natural language processing, typical challenges and the role generative pretrained transformers have in it. A short summary provides an overview of what a GPT actually is. The motivation for this work is given and the structure of the thesis is presented.

2. Chapter two gives an overview over transformers and how decoder-only transformers like generative pretrained transformers are derived from them. The complete architecture of a GPT is layed out and described. In addition, a brief differentiation is given between decoder-only and encoder-only networks.

3. The implementation of a generative pretrained transformer is discussed in chapter three. The used libraries and resources are mentioned and the key components of the framework like the self-attention mechanism or the training loop are described in detail. Also a few decisions that reduce both model size and training time are pointed out.

4. The fourth chapter deals with the conducted experiments and their methodology. The employed training process and the evaluation methods are described. It also provides an overview of the different datasets that are used in training. Furthermore, two primitive models are derived, providing a baseline for the benchmark scores.

5. In chapter five, the resulting benchmark scores are presented and discussed. The impact of pretraining on the results and the general behaviour of the tasks is addressed. To put our self-built models into context, we compare them to various models. Ultimately the human performance shows the limitations of our model.

6. The thesis is summarized in chapter six, looking back at the important steps and concluding the achieved scores.

2 Architecture

This chapter introduces the transformer architecture as described by Vaswani et al. [2017]. Based on this, the GPT architecture as a decoder-only variant of a transformer is derived and a brief overview of the encoder-only counterpart is given. After that the GPT architecture and all components are described in depth.

2.1 Overview

2.1.1 Transformer

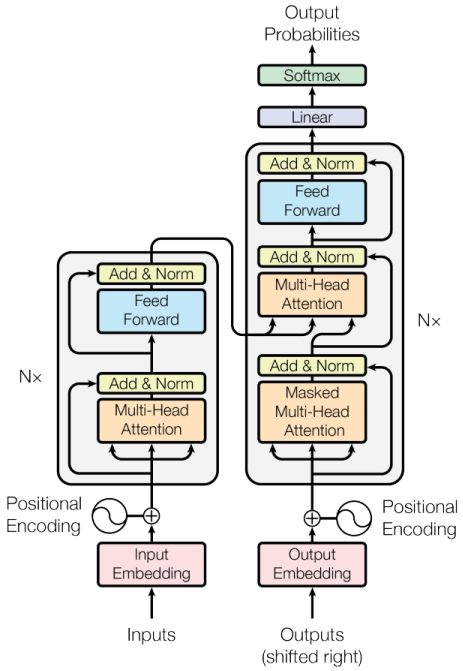


Figure 2.1: The transformer architecture as shown in Vaswani et al. [2017]. The two grey highlighted regions are the encoder block (left) and the decoder block (right). Both of these blocks are repeated N times to form the stacks.

Transformers are deep neural networks used for sequence-to-sequence modelling. They are so-called sequence transduction models, aiming at transforming a given input sequence into another sequence, potentially of different length. The transformer architecture, developed by Vaswani et al. [2017], is intended to solve translation problems where an input text needs to be translated into an output text of another language, while maintaining the semantic meaning. In contrast to the previous dominant architectures for translation tasks, like LSTMs or convolutional neural networks (CNN) [Lecun et al., 1998], a transformer does not make use of either recurrence or convolutions. While CNNs are highly parallelizable, Chen et al. [2021] argue that convolutional operations fail to model long range dependencies due to

their intrinsic locality. LSTMs do not suffer from this problem, however their recurrent structure limits the parallelizability, resulting in slow optimization. The transformer architecture relies on a mechanism called "Attention". This mechanism can model long-range dependencies in a given sequence independent from their distance while allowing for great parallelisability.
The Transformer architecture consists of two stacks of blocks that slightly differ in their architecture. These two stacks are called the encoder and decoder. The purpose of the encoder is to form internal representations of a given input sequence, which are then used in the decoder to generate a new sequence. Both stacks are connected by a mechanism called cross-attention, combines two representations of different sources into one output representation. In the example of a translation task, the encoder would encode the input text into representations that the decoder would use to generate the output text in another language, while preserving the semantic meaning of the input.
The generation process of the decoder stack happens one position at a time. The decoder's final representations are used in an output head to generate a probability distribution over all token, which is then used to sample one token. The previous output text is then extended by the predicted token and again used as an input to the decoder to predict the next token. By repeating this process, a sentence is produced one position at a time in an auto-regressive manner. More formal: with the model's parameters ρ, a maximum sequence length T and the token vocabulary V, the probability of a token $v_t \in V$ at position $t = 0, ..., T$ in the sequence is defined as

$$P(v_t|\rho, v_0, ..., v_{t-1}), \tag{2.1}$$

Even though encoder and decoder blocks have a very similar structure, they behave differently. One key difference is the use of causal masking in the self-attention modules of the decoder blocks. This masking prevents information flow from left to right[1]. Causal masking makes it possible to train for next token prediction on multiple prefixes of a sequence simultaneusly. Figure 4.1 gives an example of this training approach. Without causal masking such a training approach is not possible.

2.1.2 Decoder-based Architectures

A generative pretrained transformer is based on the described transformer architecture, however it is not aimed at sequence transduction tasks, but rather at sequence continuation tasks. It's aim is not to transform a sequence into a different one, but to extend a given sequence by further positions, which is similar to what the decoder part of a transformer is used for.
For such text generation, the encoder stack of a transformer does not serve a purpose, since there is no input text from another domain or language. As the encoder stack is not needed, it is removed. Additionally the cross-attention module that connects encoder and decoder is omitted as well. This leaves only the input embeddings, the decoder stack and the final output head, as depicted in Figure 2.2. GPTs are therefore often referred to as decoder-only transformers as they only use the decoder side of the transformer.
A very important component is the masking used in the decoder's attention modules. By preventing leftward information flow, decoder-based models can be efficiently trained for sequence continuation on varying sequence lengths simultanously, giving them their generative capabilities.

[1] No information from upcoming token may be used, considering the common reading direction in western languages.

2.1.3 Encoder-based Architectures

There also exist encoder-only transformers, a famous example being the family of "bidirectional encoder representations from transformers" (BERT) models proposed by Devlin et al. [2018]. Similar to decoder-only transformers, they are based on the transformer architecture, but remove one of the stacks. Encoder-based architectures keep the encoder part and omit the decoder stack. From a structural point of view, encoder-only and decoder-only architectures are very similar and differ mainly in the masking of the self-attention, which is present in decoder-only architectures, but not in encoder-only architectures. The key difference between the two architectures is that in encoder-only models the output at any positon can be based on the information of any position of the input, whereas a position in the output of a decoder-only model can only use information from positions earlier in the input sequence. This difference leads to a significantly different behaviour, making different training approaches necessary for encoder-only architectures, which generally do not target text generation[2]. Because encoder-based models are usually not trained to be generative, they are mostly used for other NLP tasks like document classification [Adhikari et al., 2019] or named entity recognition [Labusch et al., 2019].

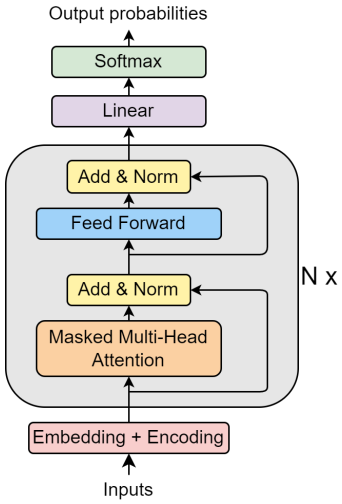


Figure 2.2: A decoder-only transformer. It consists of only the decoder side of a transformer and misses the cross-attention module.

2.2 The GPT Architecture

This section goes into depth regarding preprocessing as well as the architecture of a generative pretrained transformer. It presents all algorithms and components that take part in a GPT network. A description of the model's hyperparameters used throughout the explanations can be found in Table 2.1

Table 2.1: An overview over the hyperparameters controlling the model size

d_{model}	This parameter describes the dimension of internal input representations per token. It can be understood as the number of channels, in which a single token is represented.
n_{layers}	The amount of decoder blocks chained in a row behind each other. n_{layers} controlls the depth of the network.
n_{heads}	The amount of parallel attention heads per layer. Each individual head independently creates representations of size $\frac{d_{model}}{n_{heads}}$
l_{vocab}	The number of different token in the tokenizer's vocabulary. This value represents the number of classes.

[2] Devlin et al. [2018] pretrain their BERT models using two unsupervised tasks. The model is given two sentences as input and has to predict token at randomly masked positions, while also deciding whether the two sentences follow each other in the dataset.

2.2.1 Input Embedding

Tokenization

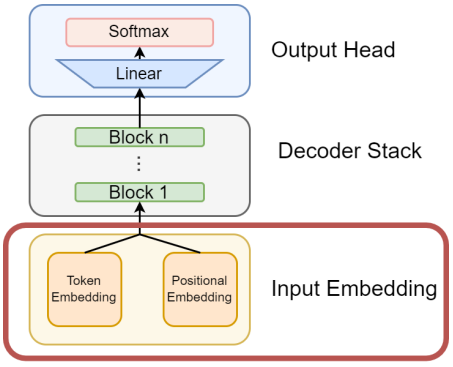


Figure 2.3: The general architecture of a GPT. This section describes the input embedding.

In order to make the input text usable for the GPT model, a preprocessing step is necessary that transforms a text sequence into a sequence of numbers. This is done by tokenization. It converts a given input text into a sequence of token IDs, each of which represent a part of the text. Depending on the tokenization approach, a single token usually stands for either a single character, or a whole word.

Tokenization on the character-level is likely the most intuitive approach, as it is common practice in digital Systems. Well-known encodings like ASCII or UNICODE act as character-level tokenization, since they are used to translate a character into a unique number. Therefore applying such an encoding to a text sequence will yield a sequence of token IDs. Character-level tokenization utilises a relatively small vocabulary, for example 128 different token when using ASCII as a base. This approach brings the benefit, that every text can be translated to a token sequence[3]. In most languages, however, a single symbol does not carry much information by itself, and the semantics of a text lies in the combination of multiple characters. Thus, such a fine-grained representation achieves only a low information density per token and requires many token to encode a given text.

Word-level tokenization adresses this problem by having a token stand for a whole word. This increases the information density but comes at the cost of a much larger vocabulary. To ensure that every text can be tokenized, the vocabulary would have to contain every existing word in the language. Because such a large vocabulary is unrealistic, it's size is usually limited to a certain size at the risk of running into out-of-vocabulary situations in which some text passages cannot be tokenized.

This limitation of tokenization on higher abstraction levels becomes even more obvious when considering the extreme case of sentence-level tokenization. In theory, the information density could be increased even further by representing whole sentences with a single token. However, it is obviously not a realistic approach as it would require an immense vocabulary.

Subword-level tokenization is an approach that combines the best of both character-level and word-level tokenization. The vocabulary contains all individual characters, guaranteeing that every text can be tokenized. In addition, the vocabulary includes parts of words and even whole words to increase the information density where possible. Text that contains only common words can be tokenized in the same way as word-level tokenization. Only in rare cases, some uncommon words that would otherwise cause out-of-vocabulary errors, need to be broken down into multiple subwords. Ideally, such subwords can still be larger parts of words. In the worst case the word has to be treated at the character level, splitting it into individual characters.

For example, the word "falling" may not be in the vocabulary, but the subwords "all" and "ing" might, as they are very common in the English language. Therefore, "falling" would not be represantable as a single token. However, it can be split into the three subwords "f", "all" and "ing". In this example, the word would be tokenized into "f-all-ing". If no matching subwords are part of the vocabulary, the worst case is to represent a word on the character-level. In this example, 3 token are needed to represent the

[3] given that the text itself can be encoded in the base format

word ”falling” which is better than the character-level approach of 7 token, but worse than the theoretical word-level optimum of 1 token. However the word-level approach would not have worked at all in this example because ”falling” was not part of the vocabulary.
Because the qualitiy of subword-tokenization depends on the choice of the vocabulary, usually a considerable amount of statistical calculations is required in advance. There are different approaches to building a suitable vocabulary with Byte Pair Encoding being one of the most popular.

Byte Pair Encoding

The byte-pair encoding (BPE) algorithm was first described in 1994 as a compression algorithm [Gage, 1994]. Sennrich et al. [2015] use the algorithm in a slightly moddified version to build a vocabulary for subword-tokenization.
The Byte pair encoding algorithm starts with a base vocabulary consisting of all characters. This guarantees that every text can be encoded and no out-of-vocabulary exceptions can occur. The vocabulary is then iteratively expanded. To do so a comprehensive reference text is tokenized with the current vocab-

Data: base vocabulary V, reference text R, number of merges M
for $i \leftarrow 0$ **to** M **do**
  $T \leftarrow$ tokenize R with V;
  $p \leftarrow$ most common token pair in T;
  $V \leftarrow V \cup p$;
end

Algorithm 1: Pseudocode of the Byte Pair Encoding algorithm

ulary and then analyzed. The most common pair of two token is then merged into a new token which is added to the vocabulary. For example if the token pair ”i”, ”n” appears most frequently in the text, a new token ”in” will be added to the vocabulary[4]. With the new token added to the vocabulary, the tokenization of the reference text is updated and the frequencies of all pairs are recalculated. Again the pair occuring the most is merged into a new token and added to the vocabulary. With each iteration of this algorithm the vocabulary grows by one token. Because all added token can be part of another pair in following merges, a complex word can be constructed over multiple iterations if it occurs often enough. To improve calculation time and to avoid overfitting on the reference text, token pairs across two words are not considered. The variant of BPE used by Radford et al. [2019] also prevents the algorithm from merging across character categories to avoid multiple token in the vocabulary that differ only in special characters like for example ”dog.” and ”dog?”.
BPE’s hyperparameters are the amount of iterations and the base vocabulary that is used. As the vocabulary grows by one token with each iteration, the final vocabulary size is the sum of the initial vocabulary’s size and the amount of iterations. Due to the greedy nature of the algorithm the most common (sub-)words are added to the vocabulary first and very rare constellations come late in the merging process. With enough iterations byte pair encoding approaches a word-level tokenization as every possible word will eventually be added to the vocabulary[5]. More merges result in tokenization with better compression strength and representability, but come at the cost of a larger vocabulary. A large vocabulary is in most cases undesired as it usually results in larger embedding layers and therefore slower training times of the model.

[4] The order of the two token matters. The pairs ”i”, ”n” and ”n”, ”i” are considered two different pairs.
[5] Given a comprehensive enough reference text

In this work, we use the BPE-based tokenization algorithm used by Radford et al. [2019]. Other popular subword tokenization algorithms are Unigram [Kudo, 2018], WordPiece [Wu et al., 2016] or SentencePiece [Kudo and Richardson, 2018].

Token Embedding

The only purpose of tokenization is to transform a text sequence into a token sequence. A token is treated as an individual class and therefore has an unique ID[6].
The first layer of the GPT model embedds all input IDs into vectors of dimension d_{model}. Usually this is done by using a tensor of shape $(l_{vocab} \times d_{model})$. This tensor is effectively used as a lookup table in which the token ID acts as an index to select a row of the tensor. The lookup table is a learnable parameter and is trained together with the model.

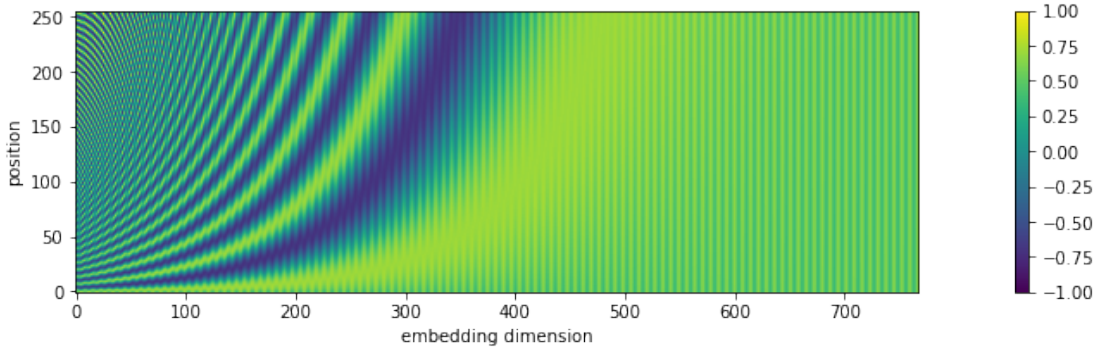
Positional Encoding

In contrast to recurrent neural networks, a GPT processes the entire input sequence at once. This is possible because the attention mechanism is capable of attending to information across multiple token in a single calculation. Even though the attention mechanism utilises information at different input positions, the actual information about a token's relative position in the sequence is not taken into account. This means that the words of a sentence could be swapped arbitrarily without changing the result. However, in natural language the order of words and the overall sentence structure are also of semantic value and can change the meaning of a sentence entirely. It is therefore necessary to add positional information to the representation of each token, so that the attention mechanism can take it into account. That is why a representation of the absolute positions and therefore the order of the token needs to be encoded in the embedding.
A positional encoding layer is used to create another vector embedding for each position in the input sequence. Just like the token representation, the position representation of a single token is of dimension d_{model}. The final embedding of an input token is the result of summing the token and position embedding vectors position-wise.
The positional encoding itself can be differentiated into two categories. A learned positional encoding typically uses a learnable lookup-table like the one used for the token embedding. It works in the same way, but uses the absolute positions as indices instead of the token IDs. Just like the token embedding such a positional embedding is trained along with the model. This approach is flexible as it allows the model to choose the positional representations itself, but it comes with a drawback. The maximum length of an input sequence is limited by the rows of the matrix. Longer sequences cannot be encoded.
A fixed positional encoding guarantees that a position can be encoded through the use of a formula which exactly defines the encoding vector for every possible value. Special formulas with sinusoidal components may allow the model to extrapolate better because they have a repeating pattern. The fixed sinusoidal embedding by Vaswani et al. [2017] is used in this work. A visualization can be found in Figure 2.4a. It is defined as

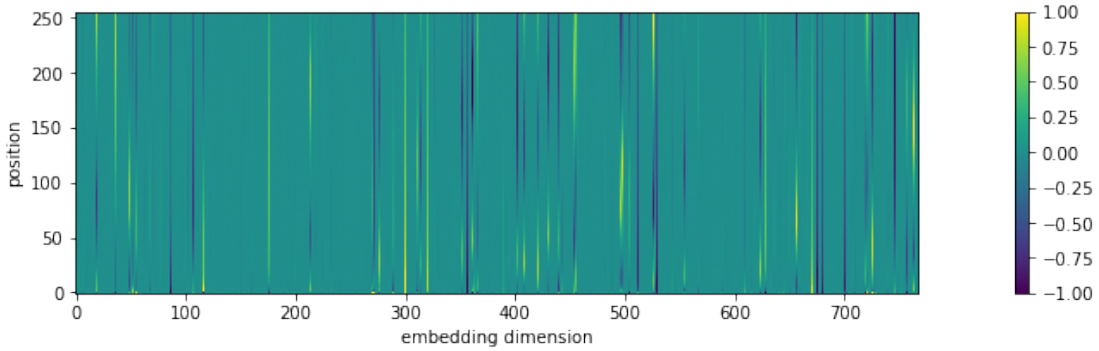
$$PE_{(pos,i)} = \begin{cases} \sin(pos/10000^{i/d_{model}}) & \text{, if i is even,} \\ \cos(pos/10000^{(i-1)/d_{model}}) & \text{, if i is odd,} \end{cases} \tag{2.2}$$

where pos is the position of the token to be encoded and i identifies the row in the resulting embedding vector.

[6] Usually the token are simply indexed, starting from 0 up to the size of the vocabulary minus one.



(a) The sinusoidal positional encoding shown for the first 256 positions. It is a fixed positional encoding used by Vaswani et al. [2017] for the original transformer. It is calculated using Equation 2.2.



(b) A learned positional encoding matrix. This is the weight matrix of the small version of GPT2, trained by Radford et al. [2019]. It is cropped to 256 positions for comparison.

Figure 2.4: A comparison between (a) a fixed positional encoding and (b) a learned positional encoding Both matrices are of same shape. Each row corresponds to the embedding vector of an individual position.

2.2.2 Decoder Stack

A GPT's decoder stack consists of n sequentially repeated decoder blocks. Each block consists of two main sublayers.The block's input, output and all intermediate results between sublayers use token representations of dimension d_{model}. The two main components of a block are a self-attention layer, followed by a feed-forward network. Both sublayers are surrounded by a residual connection as proposed by He et al. [2015b]. LayerNorm [Ba et al., 2016] is aplied to each sublayer's input.

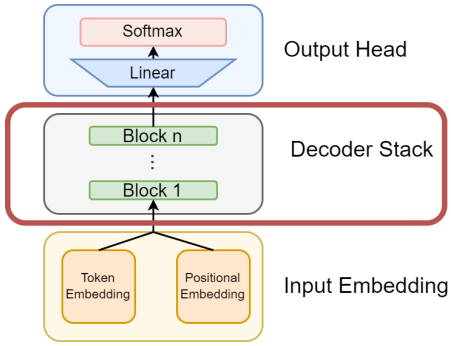


Figure 2.5: The general architecture of a GPT. This section describes the decoder stack.

Masked Multi-Head Self Attention

The self-attention mechanism is a component that allows the network to find relations between token in an input sequence. For such tasks a typical linear layer is not suitable, as it only connects across the last dimension of a matrix. In this context, each token position is represented as a vector, which means that a linear layer can only alter these vector representations token-wise, making connections across multiple token impossible. The attention mechanism overcomes this problem by computing outputs as weighted sums of all positions. In short, every output position is the result of a linear combination of all input positions. The used weights depend on the inputs and are different for each output position. There is no need for any kind of recurrence in the network, because the whole token sequence is processed at once in a single forward pass. This eliminates dependencies between multiple forward passes as they occur in recurrent neural networks and allows for better parallelization while training.
For each position i in the sequence, the first step is to calculate three vectors called q_i, k_i and v_i (query, key, value) of dimensionality d_k[7], which, for single-head attention, is usually the same as d_{model}. This is done by applying three linear projections to a row vector x_i of dimension d_{model}, which represents the ith input token. The weight matrices W_q, W_k and W_v used for the projections have the shape $(d_{model} \times d_k)$. Optionally, bias vectors b_q, b_k and b_v of dimension d_k can be added to the resulting vectors

$$q_i = x_i W_q + b_q \,, \tag{2.3}$$

$$k_i = x_i W_k + b_k \,, \tag{2.4}$$

$$v_i = x_i W_v + b_v \,. \tag{2.5}$$

The next step is to calculate scores $s_{i,j}$ by applying a similarity function between the ith query vector and the jth key vector. The comparison can be realised in different ways. Transformers and GPTs utilize the so-called dot-product attention which relies on the dot-product between query and key vectors as a similarity measure[8]. Because the result of the dot-product might grow large in magnitude for large d_k, it is scaled by a factor of $\frac{1}{\sqrt{d_K}}$ to prevent the problem of vanishing gradients in the following steps [Vaswani et al., 2017].

$$s_{i,j} = \frac{q_i \cdot k_j}{\sqrt{d_k}} \tag{2.6}$$

A softmax function is then applied to form scores $score_{i,j}$, such that $\sum_k score_{i,k} = 1$ for each i. Since the GPT architecture uses the decoder part of a transformer, a masking is applied in the attention mechanism. To understand the masking approach, it is necessary to first understand the meaning of the scores in context. $score_{i,j}$ describes the proportion of how much the value vector v_j should contribute to the output at position i. Effectively, the score values regulate the information flow between input and output positions. As illustrated in Figure 4.1, an output at position i may not use any information from input positions $j > i$, a look-ahead into inputs coming later in the sequence is not possible. Masking enforces this restriction by setting the scores representing illegal "backward" connections, i.e. all $score_{i,j}$ with $i < j$, to zero.
However, directly zeroing out some score values would violate the constraint of $\sum_k score_{i,k} = 1$. Therefore, the masking needs to be applied before the softmax by setting all $s_{i,j}$ with $i < j$ to $-\infty$.

[7] Even though the value vectors could technically be of a different length than d_k, it is common practice to use the same size for all three vectors. For simplicity this convention is used in all following explanations.
[8] Other common types of attention are additive attention[Bahdanau et al., 2014] or local-base attention[Luong et al., 2015].

This will cause the softmax to effectively ignore the masked entries in the calculation, because $e^{-\infty} = 0$.

$$score_{i,j} = \text{softmax}\big(\text{mask}(s_{i,j})\big) = \frac{e^{\text{mask}(s_{i,j})}}{\sum_k e^{\text{mask}(s_{i,k})}} \tag{2.7}$$

In the last step, these scores are used as weights of a linear combination of the value vectors to form the outputs out_i, each of dimension d_k.

$$out_i = \sum_k score_{i,k} \cdot v_k \tag{2.8}$$

Usually these calculations are optimized by performing each step on all i positions simultaneously in form of matrix multiplications. Three matrices Q, K and V contain all q_i, k_i and v_i vectors as rows. Likewise, the inputs x_i are stacked as rows in the input matrix x. They are computed by left multiplying the input matrix x, which contains all x_i vectors as rows, to the weight matrices W_q, W_k and W_v. Q, K and V are therefore of shape $(T \times d_k)$ with T being the number of token in the input.

$$Q = \begin{pmatrix} - & q_1 & - \\ & \vdots & \\ - & q_T & - \end{pmatrix} = xW_q + b_q, \quad K = \begin{pmatrix} - & k_1 & - \\ & \vdots & \\ - & k_T & - \end{pmatrix} = xW_k + b_k, \quad V = \begin{pmatrix} - & v_1 & - \\ & \vdots & \\ - & v_T & - \end{pmatrix} = xW_v + b_v \tag{2.9}$$

The dot product similarities are calculated all at once with a single large scaled matrix multiplication of Q and the transpose of K and collected in a matrix S.

$$S = \begin{pmatrix} s_{1,1} & \dots & s_{1,T} \\ \vdots & \ddots & \vdots \\ s_{T,1} & \dots & s_{T,T} \end{pmatrix} = \frac{QK^\top}{\sqrt{d_k}} \tag{2.10}$$

Masking and row-wise softmax are then applied to S to obtain the scores, also gathered in a matrix $Score$.

$$Score = \begin{pmatrix} score_{1,1} & \dots & score_{1,T} \\ \vdots & \ddots & \vdots \\ score_{T,1} & \dots & score_{T,T} \end{pmatrix} = \text{softmax}\big(\text{mask}(S)\big) \tag{2.11}$$

Right multiplying the value matrix V to the score matrix $Score$ yields the output vectors, again stacked row-wise in a matrix Out.

$$Out = \begin{pmatrix} - & out_1 & - \\ & \vdots & \\ - & out_T & - \end{pmatrix} = Score \times V \tag{2.12}$$

With Q, K and V as defined in Equation 2.9, the masked self-attention mechanism can be written as

$$\text{masked self-attention}(Q, K, V) = \text{softmax}\left(\text{mask}\left(\frac{QK^\top}{\sqrt{d_k}}\right)\right)V\,. \tag{2.13}$$

Instead of performing this self-attention only once per block, it appears to be beneficial to perform multiple independent self-attention calculations in parallel. Vaswani et al. [2017] argue, that the averaging that happens in a single attention head inhibits the model of jointly attending to information from different representation subspaces at different positions. Multiple smaller attentions experience the same averaging, but it is only limited to the respective head[9].
A similar computational cost to that of a single large attention module can be achieved by reducing the size of the attention modules in proportion to their number. If each of the self-attentions is proportionally smaller, the total computational cost is similar to the one of a single but larger self-attention. Assuming a single self-attention with $d_k = d_{model}$, a comparably sized multi-head self-attention with h modules can be realised by choosing $d_k = \frac{d_{model}}{h}$. Because there are h attention modules, h sets of query, key and value matrices Q_n, K_n and V_n with $n = 1, ..., h$ are used. On these sets, the attention function can then be applied in parallel to obtain h output vectors of dimension $d_k = \frac{d_{model}}{h}$. Such a proportionally smaller self-attention module with it's own independent projections for Q_n, K_n and V_n is called an attention head. To form a single output, the outputs of the h attention heads are concatenated position-wise, again forming output representations of dimension d_{model}. An additional position-wise linear layer allows the model to further combine the different parts of an output position's representation.

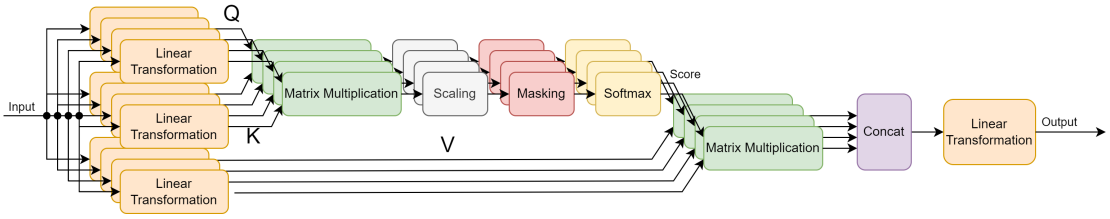


Figure 2.6: The calculations inside a masked multi-head attention module. In this example, the input is processed by four parallel attention heads. The four results are concatenated and passed through a linear layer to form the final output.

Feed-Forward Network

The feed-forward network operates token-wise, which means that every position is treated independently and only the embedding vector is altered. It consists of two linear transformations with an activation function in between. Both input and output dimensions are d_{model}, while the hidden layer is commonly chosen to have $4 \cdot d_{model}$ neurons.
The activation function can vary. The original transformer uses a rectified linear unit (ReLU), but OpenAI's GPT models utilize the gaussian error linear unit (GELU) proposed by Hendrycks and Gimpel [2016]. The output of the feed-forward network is calculated by

$$FFN(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2\,. \tag{2.14}$$

[9]Each attention head calculates a scalar similarity value $s_{i,j}$ for every query-key pair. Even though the token's representations might encode many different features, the similarity function outputs only an average similarity over all features. With multiple smaller heads, each head can focus on a different part of the representations.

Residual Connections

Residual connections [He et al., 2015a] are employed, bypassing each sublayer. This adds an additional path connecting a sublayer's input directly to it's output. Because all intermediate results are of the same embedding dimension, the two paths can simply be added together without any size adaptions[10]. This forms a structure in which both sublayers add their results on top of the block's input instead of replacing it. The residual paths can be seen in Figure 2.7. Residual connections aim to prevent the problem of vanishing gradients that occurs in deep neural networks. The connections provide a direct path for the gradient to reach the previous layers of the network.

LayerNorm

Layer normalization [Ba et al., 2016] is a technique of normalizing the internal representations of neural networks. It can increase the convergence speed and acts as a regularizer during training. Similar to batch normalization [Ioffe and Szegedy, 2015], mean μ and variance σ^2 are calculated based on the inputs to the LayerNorm. These are then used to center and scale the data.
In contrast to batch normalization, LayerNorm does not compute it's statistics feature-wise over all elements in a batch, but rather over all features, independent for each element in the batch. LayerNorm is therefore applied to every token position individually on vectors of dimension d_{model}.
This provides more stability when encountering varying input lengths and does not impose contstraints on the batch size, even allowing for a batch size of 1. Another difference is that LayerNorm behaves the same during training and inference[11].
The mean μ and variance σ^2 for a given vector x with dimension d_x are calculated as

$$\mu = \frac{1}{d_x}\sum_{k=1}^{d_x} x_k\,, \qquad \sigma^2 = \frac{1}{d_x}\sum_{k=1}^{d_x}(x_k-\mu)^2\,. \tag{2.15}$$

Based on μ and σ the output of the Layer normalization is defined as

$$\text{LayerNorm}(x) = \frac{\gamma}{\sigma}(x-\mu)+\beta\,, \tag{2.16}$$

with a gain weight γ and a bias β both of dimension d_x.
There are different approaches to using LayerNorm in transformers. While Vaswani et al. [2017] places the LayerNorm after the subcomponents and outside of the residual blocks in their transformer, most modern models use the Pre-LN transformer architecture by Xiong et al. [2020a], which applies the LayerNorm inside the residual blocks and in front of the subcomponents, as can be seen in Figure 2.7. The authors argue that the Pre-LN transformer can handle higher learning rates than the Post-LN transformer without the training becoming unstable. Also the warmup stage does not have to be designed as carefully, making the training setup simpler.

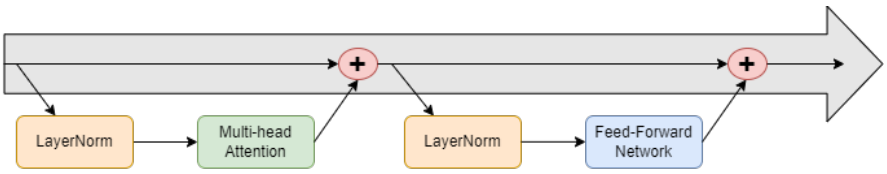


Figure 2.7: The residual paths inside a single decoder block. The attention module and the feed-forward network are each contained in their own residual block. Following the approach of Xiong et al. [2020b], LayerNorm is placed before the sublayers inside the residual blocks.

[10] Otherwise a linear transformation would be necessary to create matching dimensions.

[11] Batch norm uses moving averages during inference to estimate population statistics. During training, the actual batch statistics are used.

2.2.3 Output Head

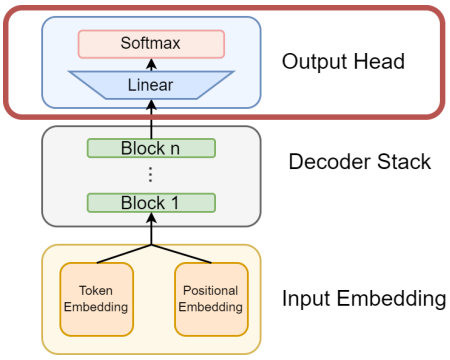


Figure 2.8: The general architecture of a GPT. This section describes the output head.

The output head uses the representations formed in the last block of the decoder stack to form the final output of the GPT. As a GPT is a very flexible architecture that is suitable of solving many different tasks, the output head may vary depending on the task.

For classification tasks the head converts the decoder stack's output representation into a probability distribution over all classes. This is done using a linear layer with an input size of d_{model} and the output size being the number of classes. To translate the values for the classes into a valid probability distribution, a softmax is applied. This ensures that all probabilities are non-negative and sum up to one.

A GPT's probably most common task, text generation, is treated just like a classification task. Every token in the vocabulary corresponds to a class, which means that the weight matrix in the linear layer is of shape $(d_{model} \times d_{vocab})$. It is common practice to reuse the token embedding weights as weights for the output head, as they share dimensions. This weight tying between the layer that embeds the token and the layer that transforms the embeddings back into token saves quite a lot of parameters. It can be interpreted as encouraging the network's decoder stack to produce outputs in the same internal language as the inputs. The shared weight matrix is learnable, which means it can adapt to the training. However a change always has an effect on both layers. A simple approach for sampling from the resulting probability distribution is to always select the class with the highest probability by using an argmax[12]. Because natural language can sometimes be ambiguous and is not always clearly classifiable, another common approach is to treat the probabilities as a multinomial distribution and sample directly from them. This introduces some variance that might actually be closer to human-written text than always choosing the most likely answer.

To solve regression tasks a linear layer is used that has an input size of d_{model} and an output size of one. The value of that single output dimension is then directly used as the prediction for the regression task. Since regression is not necessarily limited to values between 0 and 1, the softmax layer is not used.

Due to the masking used in all attention modules, the output holds only information of all inputs up to the current position. For example the output at position five is based on the input positions one to five, but not six and higher. Preventing leftward information flow means that higher output positions contain information about a longer input prefix. When training for sequence continuation, the output head is applied to all output positions individually, because all output positions perform a different prediction. For classification or actual generation of text, the model may give only a single output. To make the best possible prediction, it should utilize all information in the input. In such cases, only the last position of the stack's output is used, as it contains the most information. All other output positions are ignored.

[12] In this case, the final softmax can be omitted as it does not change the result of an argmax.

3 Implementation

This chapter deals with the code of our framework, used to conduct our experiments. The theoretical structure of a GPT is applied to an actual implementation. Furthermore, the used training loops are explained as well as the preparation of the datasets. The most important aspects are presented in depth while the complete code can be found in the Appendix or on Github[1]. All code is written in the programming language Python and the library PyTorch is used to implement the model, the training loop with dataloading and most of the optimizations.

3.1 The GPT Model

The GPT model itself is implemented as a class that inherits from pytorch's nn.module class. This allows PyTorch to treat the class as a neural network, enabling all of the built-in features that allow forward and backward passes and make the use of PyTorch's pre-built optimizers possible. Because "GPT" is a subclass of nn.module, it has to implement the method "forward" that handles the forward pass of an input through the network. With the forward pass defined, PyTorch manages the backward pass completely on it's own through a feature called autograd. The full code of the GPT model itself can be found in the file GPT.py.
This section repeatedly mentions the dimensions of tensors at different states of the calculations. Therefore the following shorthand notation is used:

- T The length of a sample, i.e. the number of token in an input sequence.
- C The dimension of a single token's representation vector. It is the same as d_{model}
- B The batch size. It describes how many samples are processed at once.

3.1.1 Embedding

The first part of the GPT architecture is the input embedding. A visualization of our approach can be seen in Figure 3.1.
The model gets a sequence of token as it's input. These token are usually represented as IDs. In the token embedding layer, each token is translated into a vector representation of dimension C by using a torch.nn.embedding layer. This contains a learnable matrix, for which a token's id is used to identify a row. That row is used as the vector embedding of the token. Therefore a sequence of T token is embedded into T individual vectors of dimension C, which are then stacked into a matrix of shape $(T \times C)$.
The fixed positional encoding employed in this work is based on a fixed formula, which can be found in Equation 2.2. This formula exactly defines every entry in the encoding vector for each input position. While it would be possible to directly implement this formula as the positional encoding layer, it is not very efficient to repeat the calculations everytime an input needs to be encoded. Since the positional

[1] https://github.com/Simrichter/GluePT

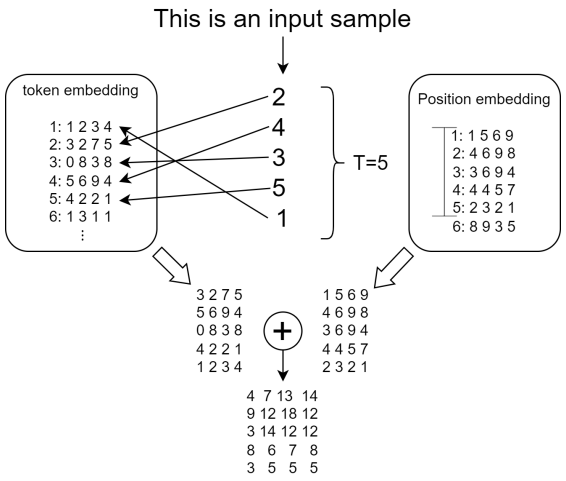


Figure 3.1: An example of how an input sequence is tokenized and embedded. In this example, the embedding dimension is $C = 4$, the number of token in the input is $T = 5$ and the maximum sequence length is $T_{max} = 6$

encoding always results in the same embedding vectors for the same positions, it is much more efficient to calculate all embedding vectors once and store them in a matrix. When stacking them row-wise, that matrix behaves similarly to the weight matrix used in the token embedding layer. The only difference is that this matrix is fixed and is therefore not optimized during training. Even though a fixed formula can encode arbitrarily large positions, such an approach of calculating the encodings in advance is only possible for a limited number of encoding vectors. With $PE(i, pos)$ as defined in Equation 2.2 and the maximum sequence length in pretraining T_{max}, we calculate the positional encoding matrix of shape $(T_{max} \times C)$ as

$$\begin{pmatrix} PE(1,1) & \dots & PE(1,C) \\ \vdots & \ddots & \vdots \\ PE(T_{max},1) & \dots & PE(T_{max},C) \end{pmatrix}, \tag{3.1}$$

such that all positions occurring in pretraining can be encoded using only that matrix.
If an input sequence in finetuning or inference exceeds this limit, we use the positional encoding matrix and fall back to the formula to compute all further positions. We also add code to detect a change in the parameter controlling the maximum sequence length of the model. In case of a new T_{max}, a matching positional encoding matrix is calculated that replaces the old one.
The positional encoding for a given input sequence of T token can easily be obtained by cropping the positional encoding matrix to T rows. This results in a matrix of shape $(T \times C)$, which is the positional encoding of the sequence.
To obtain a token's final input embedding, the token embedding and the positional embedding, both matrices of shape $(T \times C)$, are combined using element-wise addition.

3.1.2 Batching

When training with large GPUs, there are usually enough capacities to train on multiple samples simultaneously. To speed the training up, these samples are packed together into a single tensor. Contrary to popular belief, Shallue et al. [2018] find that larger batchsizes do not generally hurt out-of-sample performance, given well-tuned hyperparameters.
Batching is applied by collecting B text sequences, all of the same length T, into a tensor of shape $(B \times T)$. All operations in the GPT network are performed on each sequence independently. Therefore the embedding layer embeds each sequence individually, resulting a tensor of shape $(B \times T \times C)$.
In PyTorch, multiplication of tensors with a dimensionality higher than two is treated as a batch multiplication. The two innermost dimensions are called matrix dimensions and all other batch dimensions. The matrix multiplication happens only on the two matrix dimensions and is executed in parallel across all batch dimensions. For example, a multiplication of a tensor of shape $(A \times B \times C)$ with a tensor of shape $(C \times D)$ will perform a standard matrix multiplication over the matrix dimensions, i.e. $(B \times C) \cdot (C \times D) \rightarrow (B \times D)$, but for A times in parallel. This results in a tensor of shape $(A \times B \times D)$. Thanks to this batch multiplication, most calculations can be implemented for batches just like they would without them. For example the creation of Q, K and V as described in Equation 2.9 still holds for a batched input tensor x of shape $(B \times T \times C)$, resulting in tensors of shape $(B \times T \times d_k)$

3.1.3 Masked multi-head self-attention

The self-attention mechanism is the heart of a GPT network, hence it's implementation is now described in detail. Equation 2.13 is a good starting point, but it does not provide a good understanding of what happens in the attention module. This section gives an overview over the individual steps, the shapes of the tensors and additional optimizations.

Adding Multiple Heads:

As there are h attention heads performing independent attention in parallel, there have to be h sets of Q_n, K_n and V_n tensors. These are created following the Equation 2.9, but adding a new batch dimension to the weights W_q, W_k and W_v, which represents the attention heads. The weight tensors then have the shape $(h \times C \times d_k)$. However, when multiplying the weights with the input tensor x, this new dimension would collide with the already existing batch dimension of the input. To solve this problem, an additional dimension of size 1 needs to be introduced to the input x, resulting in the shape $(B \times 1 \times T \times C)$. When having a batch dimension paired up with a dimension of size 1, PyTorch automatically expands the tensors to be of equal size, without having to copy data. This feature is called broadcasting. If a tensor x of shape $(B \times 1 \times T \times C)$ is multiplied with W_q of shape $(h \times C \times d_k)$, the resulting tensor Q will be of shape $(B \times h \times T \times d_k)$.
After the matrices Q, K and V of shape $(B \times h \times T \times d_k)$ are created, Q and K^T are multiplied, resulting in a tensor of shape $(B \times h \times T \times T)$. The result is then scaled by $\frac{1}{\sqrt{d_k}}$.

Masking:

Since the attention module is used in the decoder-stack, information flow against the reading direction needs to be prevented, as discussed in Section 2.2.2. This is done by adding a causal mask of shape $(T \times T)$ to the result of $QK^\top$ element-wise[2]. Such a mask is filled with $-\infty$ in the upper triangular section excluding the diagonal. An example mask for a context length of $T = 4$ is shown in Figure 3.2.

$$\begin{pmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.2: An example of a causal masking matrix of shape $(T \times T)$ with $T = 4$. The upper triangular section is filled with $-\infty$ and everywhere else the mask contains zeros.

After applying this masking through element-wise addition, the result is a matrix that contains only one value that is different from $-\infty$ in it's first row, two in the second row and so on. When feeding this matrix into the softmax function, the upper triangular part is zeroed while all remaining entries in a row sum to one. Because both masking and softmax do not change the shape of the tensor, the score is still of shape $(B \times h \times T \times T)$.
That score is then multiplied with V of shape $(B \times h \times T \times d_k)$ to obtain an output tensor of shape $(B \times h \times T \times d_k)$.
Because this tensor contains individual representation vectors for each attention head, they are concatenated along the last dimension of the tensor to get a single representation vector for each position[3]. This is implemented by swapping the second and third dimension resulting in the shape $(B \times T \times h \times d_v)$. This tensor is then reshaped into three dimensions, collapsing the last two dimensions h and d_k into a single dimension C. This is possible because $d_k = \frac{C}{n_{heads}} \Leftrightarrow d_k \cdot n_{heads} = C$. The resulting tensor of shape $(B \times T \times C)$ is then passed through a position-wise linear layer with an output size of C. Therefore the final output of the attention module is a tensor of shape $(B \times T \times C)$, just like it's input.

3.1.4 Improving the Attention Module

Even though the implementation described above works correctly, two improvements have been made in the actual code. One change affects the creation of Q, K and V and aims to increase the models training and inference speed. The other change does not yield a noticable speedup but prevents a bug that sometimes occurs in combination with another technique called autocasting.

Increasing the Computation Speed

The presented method of creating the Q, K and V tensors is correct, but involves three matrix multiplications with multiple batch dimensions. Also three vector additions are needed when using a bias. However the calculations of Q, K and V can be combined into a single projection and the number of batch dimensions can be reduced by one. This does not change the total number of parameters and even though this affects only the very first step of the attention mechanism, it provides a significant increase

[2] Again, broadcasting applies this masking to all samples in the batch and to all heads in parallel.

[3] Each individual attention head calculates an embedding for each token position. Concatenating along the embedding dimension does not interfer with the number of token (T) or the different batches (B). It only alters the embedding vectors

in computation speed.
Such an improvement can be achieved by treating the multiple heads as if they were all concatenated in a single dimension. This removes the need for an additional head dimension in the calculations. The same can be applied to the three individual projections for Q, K and V by imagining them to be concatenated. Doing so allows to calculate everything in a single large projection which is then split into the individual components that are needed in the following steps.
Similar to how the results of the individual attention heads are concatenated into a single vector, a single vector can be projected and then split into the individual heads. This exploits the fact that $d_K = \frac{C}{h}$ and thus a single tensor of embedding dimension C can be split into h tensors each of size d_k.
The same trick is applied to the calculation of Q, K and V by using a weight matrix of shape $(C \times 3C)$ for the projection and then splitting the result into the three matrices, each of size $(C \times C)$.
All in all the input is once projected into a tensor of shape $(B \times T \times 3C)$, which is then disassembled into three tensors Q, K and V each of shape $(B \times T \times C)$. Each of these tensors is then further split into the individual heads, resulting in three tensors of shape $(B \times T \times h \times d_k)$. To achieve the desired shape $B \times h \times T \times d_k$, the second and third dimension are swapped.
At the end of this procedure, Q, K and V have exactly the same shape as before. The computation speed increases because the benefit of having only one large matrix multiplication outweighs the cost of additional splitting and reshaping operations.

Increasing the Stability when using Autocast

PyTorch already provides an implementation of scaled dot product attention. It expects the already projected tensors Q, K and V as input and returns the result with a shape of $B \times h \times T \times d_k$.
Even though the manual implementation described above is just as fast as the PyTorch implementation, it has a drawback. When using autocast[4], which is another technique for speeding up the calculations, sometimes a bug occurs. This bug causes an over- or underflow in the gradients, which eventually results in the loss becoming NaN. If that happens, further training is not possible.
The PyTorch implementation does not run into such problems, which is why it is activated when autocast is used. This change does not increase the computation speed by itself, but makes the stable use of autocast possible, which in turn provides a speedup.

3.1.5 Additional Optimizations

Automatic Mixed Precision

To increase training speed, automatic mixed precision [Micikevicius et al., 2017] is applied to the training. We use the PyTorch implementation, which includes autocast and gradient scaling. Autocast is a feature that automatically selects the optimal datatype for every operation. In some cases the full precision and dynamic range of a 32-bit floating point number (float32) is needed, in other cases a 16-bit floating point number (float16 / half) is sufficient. Using 16-bit datatypes wherever possible drastically reduces computation time.
However it comes with a drawback. When a computation in a forward pass is done on float16 numbers, the gradients will also be stored in float16. Since float16 is less precise than float32, very small gradients might not be representable and could potentially become zero. In some cases an overflow is also possible. Gradient scaling can prevent these problems by scaling the gradients to a safe magnitude without affecting the training.

[4] See documentation at https://pytorch.org/docs/stable/amp.html

Compiling the Model

With version 2.0, PyTorch introduced a feature called compile[5]. This feature activates the so-called just-in-time compilation, which tries to optimize python code while it is running. In this work, compile is applied to the GPT model during pretraining to reduce training times. It is not active when finetuning the model. This is because the length of the input samples varies during finetuning. With dynamic shapes the compile feature needs a very long warmup phase, which takes significantly longer than the time saved afterwards by the compiled model. Therefore the compile feature is only used during pretraining.

Gradient Clipping

To improve training stability, gradient clipping is used. It prevents exploding gradients that could cause instable training behaviour by limiting all gradients to a maximum value. There are two variants of gradient clipping. Clipping by value simply limits the values of each gradient to a certain threshold. We use clipping by norm, which first calculates a norm over all gradients. If that norm is larger than the threshold, all gradients are rescaled. A gradient g is clipped by norm to a threshold c using the equation

$$g = \begin{cases} \frac{c \cdot g}{\|g\|} & , \text{ if } \|g\| > c \,, \\ g & , \text{ else} \,. \end{cases} \tag{3.2}$$

When used in combination with gradient scaling, the gradients need to be unscaled before they can be clipped.

Gradient Accumulation

Because GPTs are generally quite large models, they can quickly max out the GPU memory. This can be counteracted by reducing the amount of training samples that are processed in a forward pass. However reducing the batch size may affect the model's performance. Gradient accumulation allows to perform a training step with a batchsize so large, that it would not fit in the GPU's memory in a single pass.
Typically one iteration of training consists of a forward pass, a backward pass and the updating step of the optimizer. The forward pass calculates the model's output. A suitable loss function is applied to calculate the difference between the model's output and the correct target, which the model should ideally have predicted. In the backward pass this difference is then propagated backwards through the calculations in a reversed order, updating the gradients for every parameter. When performing a step, the optimizer calculates a new value for each parameter based on it's gradient as well as other parameters that depend on the specific optimizer. After the update is performed, all gradients are zeroed to not interfer with the next iteration.
When using gradient accumulation, this process is slightly different. Gradient accumulation splits the forward and backward pass into multiple smaller passes. The full batch, that is too large for a single forward pass, is split into multiple microbatches. For each microbatch, a forward pass and a backward pass is performed. However the optimization step is not performed until the whole batch has been passed through the model. Because the gradients are not zeroed between passes, the gradients accumulate over the multiple passes. After the whole batch has been processed, an optimization step is performed on the gathered gradients. Only then the gradients are reset to zero.
Even though the gradients accumulate over multiple smaller passes instead of a single large one, both approaches are equivalent for training. Gradient accumulation has only a marginal impact on training time, but enables training with a batch size that would otherwise not be possible.

[5] https://pytorch.org/docs/stable/generated/torch.compile.html

Approximated GELU

The Gaussian Error Linear Unit (GELU) can be approximated using the equation

$$\text{GELU}(x) = 0.5x(1 + \tanh[\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)]) \tag{3.3}$$

from Hendrycks and Gimpel [2016]. This approximation yields greater feedforward speed but comes at the cost of exactness. In this work the tanh approximation is used.

Setting the Gradients to None

Another small optimization is to set the gradients to 'None' after an optimization step, instead of zero. This can change the training behaviour in certain situations, but it does not affect training in our case. Setting the gradients to 'None' increases the training speed slightly and reduces the memory footprint.

3.2 Training Loop

The training loop is an essential component in the process of training a neural network. It is a control structure that manages the training process. The training loop is named after a large loop in it's structure. This loop usually contains code that manages everything needed to perform a single optimization step. The code inside the loop is responsible for forward and backward passes as well as the calculation of the loss and finally the optimization step and the resetting of all gradients. On it's own it performs an independent step. An outer loop is used to repeat these steps until the desired number of training iterations has been performed.
Our training loops contain not only one, but two loops. The inner loop repeats until all samples in the dataset have been used in training once. Therefore the inner loop accomplishes training for one epoch. Since we mostly train our models for more than one epoch, the outer loop is used to repeat the training until the desired number of epochs is reached. Although the two loops could in theory be collapsed into a single one, this split into two loops guarantees that each epoch contains every batch exactly once and that the order in which the samples are trained varies between epochs[6].
To incorporate the optimizations described above, the training loop gains a considerable amount of additional complexity.
The use of gradient accumulation means that one optimization step consists of multiple forward and backward passes. Therefore a third loop is needed. This loop repeats the code that manages the forward and backward pass on a microbatch until the total batchsize is reached and the optimization step can be performed.
Automatic mixed precision is a feature that is provided in the PyTorch library. Therefore most of it's complexity is hidden. Casting the parameters and variables in a model's forward pass is completely managed by a context manager[7] called autocast. Everything that is needed to use automatic mixed precision is to execute all forward passes and loss calculations inside an autocast environment. It is not recommended to perform the backward passes inside it, because by default the backward operations already adapt to the datatype used in the corresponding forward operations.

[6] Assuming that the data is independently, identically distributed, shuffling the sample order between epochs prevents the model from overfitting to random patterns in the data order.

[7] In python a context manager is a structure that is used to condition the execution of code on the success of a preceeding operation. It works similar to a try catch block and is used with the keyword "with".

The PyTorch library also provides support for gradient scaling in form of the class GradScaler[8]. Before the backward pass can be performed, the loss of the forward pass needs to be scaled. On the scaled loss, the usual backward pass can be performed. Because the loss is multiplied by a certain factor, all gradients also emerge scaled. However the optimizer must not be affected by the scaling, which is why all gradients need to be unscaled and checked for overflows or underflows before the optimizer can perform a step. This is automatically handled by passing the optimizer object to the GradScaler's built-in function `step()`. As the GradScaler internally uses variables to approximate a good scaling factor based on the training history, the GradScaler's state needs to be stored and reloaded when stopping and continuing training from a checkpoint. Also an update has to be performed with every step of the optimizer.
Gradient clipping by norm is also provided by PyTorch in form of the function `clip_grad_norm()`, which takes all the parameters and the threshold value and automatically performs the norm calculations and the clipping. In combination with gradient scaling however it is not that simple anymore as the clipping must be done on the actual gradient values. Therefore all the gradients need to be unscaled before they can be clipped. This does not noticeably slow down the training speed as the gradients would have been unscaled before the optimization step anyway[9].
Since we use a non-constant learning rate, it has to be updated after every optimizer step[10]. The learning rate is calculated in dependence on the total number of performed steps and then applied to every parameter group[11] in the optimizer. We use schedules, which, after a short warmup phase, decay the learning rate using cosine-decay for pretraining and linear decay for finetuning. The exact formulas are shown in Equation 4.7 (cosine schedule) and Equation 4.8 (linear schedule).
Additionally the training loop also contains code that evaluates the trained model from time to time to track the progress. The occuring losses are also saved in regular intervalls to be able to observe the convergence behaviour while training. Furthermore checkpoints are saved in regular intervalls so that training can be interrupted and continued at another time.

[8] https://pytorch.org/docs/stable/amp.html

[9] The `step()` function of the GradScaler detects if the gradients have already been unscaled and skips that step if it is not necessary.

[10] Updating the learning rate even more frequently would be pointless because the learning rate is only used in optimization steps.

[11] Our optimizers need two parameter groups because weight decay is not applied to all weights

Different Versions of the Training Loop

This work uses two slightly different versions of a training loop, one for the pretraining and another one for the finetuning. Even though they differ, they mostly follow the same concept and are only adapted to the different ways in which the samples are provided. The training loop used for pretraining simply loads a training batch with every iteration of the second outermost loop, splits it into microbatches and then performs multiple passes. This is possible, because all pretraining samples are of the same length.
The finetuning loop cannot follow this scheme, because the samples vary in length and can therefore not be packed together into a single tensor. Forming a tensor that contains multiple samples of different length is only possible with padding. However a big advantage of the GPT architecture is that it does not require padding to work with non-constant input lengths. Padding is not used in pretraining and the used vocabulary does not even contain a padding token. Therefore such a padding approach is not trivial. It could be solved by implementing a masking that blocks out all padded positions in the attention layers, but it is much effort and would require altering a key component of the GPT architecture. Therefore we use another solution that is much easier to accomplish. For finetuning we treat every sample as an individual microbatch and perform a gradient accumulation step for each sample in the batch. This does not affect the scores of the resulting model, since an optimizer step is still only performed after all samples of a batch have been forwarded.
On the other hand it affects the training speed drastically as it almost[12] nullifies the speed increase of batching. This drawback is acceptable for finetuning because it is usually done on small datasets when compared to the ones used for pretraining.
Because the finetuning samples cannot be loaded as a batch, they must be loaded individually. This requires a slightly different structure in the training loop, especially in the third loop that handles the gradient accumulation. Nevertheless, both training loops are conceptually equivalent.

[12] A small benefit over single sample training remains, because the optimizer does not have to perform a step after every sample.

3.3 Data Preprocessing / Loading

The first step of training is to download the datasets. All datasets used throughout training are loaded from the platform Huggingface[13] by using the `load_dataset()` function of the library `datasets`.

3.3.1 Preprocessing

After a dataset is downloaded, the raw text data is tokenized using the GPT2 tokenizer. That tokenizer is provided by OpenAI in form of the `tiktoken` library[14]. The tokenizer's encode function expects a string input and returns a list of token IDs. This list of IDs is converted to a torch tensor and saved. Further preparation is not necessary because the datasets have already been processed by the publishers. The embedding of the token happens in the GPT model during training and cannot be outsourced into the preprocessing as the embeddings are subject to change while training.

Pretraining Samples

In pretraining, the used dataset is a long sequence of text from which the individual samples are taken. Generating a sample can be imagined as moving a sliding window of the desired context length along the text sequence and using the text inside that window for training. The stride with which the window moves can be altered, leading to different results. In general, a stride larger than the context length is not advisable, as this wil skip some token in the dataset. Using a stride of equal size as the context length guarantees that every position in the dataset will be contained in exactly one sample. Reducing the stride even further causes adjacent samples to overlap. To some extent, this overlap may be beneficial for training, as this causes an individual token to appear at different positions in different samples. Furthermore, a smaller stride increases the number of samples that can be extracted from a given text sequence[15]. However, for pretraining we consider the dataset to be sufficiently large, which is why it is not necessary to increase the dataset artificially. In this work, the stride is chosen to be half of the context length, which means that every individual token will occur in two samples. We suspect that this choice increases the effectiveness of pretraining for rare token that might not occur at multiple positions in the samples on their own.
When using a sliding window to generate samples, the special case of reaching the end of the text needs to be considered. We solve this edge case by reducing the stride on the last sample so that the last token in the text is also the last token in the sample, while maintaining full length. Although this creates a small bias towards the few token that might be included in three samples, we assume that this is negligible for sufficiently large data sets.

Finetuning Samples

In contrast to the pretraining dataset, the finetuning datasets already contain individual samples that can be used directly for training. Depending on the task, a sample can consist of multiple inputs, which are then usually concatenated in the finetuning process. Because finetuning falls into the category of supervised learning, the datasets also contain labels that need to be handled together with their associated inputs. All of these aspects are considered when storing and loading the samples while training.

[13] huggingface.co
[14] https://github.com/openai/tiktoken
[15] This comes at the cost of creating more and more similar samples.

3.3.2 Data Loading

During pretraining, a training step is performed on a whole batch of samples. That is why several pretraining samples must be generated and batched together into a tensor before they can be used in training. For finetuning it is more practical to load the data as individual samples, as each sample passes throug the network individually anyway due to the varying length of the samples.
During training the used datasets are held in the server's RAM to reduce access times. Because the GPT network's forward and backward passes happen on the GPU, the samples need to be transferred from the RAM to the GPU's memory. Since the GPU is used to significantly accelerate the training, a key to fast training is to keep the GPU constantly working. Therefore, idle times, in which the GPU is waiting on new data from the CPU, must be avoided as much as possible. A solution is to use the "DataLoader" class from PyTorch, which allows to asynchronously prepare the next batch of samples while the GPU is still processing the last one. In case of the preparation time taking longer than the actual training step on the GPU, multiple worker threads can be used to load the batches for more time steps into the future. This technique prevents a bottleneck in preparing the data for training and ensures that the next batch is ready by the time the GPU has finished the last one. The only remaining overhead comes from the transfer of the next batch to the GPU. This could be solved by transferring the data in a similar asynchronous manner, but this would take up more space in the memory of the GPU, as both the old and the new batch would be on the GPU for a short duration. To reduce the memory footprint and avoid out-of-memory errors during training, the batches are transferred at the beginning of a training step after the previous step has finished.

4 Experiments

4.1 Our Models

We train two models, with the large model having over three times as many learnable parameters as the small one. The architecture hyperparameters follow the choices Radford et al. [2019] made for their GPT2 model in it's small and medium sized versions. Note that even though we follow the exact model dimensions of GPT2, our total number of learnable parameters is significantly smaller. This is due to differences in the embedding layers described in the following section.

Table 4.1: An overview over the architecture hyperparameters for our two models.

Model	Total parameters	Learnable parameters	Layers	Heads	d_{model}	d_k
Small model	124M	85M	12	12	768	64
Large model	355M	302M	24	16	1024	64
GPT2-small	124M	124M	12	12	768	64
GPT2-medium	355M	355M	24	16	1024	64

Tokenization and Embedding:

The tokenization algorithm by Radford et al. [2019] has a vocabulary containing 50257 token. In order to reduce the total amount of learnable parameters, the weight matrix of the token embedding layer is taken from the fully pretrained GPT-2 model, downloaded via Huggingface[1] and frozen in the training process. As described in Section 2.2.3, the weights of the token embedding layer are reused in the output head. Therefore the output head is also frozen. The use of an already trained embedding weight reduces the number of learnable parameters by $50257 \cdot 768 \approx 39M$ parameters for the small model and by $50257 \cdot 1024 \approx 51M$ parameters for the large model. Another small portion is trimmed of the learnable parameters by using a fixed positional encoding as described in Equation 2.2 instead of the learned version usually found in GPTs.
Even if this parameter reduction is hardly significant, the use of a fixed positional encoding has another advantage. The score matrices inside the attention modules are of quadratic size to the context length. Therefore a smaller context length allows for faster and more memory efficient training. When looking at the finetuning datasets, which are targeted in this work, only very few samples require a context length larger than 256. A learned positional embedding would require pretraining with the largest context length that occurs during finetuning. With a fixed encoding however the training can be performed on a context length of 256, as longer sequences can still be embedded. Even though this does not mean that the model will automatically extrapolate well, this approach is preferable given the very small number of cases in which such extrapolation is needed.

[1] https://huggingface.co/gpt2

GPT2 Models

In order to provide reference scores, we also finetune and evaluate two versions of the GPT2 model from Radford et al. [2019]. These models are pretrained by the authors and available via Huggingface. We compare our small model to GPT2-small and our large model to GPT2-medium, as they are similar in size. Since the GPT2 models are already pretrained, we only add a new output head and perform finetuning just like with our own models. We use the GPT2 models as a reference, because they are pretrained on a large corpus. Therefore, the results should provide a reference of what scores our models might reach with even larger amounts of pretraining.

4.2 Datasets

As training is usually split into pretraining and finetuning, the used datasets are grouped into two categories. Datasets used for pretraining are unlabeled and therefore larger in comparison to the datasets used for finetuning. For pretraining, we use different datasets for testing, but the final experiments are conducted using a subset of OpenWebText by Gokaslan et al. [2019]. All datasets are tokenized once using the tiktoken[2] tokenizer from OpenAI, and saved for later use.

4.2.1 Pretraining with OpenWebText

GPT1 is pretrained on the BooksCorpus dataset from Zhu et al. [2015]. This dataset contains over 11.000 books, totalling in around 5GB of text data. In this work, the OpenWebText dataset is used for pretraining. As it is a much larger dataset with around 40GB of text data, only a subset of the first 5GB is used to maintain comparability.
The OpenWebText dataset is an open-source replication of the WebText dataset described by Radford et al. [2019]. WebText is the result of ". . . building as large and diverse a dataset as possible in order to collect natural language demonstrations of tasks in as varied of domains and contexts as possible." [Radford et al., 2019]. By scraping outbound links from the social media platform Reddit[3], the platform's karma system can be used as a manual filtering. Only links created up to December 2017 are included. Wikipedia articles are excluded because they might overlap with data used for evaluation. The text is extracted from the websites by using a combination of the Dragnet [Peters and Lecocq, 2013] and Newspaper[4] content extractors. After de-duplication and some heuristic based cleaning, the resulting dataset contains slightly over 8 million documents totalling in 40 GB of text.
The WebText dataset has not been published, but Gokaslan et al. [2019] provide an open-source version called OpenWebText. OpenWebText recreates the WebText dataset following the described steps. Links are extracted from Reddit, de-duplicated, filtered to exclude non-html content and shuffled. After downloading, the documents are extracted using the newspaper python package. Non-English content is filtered out using fastText[5]. Near-duplicates are excluded by hashing the documents into 5-grams using local-sensitivity hashing and all documents with a similarity threshold of greater than 0.5 get removed. Short documents containing less than 128 token after tokenization are also removed. The resulting dataset contains roughly the same amount of documents as WebText.
In this work, only a subset of the first 5 GB from OpenWebText is used, totalling in roughly 1 million documents. This subset is downloaded from the parquet files of Huggingface's OpenWebText dataset [6].
An evaluation set is used to monitor the pretraining and to select good hyperparameters. The evaluation set consists of the last 0.5% of the data and is not used for training.

[2] https://github.com/openai/tiktoken
[3] https://www.reddit.com
[4] https://github.com/codelucas/newspaper
[5] https://github.com/facebookresearch/fastText
[6] https://huggingface.co/datasets/Skylion007/openwebtext/tree/refs%2Fconvert%2Fparquet/plain_text/partial-train

4.2.2 Finetuning on the GLUE Benchmark

The General Language Understanding Evaluation (GLUE) benchmark is used to compare various NLP models as objectively as possible. It is a collection of nine preexisting datasets suitable for benchmarking the performance of NLP models across different tasks and domains. To include as many models as possible, the tasks are designed to be broadly compatible, which means that the generative capabilities of the GPT network are not needed to solve them. Some tasks come with plenty of training data while others provide only limited training possibilities or have differences between the train and the test set. Wang et al. [2018] argue, that "GLUE therefore favors models that can learn to represent linguistic knowledge in a way that facilitates sample-efficient learning and effective knowledge-transfer across tasks.". The authors expect, that models trained solely on the respective training datasets without any sort of pretraining cannot achieve competitive results across all tasks.
As it is typical for benchmarks, a total score is assigned to the tested model to allow for a simple ranking. The individual scores on all nine tasks can be used to get a more detailed understanding of the strengths and weaknesses of the model. To obtain the scores, metrics are applied that differ between tasks. As all used metrics give results in the range of either [0, 1] or [-1, 1], they are multiplied by 100 and rounded to one decimal place for better human understanding. The total score is calculated as the average over the scores of the nine tasks. If a task is evaluated on more than one metric, the average of those metrics is calculated first and the averaging for the total score happens on these intermediate results.
While train and validation datasets of all nine tasks are published, the test labels of four tasks are held secret. These private labels "ensure that the benchmark is used fairly."[Wang et al., 2018]. To obtain the scores on these four tasks, the predicted labels have to be submitted on the GLUE benchmark's website[7].
In the following, all nine tasks of the GLUE benchmark are presented. An overview over the tasks, the used metrics and the sizes of train and test set are presented in Table 4.2. Examples from the task's datasets are shown in Table 4.3. The Finetuning happens on the training datasets that come with the tasks of the GLUE benchmark. They are considerably smaller than the ones used for pretraining and come labeled, therefore a supervised learning approach is used. The inputs consist of natural language text and are tokenized in the same way as before. CoLA and SST2 are single-sentence tasks and their samples can therefore directly be used as input to the model. For the other tasks, we concatenate all input sentences to form a single input. Since there is no inherent ordering between the two sentences of STS-B, we follow the approach of Radford et al. [2018] and create a sample for each of the two possible orders, in which the sentences can be concatenated. Both of these new samples are passed through the model and the two outputs are added together element-wise to form the final result[8].

[7] https://gluebenchmark.com/submit

[8] Radford et al. [2018] add the outputs of the decoder stacks and then pass the result to the output head. Since we do not want to alter our GPT architecture, we add the results after the output head, which is an equivalent solution.

Table 4.2: An overview over all GLUE tasks. The test sets written in bold have secret labels that are not publicly available. STS-B is a regression task, MNLI is three class classification. All other tasks are two class classification problems.

Corpus	**|Train|**	**|Test|**	**Task**	**Metrics**
			Single-Sentence Tasks	
CoLA	8.5K	**1K**	acceptability	Matthews correlation coefficient
SST-2	67K	1.8K	sentiment	Accuracy
			Similarity and Paraphrase Tasks	
MRPC	3.7K	1.7K	paraphrase	Accuracy / F1-score
STS-B	7K	1.4K	sentence similarity	Pearson / Spearman correlation coefficient
QQP	364K	**391K**	paraphrase	Accuracy / F1-score
			Inference Tasks	
MNLI	393K	**20K**	natural language inference (NLI)	Accuracy on matched and mismatched test sets
QNLI	105K	5.4K	question answering / NLI	Accuracy
RTE	2.5K	3K	NLI	Accuracy
WNLI	634	**146**	coreference / NLI	Accuracy

Table 4.3: Example samples of all nine GLUE tasks. STS-B is a regression task, MNLI has three classes, all other tasks are two class classification.

Task	Input	label	Objective
CoLA	They drank the pub	0 (unacceptable)	Grammatical correctness
	They drank the pub dry	1 (acceptable)	
STS-B	The man is playing the piano - The man is playing the guitar	1.6	Similarity
	A plane is taking off - An air plane is taking off	5	
SST2	goes to absurd lengths	0 (negative)	Sentiment analysis
	enriched by an imaginatively mixed cast of antic spirits	1 (positive)	
WNLI	The trophy would not fit in the brown suitcase because it was too big. - The trophy would not fit in the brown suitcase because the suitcase was too big.	0 (not_entailment)	Entailment
	I couldn't put the pot on the shelf because it was too tall. - The pot was too tall.	1 (entailment)	
RTE	Valero Energy Corp., on Monday, said it found "extensive" additional damage at its 250,000-barrel-per-day Port Arthur refinery. - Valero Energy Corp. produces 250,000 barrels per day.	0 (entailment)	Entailment
	Oil prices fall back as Yukos oil threat lifted - Oil prices rise.	1 (not_entailment)	
QNLI	What two things does Popper argue Tarski's theory involves in an evaluation of truth? - He bases this interpretation on the fact that examples such as the one described above refer to two things: assertions and the facts to which they refer.	0 (entailment)	Entailment
	How were the Portuguese expelled from Myanmar? - From the 1720s onward, the kingdom was beset with repeated Meithei raids into Upper Myanmar and a nagging rebellion in Lan Na.	1 (not_entailment)	
MRPC	That compared with $ 35.18 million, or 24 cents per share, in the year-ago period. - Earnings were affected by a non-recurring $ 8 million tax benefit in the year-ago period.	0 (not_equivalent)	Similarity
	Six months ago, the IMF and Argentina struck a bare-minimum $ 6.8-billion debt rollover deal that expires in August. - But six months ago, the two sides managed to strike a $ 6.8-billion debt rollover deal, which expires in August.	1 (equivalent)	
QQP	How is air traffic controlled? - How do you become an air traffic controller?	0 (not_duplicate)	Similarity
	What is the best self help book you have read? Why? How did it change your life? - What are the top self help books I should read?	1 (duplicate)	
MNLI	The supervisor instructed the controller to follow standard procedures for handling a no radio aircraft. - Standard procedures were followed for an aircraft with no radio.	0 (entailment)	Entailment
	The city is going to explode with Herron's artistic contributions! - Herron is donating some works of art to the city.	1 (neutral)	
	This analysis pooled estimates from these two studies to develop a C-R function linking PM to chronic bronchitis. - The analysis proves that there is no link between PM and bronchitis.	2 (contradiction)	

CoLA

The Corpus of Linguistic Acceptability (CoLA) by Warstadt et al. [2019] is a collection of sentences drawn from a broad variety of linguistics publications. The task is to predict if a given sentence is grammatically acceptable or not. Even though the original dataset is split into in-domain and out-of-domain parts, the version used for GLUE combines the two. CoLA is an unbalanced two class classification task with around 70% of the samples being labelled 1 ("acceptable"). The used metric is the matthews correlation coefficient. The labels of the test set are held private.

STS-B

The Semantic Textual Similarity Benchmark (STS-B) by Yang et al. [2018] is a collection of sentence pairs collected from news headlines, captions and other sources. The model has to predict the similarity of the given sentences on a scale from zero to five. A rating of five relates to equivalence (both sentences mean the same thing), while zero refers to two completely dissimilar sentences. STS-B is a regression task and the performance is measured by using the Pearson correlation coefficient and the Spearman correlation coefficient.

SST2

The Stanford Sentiment Treebank (SST) by Socher et al. [2013] is a sentiment analysis task. It's dataset is a collection of sentences from movie reviews and their corresponding sentiment labels. GLUE uses the binary version (SST2), in which the task is to predict whether a review is positive or negative. The individual samples are no coherent texts but only sentences or parts of sentences. SST2 is a two class classification task that uses accuracy as it's metric.

WNLI

The Winograd NLI is a natural language inference task based on the Winograd Schema Challenge by Levesque et al. [2012]. In the Winograd Schema Challenge, a model gets a text with a pronoun and a list of words as input. It then has to select the word that is the correct referent of that pronoun. Wang et al. [2018] convert this task into sentence pair classification by creating a second sentence that contains one of the words in place of the pronoun. The pair of original sentence and the newly formed sentence is used as input and the model has to predict if the original sentence entails the moddified one.
WNLI is a two class classification task and accuracy is used as the metric. The labels of the test set are not published.

RTE

The Recognizing Textual Entailment (RTE) task is a collection of four annual textual entailment challenges [Dagan et al., 2006, Haim et al., 2006, Giampiccolo et al., 2007, Bentivogli et al., 2009] that are based on news and Wikipedia text. The goal is to predict wether the first input entails the second. RTE is a two class classification task that uses accuracy as the metric.

QNLI

The Question-answering NLI (QNLI) is based on the Stanford Question Answering Dataset (SQuAd) [Rajpurkar et al., 2016]. It comes closest to what most people would likely expect from a language model. The original goal is to answer a question based on a paragraph from Wikipedia. The input is a pair of question and the reference text.
For GLUE, the dataset is converted into sentence pair classification. The model is only required to decide if the answer is contained in the reference text or not. Even though this removes the challenge of generating a valid answer, it still requires the model to put question and reference into context. QNLI is a two class classification task that uses accuracy as it's metric.

MRPC

The Microsoft Research Paraphrase Corpus (MRPC) by Dolan and Brockett [2005] is a dataset containing sentence pairs extracted from online news sources. The goal is to predict if the two given sentences are semantically equivalent. MRPC is similar to STS-B, but it is a two class classification task instead of regression. The classes are imbalanced, so the F1 score is used as a metric along with accuracy. Around 68% of the samples are labelled 1 ("equivalent"). For the F1 score, the class with the label 1 is defined as the positive class.

QQP

The Quora Question Pairs (QQP) dataset[9] contains question pairs drawn from the website Quora. All questions are written by users of the platform. The model has to determine the semantic equivalence of a given pair of questions, or in other words, find duplicates that may differ in their sentence structure but not semantically. QQP is an unbalanced two class classification task with 63% of the samples labelled 0 ("not_duplicate") in both the train and validation set. The test set has a different but unknown distribution as it is held private. Accuracy and F1 score are the metrics used. The class with the label 1 is defined as the positive class for the F1 score.

MNLI

The Multi-Genre Natural Language Inference Corpus (MNLI) by Williams et al. [2018] is a collection of premise-hypothesis pairs. The model has to decide if the premise entails or contradicts the hypothesis. A neutral relationship is also possible. The premises originate from ten different sources and the evaluation is performed both in-domain (matched) and cross-domain (mismatched). MNLI is a balanced three class classification task. Both the matched and mismatched sets have unpublished labels. They are individually evaluated using accuracy as the metric.

[9] https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs

4.3 Introducing a Baseline

When evaluating machine learning models on a benchmark, scores of other models help to better understand the own results. This section establishes two primitive baselines for all tasks of the GLUE benchmark. The used label selection strategies are majority guessing and weighted guessing. These baselines can be achieved with very simple methods and and act as a lower bound that the model should at least be able to beat.
The following definitions are used throughout this chapter when talking about two-class problems[10].

1. One class is defined as positive, the other as negative[11].
2. k describes the probability that a random sample from the test set is positive.
3. p is the probability of the model choosing the positive class label.
4. n is the number of samples.
5. A prediction on a sample can only have four possible outcomes, which leads to the following definitions:
   a) TP (true positive) describes the number of correct predictions of the positive class.
   b) TN (true negative) describes the number of correct predictions of the negative class.
   c) FP (false positive) is the number of times the model predicts the positive class when the negative class would have been correct.
   d) FN (false negative) is how often the model falsely predicts the negative class.
6. The expected values for these outcomes are
   $TP = n \cdot p \cdot k\,,$
   $TN = n \cdot (1-p) \cdot (1-k)\,,$
   $FP = n \cdot p \cdot (1-k)\,,$
   $FN = n \cdot (1-p) \cdot k\,.$

The GPT-models are finetuned on the training sets and the hyperparameter selection as well as the stabilization is based on the validation set. The actual test set is never seen, it is only used for the very last evaluation to obtain the scores. To ensure a fair competition, the baselines should therefore not be allowed to utilize information about the test set either. Thus, both baselines rely on minimal information about the validation set, which is assumed to be similar to the test set.

Majority guessing

To establish a lower bound on GLUE task performance, a model can be imagined that always predicts the same output class. On unbalanced sets, it is obviously a better choice to always choose the class that is more common than the other, as this will likely produce a higher number of correct predictions. Allowing such a majority guessing model to use general information about the label distribution makes it possible to identify the most frequent class and to constantly choose it.
Without loss of generality, the positive class of a two class problem can be assumed to be the more frequent

[10] All GLUE tasks apart from STS-B and MNLI are two-class problems.
[11] For all metrics except the F1 score, the choice of which class is positive does not affect the result.

class. In that case the model will always predict the positive label, i.e. $p = 1$. With that, the four possible outcomes of a prediction can be simplified to

$$\begin{aligned} TP &= n \cdot k\,, \\ TN &= 0\,, \\ FP &= n \cdot (1-k)\,, \\ FN &= 0\,. \end{aligned}$$

Weighted guessing

A viable alternative to majority guessing is a model that randomly chooses a label. Typically, random guessing selects all labels with the same probability. However, such a random guessing baseline provides only minimal contribution to understanding our model's performance on the tasks, because on most of the metrics it does not react to the dataset distributions of the different tasks. Therefore we moddify the random guessing model to select classes with probabilities matching the class distributions of the validation sets. We call this approach of choosing $p = k$ weighted guessing. In general, weighted guessing is equivalent to random guessing on balanced datasets, but beats it if the label distribution is not balanced. Just like majority guessing, such a weighted guessing model does not take either the given input or the guessing history into account and only makes use of knowledge about the distribution of the validation set. The possible prediction outcomes simplify to

$$\begin{aligned} TP &= n \cdot k^2\,, \\ TN &= n \cdot (1-k)^2\,, \\ FP &= n \cdot k \cdot (1-k)\,, \\ FN &= n \cdot k \cdot (1-k)\,. \end{aligned}$$

4.3.1 Accuracy

The metric accuracy acc is defined as the proportion of correct predictions in the total number of predictions.

$$acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

Majority guessing

With $p = 1$, the average accuracy of our majority guessing baseline is: $acc = \frac{n \cdot k}{n} = k$.
The majority score increases linearly with more unbalanced datasets and reaches it's minimum of 50 on a perfectly balanced dataset.

Weighted guessing

Under weighted guessing with $p = k$, the expected accuracy is

$$acc = \frac{n \cdot k^2 + n \cdot (1-k)^2}{n} = k^2 + (1-k)^2 = 2 \cdot k^2 - 2k + 1\,. \tag{4.2}$$

Similar to majority guessing, the accuracy of weighted guessing has a minimum at 50 and increases as the data becomes more unbalanced.

4.3.2 Matthews correlation

Although accuracy is the most frequently used metric throughout the GLUE benchmark, there are a few other metrics that need to be considered to establish a baseline. The matthews correlation coefficient is first used by Matthews [1975]. We use the re-proposed formulation

$$mcc = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4.3}$$

from Baldi et al. [2000]. Inserting the expected values for all possible outcomes yields

$$\begin{aligned} mcc &= \frac{n \cdot p \cdot k \cdot n \cdot (1-p) \cdot (1-k) - n \cdot p \cdot (1-k) \cdot n \cdot (1-p) \cdot k}{\sqrt{(n \cdot p \cdot k + n \cdot p \cdot (1-k))(n \cdot p \cdot k + n \cdot (1-p) \cdot k)(n \cdot (1-p) \cdot (1-k) + n \cdot p \cdot (1-k))(n \cdot (1-p) \cdot (1-k) + n \cdot (1-p) \cdot k)}} \\ &= \frac{p \cdot k \cdot (1-p) \cdot (1-k) - p \cdot k \cdot (1-p) \cdot (1-k)}{\sqrt{p \cdot k \cdot (1-p) \cdot (1-k)}} \\ &= 0\,. \end{aligned}$$

Therefore both majority guessing and weighted guessing are expected to reach a matthews correlation coefficient of zero.

4.3.3 F1-score

The F1-score is the harmonic mean of the metrics precision and recall.
Precision describes the proportion of how many positive predictions of the model are actually correct. Recall describes the ratio between correct positive predictions and the total amount of positive samples (i.e. the accuracy on the positive class).

$$\begin{aligned} Precision &= \frac{TP}{TP + FP} \\ Recall &= \frac{TP}{TP + FN} \end{aligned}$$

We use the definition of the F1-score as the F-measure [Chinchor, 1992] with $\beta = 1$:

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot \frac{TP}{TP+FP} \cdot \frac{TP}{TP+FN}}{\frac{TP}{TP+FP} + \frac{TP}{TP+FN}} = \frac{2 \cdot TP}{2 \cdot TP + FN + FP}\,. \tag{4.4}$$

Majority guessing

With $p = 1$, precision and recall can be written as

$$\begin{aligned} Precision &= \frac{n \cdot k}{n \cdot k + n \cdot (1-k)} = k\,, \\ Recall &= \frac{n \cdot k}{n \cdot k + 0} = 1\,. \end{aligned}$$

When inserting these, the F1-score collapses to

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot k}{k + 1}\,.$$

However this only holds under the assumption that the positive class is the most common. Since the F1-score is not symmetric, the positive and negative labels cannot be swapped without altering the result. Therefore the result is different if the majority guessing model always selects the negative class. Assuming the negative class to be more common, the probabilities for the four possible outcomes of a sample are

$$\begin{aligned} TP &= 0\,, \\ FP &= 0\,, \\ TN &= n \cdot (1-k)\,, \\ FN &= n \cdot k\,. \end{aligned}$$

Therefore $Precision$ and $Recall$ become

$$\begin{aligned} Precision &= \frac{0}{0+0} = 0\,, \\ Recall &= \frac{0}{0+n\cdot k} = 0\,. \end{aligned}$$

With these, the F1-score equals

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} = \frac{2\cdot 0 \cdot 0}{0+0} = 0\,.$$

This shows that the majority guessing model will not provide a good baseline if the negative class is more common. In that case it is better to still always predict the positive class, even though it might not be the most common one. For our baseline, we therefore use such a positive guessing model instead of the majority guessing model.

Weighted guessing

Under weighted guessing with $p = k$ the following holds for the F1-score:

$$F1 = \frac{2\cdot TP}{2\cdot TP + FN + FP} = \frac{2\cdot n\cdot k^2}{2\cdot n\cdot k^2 + n\cdot(1-k)\cdot k + n\cdot k\cdot(1-k)} = \frac{2\cdot k^2}{2\cdot k^2 - k + 1}$$

4.3.4 Accuracy on a three class problem

MNLI is the only task in the GLUE benchmark that has three classes. It uses accuracy as the metric, which resembles the percentage of correct classifications in relation to the total amount of tries. For a task with three classes, the differentiation of positive and negative is no longer applicable. In the following, k_c describes the probability that a randomly drawn sample is of class c. Analogue to this, p_c is the probability of our random model selecting class c. Since p and k resemble probabilities, the following still holds: $\sum_c p_c = 1 \wedge \sum_c k_c = 1$.
The accuracy of a three class problem acc_3 is then defined as:

$$acc_3 = p_0k_0 + p_1k_1 + p_2k_2$$

Majority guessing

Without loss of generality, the class with label zero can be assumed to occur the most in the dataset. In that case the majority model always selects label zero, i.e. $p_0 = 1$ and $p_1 = p_2 = 0$.
Therefore, the accuracy can be simplified to $acc_3 = k_0$.
As there are now three classes, the majority guessing model reaches it's lowest ($0.\overline{3}$) on a perfectly balanced dataset.

Weighted guessing

If the model predicts classes with the same probabilities as they occur in the dataset, i.e. $p_i = k_i \; \forall i \in \{0, 1, 2\}$, the accuracy can be calculated as follows:

$$acc_3 = k_0^2 + k_1^2 + k_2^2$$

Similar to majority guessing, weighted guessing reaches it's minimum value of $(0.\overline{3})$ on a perfectly balanced dataset.

4.3.5 Baseline on a regression task

With STS-B the GLUE benchmark also contains a regression task. Since there are no discrete classes, our baseline models are not applicable to this task.
STS-B uses the pearson and spearman correlation coefficients as metrics for evaluation. Following Freedman et al. [2007], the pearson correlation coefficient is defined as

$$\text{pearson}(X, Y) = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y}, \tag{4.5}$$

with a pair of random variables (X, Y), their covariance $\text{cov}(X, Y)$ and the standard deviations σ_X and σ_Y.
The spearman rank correlation coefficient is the pearson correlation coefficient applied to the rank R of the datapoints:

$$\text{spearmanr}(X, Y) = \text{pearson}(R(X), R(Y)) = \frac{\text{cov}(R(X), R(Y))}{\sigma_{R(X)} \sigma_{R(Y)}}. \tag{4.6}$$

To establish a baseline on a regression task, an intuitive approach that is somewhat similar to majority guessing is to select a constant value as the prediction for all samples. For example the center of the interval of all possible values could be chosen.[12]
However both the pearson correlation coefficient and the spearman rank correlation coefficient are undefined if one or both input sequences are constant, since the standard deviation σ of a constant variable is zero. Therefore an approach with constant values is not suitable to find a baseline.
Another approach that comes close to weighted guessing is to predicting labels by sampling from a simple probability distribution, for example an uniform or normal distribution. This approach introduces some random variance that allows the metrics to be calculated.
However all presented ways of establishing a baseline imply the self-chosen constraint of predicting an output without taking the specific input into account. The only information that may be used is the overall distribution of the targets. Therefore input and output are, by definition, noncausal. Any observed correlation is of random nature and likely cancels out over multiple evaluations. Therefore the performance of our random baseline on STS-B is expected to be zero. Without allowing the baseline models to use more information, it is impossible to obtain average correlation coefficients better than zero. We assume a baseline of zero on the task STS-B.

[12] Since all labels of STS-B are in the interval [0, 5], such a model would assign the label 2.5 to every sample.

4.3.6 The final baselines

Table 4.4: The expected scores of the two baseline models on the validation sets. The scores calculated by scaling the different metrics' results by a factor of 100. On QQP we substitute majority guessing with positive guessing. For the random guessing model, the following percentages for the classes labeled 1 are used: CoLA: 69.1, SST-2: 50.9, MRPC: 68.4, QQP: 36.8, QNLI: 50.5, RTE: 52.7, WNLI: 43.7.
For MNLI the percentages 35.4, 32.7 and 31.8 are used for the matched score and 35.2, 33 and 31.8 on the mismatched part.

Task	Metric	Majority guessing scores	Weighted guessing scores
CoLA	mcc	0	0
SST-2	acc	50.9	50
STS-B	pc/sp	-/-	0/0
MRPC	F1/acc	81.2/68.4	74.8/56.8
QQP	F1/acc	53.8/63.2	30.0/53.5
MNLI-m/mm	acc	35.4/35.2	33.3/33.3
QNLI	acc	50.5	50
RTE	acc	52.7	50.1
WNLI	acc	56.3	50.8
Total		39.1	38.4

Table 4.5: The actual results of the majority and random guessing models evaluated on the test sets. The predictions are based on the distributions of the validation sets and the scores come from evaluation on the actual test sets by submitting the majority guessing labels to the GLUE website and calculating the test set distributions based on the majority guessing scores. These distributions are then used to calculate expected scores for weighted guessing. On QQP we substitute majority guessing with positive guessing.

Task	Metric	Majority guessing scores	Weighted guessing scores
CoLA	mcc	0	0
SST-2	acc	49.9	50
STS-B	pc/sp	-/-	0/0
MRPC	F1/acc	79.9/66.5	73.1/56.1
QQP	F1/acc	29.9/82.4	13.6/58.6
MNLI-m/mm	acc	35.6/36.5	33.4/33.4
QNLI	acc	49.5	50
RTE	acc	49.7	50
WNLI	acc	65.1	51.9
Total		40.5	37.3

4.4 Training

For training the model, we use the PyTorch implementation[13] of the AdamW optimizer. AdamW [Loshchilov and Hutter, 2017] is a version of Adam [Kingma and Ba, 2015] that decouples the weight regularization from the adaptive gradient mechanism.
The parameters are split into two groups. Weight decay with $\omega = 1e^{-2}$ is only applied to the group containing all non bias or gain[14] weights, following the approach of Radford et al. [2018]. The filtering of the groups is done by checking whether a parameter has a shape with more than one dimension[15], following the approach of Andrej Karpathy's nanoGPT[16]. Gradient clipping is used with a threshold of 1.0.

Pretraining

Pretraining is done with a next token prediction objective, which means that the model has to predict the token that follows a given input sequence. This approach allows for self-supervised learning, since a given text sequence of n token can easily be splitted into an input sequence of $n-1$ token and the corresponding label, i.e. the n-th token. A benefit of the masking in the attention heads of a GPT network is that it is not necessary to consider training for shorter inputs. The masking allows to train sequence completion on multiple sequences of variable length at the same time by using the prefixes of the input sequence as individual samples. The corresponding labels are the prefixes of the same sequence, but shifted by one position to the right.

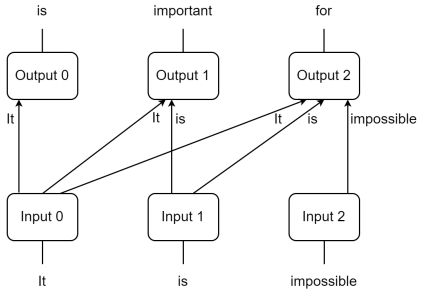


Figure 4.1: A simplified example showing how masking enables training on all prefixes of the input simultaneusly.

This training concept becomes clear when looking at Figure 4.1. Due to the masking, an output at a certain position pos can only be based on inputs up to that position. For example output one is based on the inputs one and two, but not three. Because of that, every output position pos is used to make an individual next token prediction with a prefix of length pos of the network's input. The target of this prediction is the token immediately following that prefix in the sequence, i.e. the token at position $pos + 1$. When looking at the simplified example in Figure 4.1, the output two predicts the token "important" based on the prefix "It is". The correct token would have been "impossible". This type of prefix training happens for every possible prefix of the input except for the empty string.
In our pretraining, we limit the maximum length of an input sample to 256 token, which is relatively small compared to other models [Radford et al., 2018, 2019, Devlin et al., 2018, Touvron et al., 2023]. Such a small context consumes less memory on the GPU and allows for faster training. As this work solely aims for the GLUE benchmark, a maximum sequence length of 256 is sufficient most of the time. Throughout all nine datasets there exist only 10 samples in three of the test sets[17] that would require a slightly longer sequence length. Because a fixed positional encoding is used, these few samples can still be processed, hoping that the model can extrapolate to longer sequences. Considering the worst case scenario of the

[13] https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html
[14] gain weights are used in the LayerNorm
[15] Bias and gain weights are vectors, all other weights are tensors of higher dimension.
[16] https://github.com/karpathy/nanoGPT
[17] Those sets are QQP, MNLI and QNLI

model guessing all 10 of the samples wrong, the damage would likely not exceed 0.1 points in total. Therefore, the improvement in training speed is preferred.
We decay the learning rate using a moddified cosine schedule with warmup. Similar to the approach of Radford et al. [2018], the learning rate is linearly increased from zero to $lr_{max} = 2.5e^{-4}$ over the first $i_{warmup} = 400$ optimization batches and then annealed to $lr_{min} = 1e^{-5}$ using a cosine schedule. For any further training beyond i_{total}, the learning rate is constant at lr_{min}. With these variables, our learning rate for pretraining is scheduled as

$$lr(i) = \begin{cases} lr_{max} * \frac{i}{i_{warmup}} & \text{, if } i < i_{warmup}\,, \\ lr_{min} + 0.5 * (lr_{max} - lr_{min}) * \left(1 + \frac{\cos\left((i - i_{warmup}) * \pi\right)}{i_{total} - i_{warmup}}\right) & \text{, if } i_{warmup} \leq i \leq i_{total}\,, \\ lr_{min} & \text{, otherwise}\,. \end{cases} \tag{4.7}$$

The model is trained over 15 epochs with a batchsize of 256 full-length samples. The employed gradient accumulation splits the batchsize into four microbatches with a size of 64 per forward pass.
One Nvidia A100-SXM4-40GB GPU is used for training. A single iteration takes around 0.56 seconds for the small model and around 1 second for the large model. One epoch contains 34481 batches. The total computing time required is approximately 100 hours for the small model and 200 hours for the large model.
Throughout training, checkpoints are saved that contain the model's states with increasing amounts of pretraining. This yields multiple model states that differ only in the amount of pretraining. These checkpoints are then finetuned and evaluated on the GLUE benchmark.

Finetuning

The goal of finetuning is to adapt an already pretrained model to a new task. Usually this is done with supervised learning and a comparably small dataset. In this work, the pretrained checkpoints are evaluated on the nine tasks of the GLUE benchmark. The evaluation on a task is done by letting the model predict labels on the task's test set. These predictions are then compared to the actually correct labels and a specific metric is used to assign a score.
Before the models are evaluated, they are finetuned to the specific tasks. This finetuning starts with the pretrained model and continues training with a lower learning rate on the training dataset of a task. Because the objective changes from language modelling to either classification or regression, depending on the task, the output head needs to be moddified. It's linear layer that was used in the pretraining process is replaced by a new linear layer with another output dimension, suitable for the new task. Apart from this new head, all weights are carried over from pretraining.
The finetuning happens on a smaller learning rate with a different schedule than the pretraining. The learning rate for finetuning is linearly increased from zero to $lr_{max} = 5e^{-5}$ over the first $i_{warmup} = 0.2\% \cdot i_{total}$ optimization batches. It then decreases linearly until $lr_{min} = 0$. With these variables, our learning rate for finetuning is scheduled as

$$lr(i) = \begin{cases} lr_{max} * \frac{i}{i_{warmup}} & \text{, if } i < i_{warmup}\,, \\ (lr_{max} - lr_{min}) * (1 - (i - i_{warmup})/(i_{total} - i_{warmup})) + lr_{min} & \text{, if } i_{warmup} \leq i \leq i_{total}\,, \\ lr_{min} & \text{, otherwise}\,. \end{cases} \tag{4.8}$$

Some tasks have shown to be quite instable and produce strongly varying results when performing multiple finetuning runs with the exact same settings. To handle this problem, multiple finetuning runs are performed. Only the run that produces the highest score on the validation set is kept. This method is a reliable way of sorting out underperforming runs. However it requires much more computation time. Because of that, this approach is only done on the smaller datasets of GLUE. This is not a problem because the observed instabilities occur mostly on the smaller datasets. The amount of finetuning tries per task can be found in Table 4.6.
Every task is finetuned individually, always starting from a pretrained checkpoint. The training hyperparameters need to be adapted to each specific task, as the tasks have strong variance regarding dataset size, overfitting affinity etc. Even though the settings change from task to task, they remain constant between different models for the same task. This ensures comparability as the only changing factor is the amount of pretraining of a model.

Table 4.6: Hyperparameter settings for finetuning the GLUE tasks.

Task	Epochs	Runs	Evaluation interval
CoLA	3	7	50
STS-B	3	5	50
SST2	3	2	50
WNLI	6	10	5
RTE	3	10	50
QNLI	2	1	50
MRPC	3	3	50
QQP	1	1	500
MNLI	1	1	500

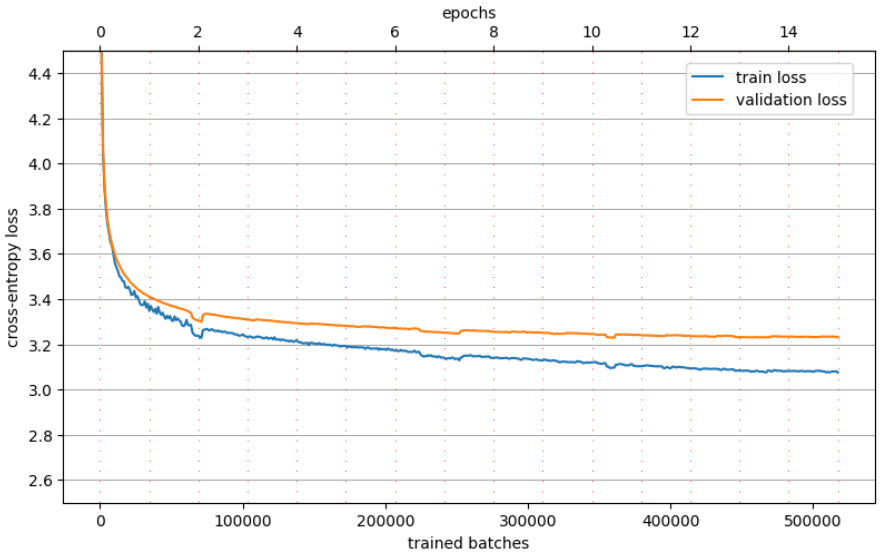
After a model is finetuned, it is evaluated on the test sets of all tasks to obtain the reached scores. For each task it has to predict the labels for the test set. The predicted labels of all nine test sets are collected in tsv files and zipped. This zip-file can then be submitted the GLUE benchmark's website[18] to obtain the scores. Five of the tasks could also be evaluated without submitting, because their labels are publicly available. For the other four tasks however it is necessary to submit the predicted labels, as the true labels are held secret.

[18] Submissions are free, but limited to 2 per day.

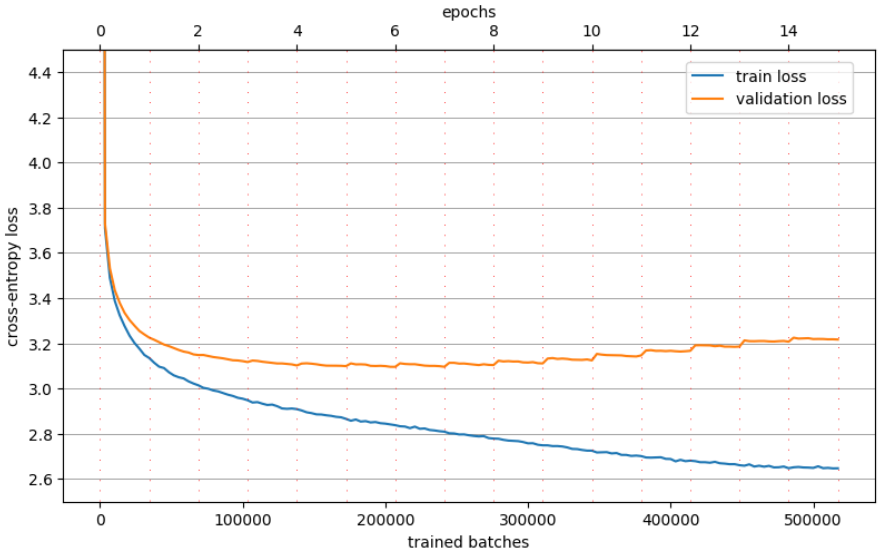
5 Results

5.1 Impact of Pretraining on the Results

5.1.1 Pretraining results



(a) small model



(b) large model

Figure 5.1: The pretraining loss curves on the training and validation sets. The x-axis shows the number of optimization batches, the y-axis shows the reached cross-entropy loss. The red dotted lines mark the corresponding number of trained epochs.

Small model

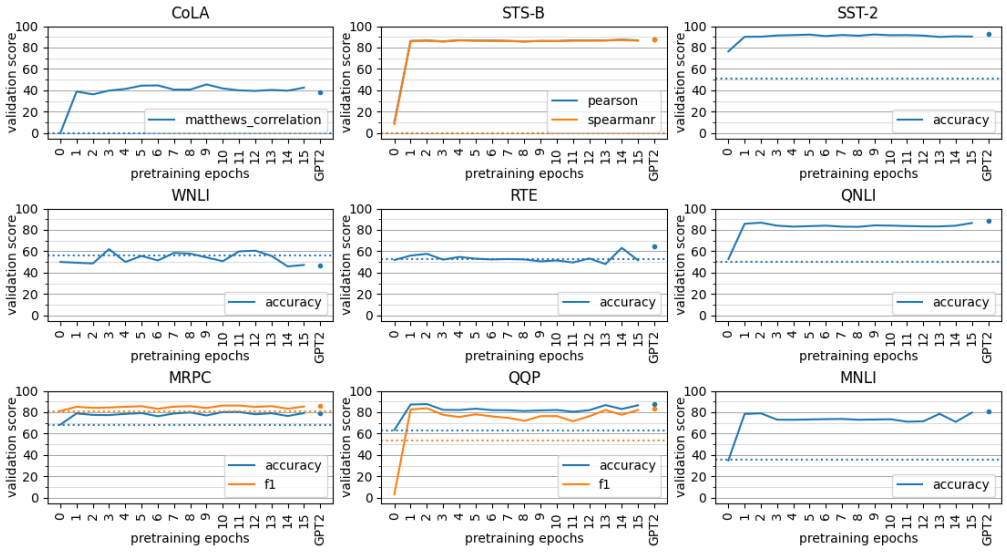
As it can be seen in Figure 5.1, the small model shows nice convergence behaviour. The losses start high but quickly fall to around 3.4 over the course of the first epoch. After 15 epochs, the train loss settles at around 3.1 while the validation loss converges to just over 3.2. The train loss starts to get ahead of the validation loss, but both losses decrease almost monotonically. Over the course of the pretraining, the distance between train and validation loss becomes gradually larger. At the end of the pretraining, the difference between the two losses is just over one.

Large model

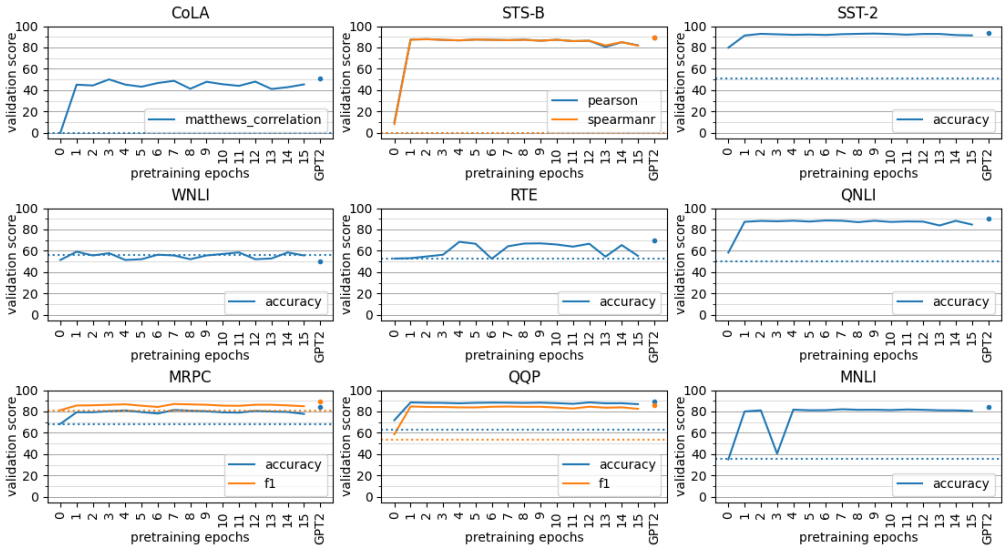
The large model also appears to converge until around six epochs or 200.000 trained batches. After the first epoch, the train loss reaches 3.1 and the validation loss 3.2. At six epochs, the validation loss reaches it's lowest at 3.1, while the train loss is slightly above 2.8. With further training, signs of overfitting start to show. The validation loss increases again, while the train loss keeps decreasing. At 15 epochs, the validation loss is equal to that of the small model at 3.2 and the train loss decreases to approximately 2.7. Looking at this trend, we hypothesize that with more training, the discrepancy between train loss and validation loss will increase further.
The observed behaviour occurs due to the large model having more than three times as many learnable parameters as the small model while the size of the dataset stays the same. Usually a neural network tries to find parameter values that work well on all samples of a dataset[1]. With more parameters, a neural network has more degrees of freedom to memorize the individual samples of the training dataset. If there are too many parameters in relation to the dataset size, the network tends to use the parameters to simply memorize all samples in the training dataset. This behaviour is called overfitting. It results in very good training losses, but poor generalization and therefore low performance on unseen data such as the validation set, very similar to what we observe in Figure 5.1.

[1] Focussing parameters only on one sample would make the results on other samples worse.

5.1.2 Finetuning results



(a)



(b)

Figure 5.2: The scores of (a) the small model and (b) the large model on all nine tasks. The x-axis describes the amount of pretraining that the model received. The y-axis shows the reached validation scores on the respective metrics. GPT2 is also finetuned in it's versions "small" for (a) and "medium" for (b) and is shown in the plots as a single scatter point. These results are not the actual achieved points on the benchmark as they are calculated over each task's validation set and not the test sets. The dotted lines mark the scores reached by our majority guessing baseline on the validation sets, taken from Table 4.4.

Small model

During pretraining, model checkpoints are saved after each epoch. All of them are finetuned on all nine tasks of the GLUE benchmark. The checkpoints allow to analyze how the pretraining impacts the results of the models after finetuning. To provide an upper bound regarding the amount of pretraining, the GPT2 model from Radford et al. [2019] is also finetuned with the same setup. The already pretrained versions "small" and "medium" of GPT2 are used to ensure comparability between models of similar size. Figure 5.2 gives an overview over the reached scores in relation to the amount of pretraining. The shown values are not the actual GLUE scores, but evaluations on the validation sets, assuming that the actual scores are similar. These results should therefore not be compared to actual GLUE scores of other models. Also, the tasks should not be compared to each other either, as they are evaluated using different metrics. For example, CoLA generally reaches lower scores than SST2. This is typical behaviour[2], as CoLA uses the matthews correlation coefficient as a metric, while SST2 is evaluated on accuracy.

An interesting aspect that can be observed on most tasks is the noticeable increase between zero and one epoch of pretraining. The initial jump is more pronounced on the tasks STS-B and CoLA, which is likely due to the used metrics, but can be seen to some extent in seven out of the nine tasks. Only WNLI and RTE do not show any increase with one epoch of pretraining. As WNLI has an extremely small training set of only 634 samples, a possible explanation is that none of the models are able to learn the task, regardless of the amount of pretraining.

After that initial jump, the scores remain relatively constant and show only slight growth on some of the tasks. Even the GPT2-small model only marginally improves on our scores. On some tasks, it actually performs worse than our model. Despite our stabilization approach of selecting the best model from multiple tries, some curves appear to fluctuate slightly. The tasks STS-B and SST2 appear to be most stable.

The most instable task seems to be WNLI. This is probably due to the fact that the test dataset only contains 146 samples. Due to the small size of the test set, the scores of WNLI are subject to chance[3]. We encountered strong fluctuations of the score between multiple finetuning runs. Even with the stabilization approach of selecting the best run of 10 tries, the scores remain unstable. The achieved results do not exceed simple majority guessing.

This instable behaviour becomes even more evident when looking at the results of WNLI in Table 5.2, which shows the baseline models proposed by Wang et al. [2018]. Because none of those baseline models manages to perform better than simple most frequent class guessing, the authors decided to substitute the WNLI scores of all GLUE baseline models for the majority baseline with a score of 65.1 instead. Radford et al. [2018] go even further and do not include WNLI in their reports at all.

The instability of further tasks is examined in Section 5.2.

On the tasks QNLI, QQP and MNLI, the small model's scores initially reach relatively high values after one and two epochs, before slightly declining with more epochs. Towards fifteen epochs of pretraining, the scores increase to a level similar to that of the first two epochs.

[2] See other models at https://gluebenchmark.com/leaderboard for comparison.

[3] A single wrong classification on WNLI decreases the score by around 0.7 points.

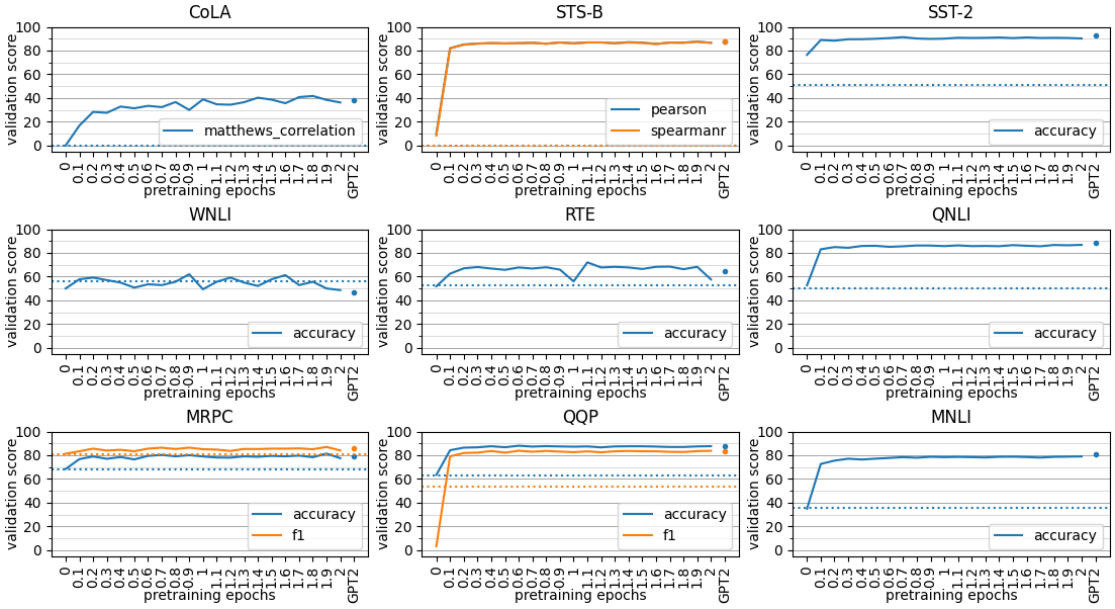


Figure 5.3: A more detailed overview over the validation scores reached by the small model between zero and two epochs of pretraining. Checkpoints are saved every tenth of an epoch and finetuning is performed with the exact same setup as before. The dotted lines show the results of our majority guessing baseline. The single scatter point marks the score of GPT2 in it's small version.

To investigate the initial jump between zero and one epoch further, we repeat the first two epochs of pretraining on the small model to generate additional checkpoints. The model is saved every tenth of an epoch, creating 20 checkpoints ranging from zero to two epochs of pretraining. We finetune these checkpoints on all nine tasks of the GLUE benchmark, using the same finetuning setup as before. The resulting scores can be seen in Figure 5.3. Looking at the more detailed scores, initial increases can be observed, similar to the ones in Figure 5.2. While most tasks show a significant jump from zero to a tenth of an epoch and only a small growth thereafter, other tasks like CoLA or RTE rise more evenly over the first few checkpoints. Apart from this, the general trend is similar to that shown in Figure 5.2a.

Large model

The plots in Figure 5.2b show the reached scores on the validation sets for all checkpoints of the large model. Just like the small model, a remarkable jump occurs between zero and one epoch of pretraining. In general, the curves are very similar. The observed instabilities occur on the same tasks as on the small model. The task MNLI shows a significant drop at epoch three, falling back to the performance of majority guessing.

Another very interesting aspect is, that the overfitting in the pretraining, starting around epoch six, is reflected in some of the curves. Especially on the tasks with a more stable behaviour, small inconsistencies start to show in the higher epochs. On many of the tasks the score starts to decrease and the curve becomes unstable towards the end. This behaviour can be best observed on the STS-B task.

The scores achieved by the GPT2-medium model appear to improve on our large model, however they are only slightly better than the best scores that are reached at around six epochs, right before the overfitting becomes evident. Ignoring the decrease caused by the overfitting, the GPT2 model shows only minimal improvement over our model, just like with our small model.

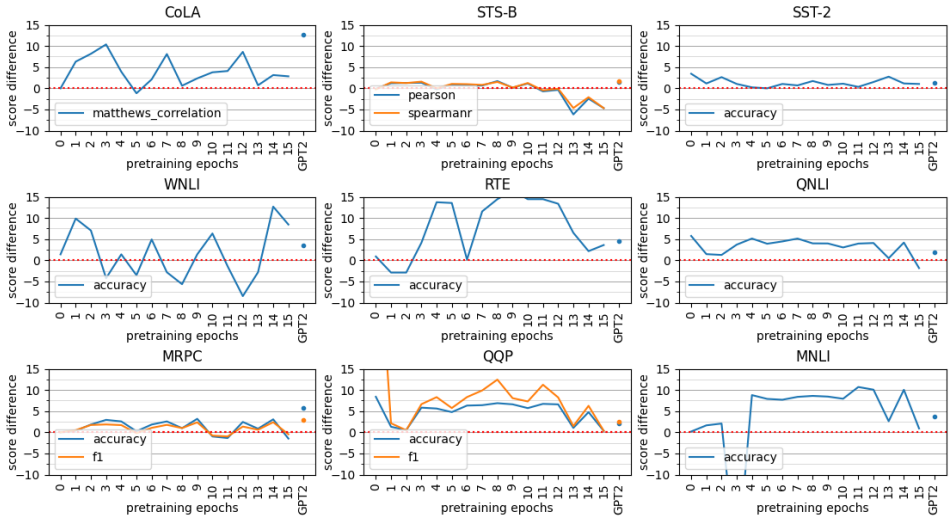


Figure 5.4: The difference in validation scores between the large and small model in relation to the number of pretraining epochs. All values above the red dotted line correspond to the large model being better and all values below correspond to the small model being better. The differences between GPT2-medium and GPT2-small are shown as scatter points.

A difference between the two models can be noticed that had already been hinted at in the pretraining losses. Comparing the validation scores of the small and the large model, the large model generally reaches slightly higher scores, even when looking at the late epochs with overfitting. This can be seen when looking at Figure 5.4, which shows the result of subtracting the small model's scores from the scores of the large model. This seems logical as a larger model is expected to perform better and shows that an improvement in scores can be made by increasing the model's size without changing anything in the training setup. However, the score increase is very small and is not proportional to the vast scale-up of the model. Combined with the signs of overfitting, this suggests that saturation has been reached in the current setup using a 5GB pretraining dataset. Selecting a larger subset of OpenWebText might therefore increase the performance even further.

5.2 Instability on Some Tasks

As Figure 5.2 shows, the tasks CoLA, RTE and WNLI are remarkably unstable. For WNLI this can be explained with the small datasets for both training and testing. However, this is no viable explanation for the other tasks, as they all have larger datasets.
Dodge et al. [2020] analyse the tasks MRPC, RTE, CoLA and SST-2 regarding their results in dependence on different random seeds. More precisely, they look at the seeds that control the order of the training data and the initialisation of the weights. Those two seeds are the only hyperparameters that are changed while performing multiple runs. It shows that some seed combinations produce good finetuning results, while others perform worse or do not even converge at all.

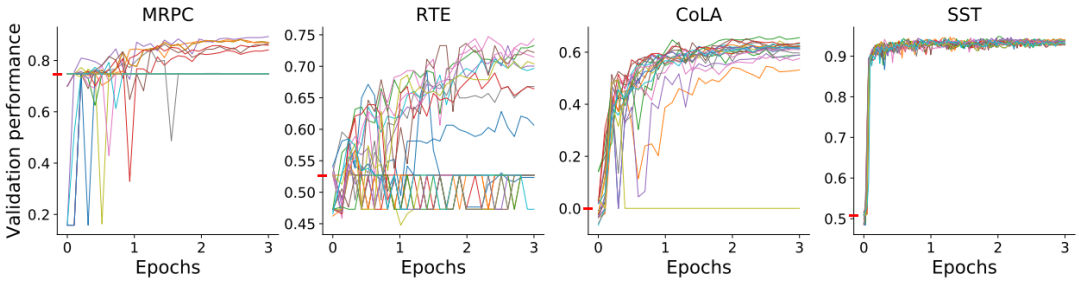


Figure 5.5: Training curves for 20 random seed combinations on MRPC, RTE, CoLA and SST-2. All scores are based on the validation sets and not the actual test sets. The usual metrics are used, for MRPC the validation performance is the average over accuracy and F1-score. While SST-2 shows a very similar progression over all runs, variations can be observed on the other three tasks. RTE and CoLA show runs that do not converge. The majority baseline scores are marked in red on the y-axis. [Dodge et al., 2020, Figure 4]

Figure 5.5 gives an overview over the training behaviour over multiple runs. Quite noticably, SST-2 appears to be very stable, which is consistent with our own observations. The other tasks show more unstable behaviour, with MRPC appearing to be the second most stable. Although not immediately obvious due to the different scaling of the y-axis, the good runs of RTE and CoLA have roughly a similar variance.
MRPC, RTE and CoLA all show some runs that do not converge at all and group around the majority guessing baselines. On CoLA, we also encountered these rare cases of total collapse, in which a model would suddenly reach only a minimal score. Figure 5.6 shows finetuning results on CoLA without the use of stabilization. A noticable drop in performance at epoch 13 underlines the non-converging behaviour. Our stabilization approach of performing multiple finetuning runs and selecting the best performing model manages to reliably sort out those bad models. Because MNLI has the largest training dataset with almost 400.000 samples, we cannot employ our stabilization on that task. Therefore we cannot rule out these non-converging runs, as it can be seen in Figure 5.2b at epoch three.
Figure 5.7 shows density estimations on the four tasks for the best and worst seeds. Each task's majority guessing baseline is marked in red on the x axis. As expected, SST2 has very low variances and the means of all seeds lie close together. MRPC and RTE show pronounced bimodal shapes. As MRPC is imbalanced with 68% positive samples, the region around 0.75 is only as good as majority guessing. As the RTE dataset is balanced, the observed cluster around 0.54 is again not better than the majority baseline.

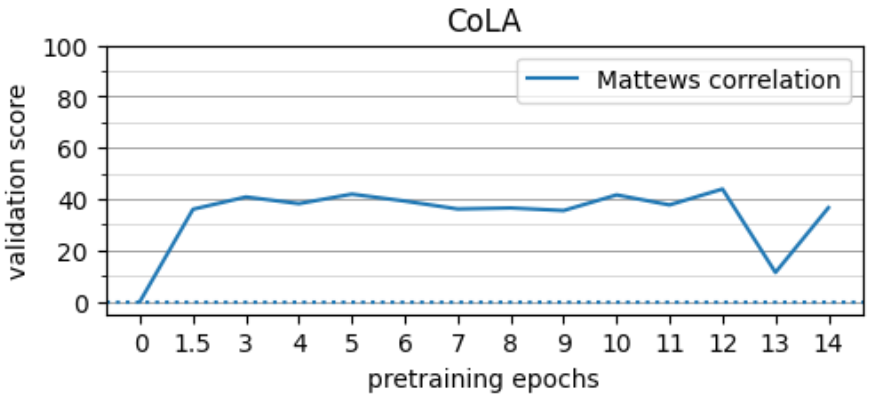


Figure 5.6: Validation scores of the small model on the task CoLA without our stabilization approach. The finetuning run of the checkpoint at epoch 13 does not converge properly and reaches only around 10 points. The dotted line marks the performance of our majority guessing baseline.

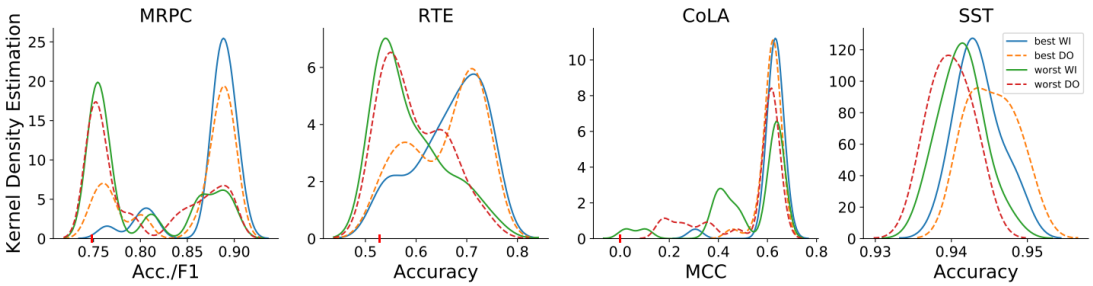


Figure 5.7: The plots show the kernel density estimations on the validation sets of four GLUE tasks. The majority baseline scores for MRPC, RTE and CoLA are marked in red on the x-axis. The baseline score of SST-2 is 50.9 and can therefore not be displayed. The best and worst weight initialisation seeds (WI) as well as the best and worst data order seeds (DO) are plotted. For MRPC the average between both metrics is used. It should be noted that the scales vary substantially from plot to plot. [Dodge et al., 2020, Figure 3]

CoLA has a distinct mode, but also shows concentrations at very low scores. The high peak still has a considerable variance but hardly differs between best and worst seeds. The worst seeds produce smaller accumulations at 0.4, 0.2 and 0.0. CoLA suffers from the largest distance between accumulation areas, which underlines how bad ”unlucky” runs can perform.

5.3 Comparison to Other Models

This section moves the focus towards the actual scores of our model and how they compare to other models. The scores on the test datasets are obtained by submitting to the GLUE benchmark's website. To obtain scores for our small model, we use it's checkpoint after 15 epochs of pretraining, as it receives the most training. Because our large model shows signs of overfitting, we use an earlier checkpoint at six epochs of pretraining. This checkpoint is chosen to prevent any impact of overfitting on the results. The achieved results of both models on the benchmark can be found in Table 5.1.

Table 5.1: The results of our models on the GLUE benchmark in comparison to the best scores of the GLUE baseline models and our two simple baselines, evaluated on the test sets, as described in Table 4.5. The small model is finetuned after 15 epochs of pretraining, the large model after six. The best score of a task is printed in bold.

Model	**Total**	**CoLA** (mcc)	**SST-2** (acc)	**MRPC** (F1/acc)	**STS-B** (pc/sc)	**QQP** (F1/acc)	**MNLI** (m-acc/mm-acc)	**QNLI** (acc)	**RTE** (acc)	**WNLI** (acc)
Small model	71.7	**41.2**	87.5	83.6/77.4	80.9/79.5	66.3/86.0	78.6/79.1	86.1	49.6	**65.1**
Large model	**73.3**	40.8	**91.4**	**85.1/79.9**	**83.1/81.8**	**68.6/87.4**	**81.3/81.6**	**88.1**	50.3	**65.1**
GLUE baseline	71.5	35.0	90.4	84.7/78.0	79.3/79.2	66.1/86.5	76.9/76.7	79.8	**59.2**	**65.1**
Majority guessing	40.5	0	49.9	79.9 / 66.5	-/-	29.9 / 82.4	35.6 / 36.5	49.5	49.7	**65.1**
Weighted guessing	37.3	0	50	73.1 / 56.1	0 / 0	13.6 / 58.6	33.4 / 33.4	50	50	51.9

Just like with the validation scores, our large model surpasses the small model on the test set, reaching 1.6 points more on average. Apart from CoLA and WNLI, the large model performs better on all tasks. On CoLA, the small model performs 0.4 points better. For WNLI, both models reach the same score of 65.1 which is the exact score of the majority guessing baseline on the test set. In fact both models constantly predict the negative class on the test set, which explains why the scores exactly match the majority baseline. It seems that the models learned majority guessing behaviour during training[4]. Apart from WNLI and RTE our models show significant improvements over our majority guessing and weighted guessing baselines. Our models tie with the majority guessing baseline on WNLI and differ only very little from both baselines on RTE.

5.3.1 GLUE baseline models

Along the GLUE benchmark itself, Wang et al. [2018] propose their own baseline in form of multiple models. The baselines can be divided into three approaches. With single-task training, every model is trained independently for every task. In Multi-task training, only the classifier components are trained separately, while everything else is shared across tasks. The third approach uses pretrained sentence representation models, on top of which task-specific classifiers are trained separately for each task. Table 5.2 shows an excerpt of all their models that contribute to the highest scores on a task.
The model setup reaching the highest average score in both single- and multi-task training is a model using a combination of two bidirectional long short-term memory networks (BiLSTM)[Graves and Schmidhuber, 2005] and an attention mechanism[5] in conjunction with a pretrained ELMo word embedding model [Peters et al., 2018]. On multi-task training it reaches the overall best score of 70.0.

[4] We did not influence the models in any way to make them behave like our majority guessing baseline.
[5] This attention mechanism differs from the one used in this work.

Table 5.2: The results of the GLUE baseline models. The table contains an excerpt from Wang et al. [2018, Table 4] showing only the models that contribute to the highest scores. The highest score on a task is printed in bold.

		Single Sentence		Similarity and Paraphrase			Natural Language Inference			
Model	**Avg**	**CoLA**	**SST-2**	**MRPC**	**QQP**	**STS-B**	**MNLI**	**QNLI**	**RTE**	**WNLI**
Single-Task Training										
BiLSTM+Attn,ELMo	66.5	**35**	90.2	68.8/80.2	**86.5/66.1**	55.5/52.5	**76.9/76.7**	76.7	50.4	**65.1**
Multi-Task Training										
BiLSTM+ELMo	67.7	32.1	89.3	**78.0/84.7**	82.6/61.1	67.2/67.9	70.3/67.8	75.5	57.4	**65.1**
BiLSTM+Attn,ELMo	**70.0**	33.6	**90.4**	**78.0**/84.4	84.4/63.1	74.2/72.3	74.1/74.5	**79.8**	58.9	**65.1**
Pre-Trained Sentence Representation Models										
GenSen [Subramanian et al., 2018]	66.2	7.7	83.1	76.6/83.0	82.9/59.8	**79.3/79.2**	71.4/71.3	78.6	59.2	**65.1**

On the task WNLI, no baseline model is capable of exceeding the majority guessing baseline of 65.1, which is why Wang et al. [2018] substitutes the models for most-frequent-class guessing.
It is notable that none of the baseline models is the best across all tasks. A model might reach highscores on certain tasks, but gets surpassed by other versions on different tasks. Therefore we form a reference by always choosing the result of the best performing model for each task and recalculating the average score based on these numbers. The best scores are collected in Table 5.1.

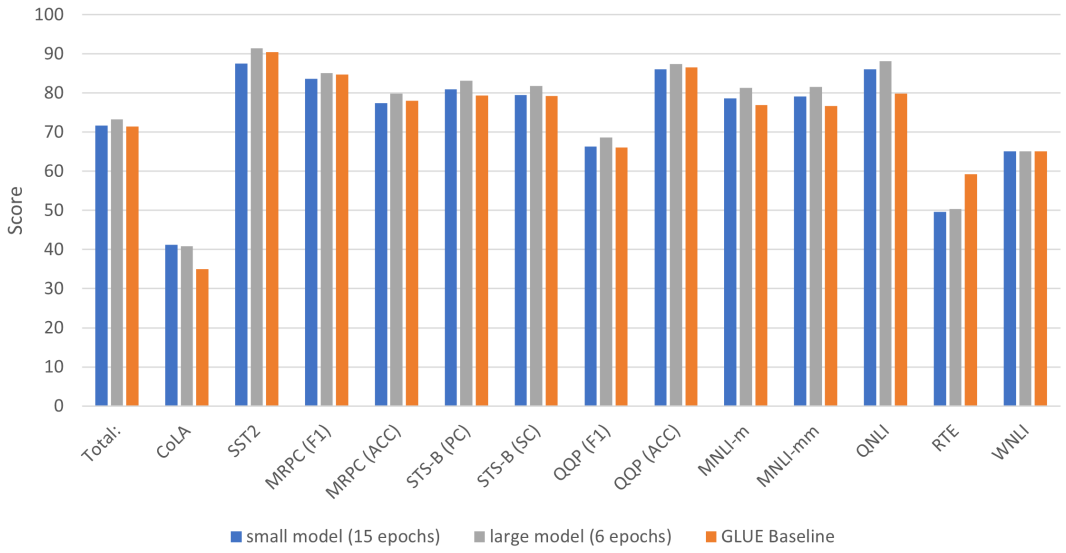


Figure 5.8: A comparison of our models to the best results of the GLUE baseline models

While our small model reaches a slightly higher average score than the GLUE baselines, it does not outperform them as clearly when looking at the individual tasks. The small model is only better at about half of the tasks, with the biggest improvement on CoLA. At RTE it performs significantly worse and it ties with the GLUE baseline at the WNLI test.
Our large model improves on our small model at almost every task and is only inferior on CoLA. It beats the GLUE baselines on all tasks except from RTE and WNLI, and improves on the baseline by 1.8 points on average.

5.3.2 GPT1

Since our models follow the GPT architecture first proposed by Radford et al. [2018], we compare our results to the performance of GPT1. Radford et al. [2018] report only the reached F1-scores on MRPC and QQP and do not include WNLI in their results. All known GLUE scores of GPT1 are collected in Table 5.3.

Table 5.3: The achieved scores of GPT1 on the GLUE tasks. Not all results are published by Radford et al. [2018, Tables 2, 4]. Our models are also listed for comparison. The highest score on a task is printed in bold.

Model	**Avg**	**CoLA** (mcc)	**SST-2** (acc)	**MRPC** (F1/acc)	**STS-B** (pc/sc)	**QQP** (F1/acc)	**MNLI** (m-acc/mm-acc)	**QNLI** (acc)	**RTE** (acc)	**WNLI** (acc)
GPT1	72.8	**45.4**	91.3	82.3 / -	82.0/-	**70.3**/-	**82.1**/81.4	**88.1**	**56.0**	-
Small model	71.7	41.2	87.5	83.6/77.4	80.9/79.5	66.3/86.0	78.6/79.1	86.1	49.6	**65.1**
Large model	**73.3**	40.8	**91.4**	**85.1**/**79.9**	**83.1**/**81.8**	68.6/**87.4**	81.3/**81.6**	**88.1**	50.3	**65.1**

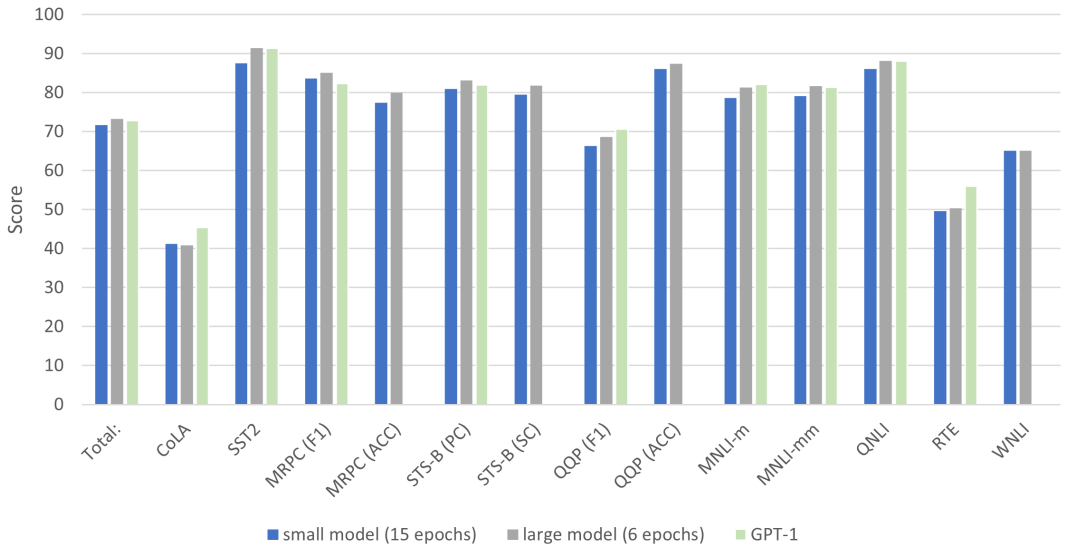


Figure 5.9: A comparison of our small and large models to the results of GPT1. Not all tasks and scores of the GLUE benchmark are published by Radford et al. [2018].

Comparing our models to the results of GPT1, it becomes evident that our small model performs worse on almost all tasks. Only at MRPC, the small model reaches a higher F1-score. This becomes evident when looking at the total score, where our small model reaches 1.1 points less than GPT1.
Our large model performs better, as it beats GPT1 by 0.5 points on the total GLUE score and is only remarkably worse on CoLA and RTE. On the other tasks the two models behave very similarly, with our model reaching slightly higher scores on the tasks SST2, MRPC and STS-B. For MNLI, GPT1 performs better on the matched test set, while our large model is better on the mismatched test set. On QNLI they both reach the exact same score of 88.1 points.

5.3.3 Human performance

The GLUE benchmark is mainly used to compare AI models to each other using typical tasks in the field of natural language processing. People use natural language on a daily basis to such an extent that performing most of these NLP tasks isn't even perceived as a challenge.
When talking about artificial intelligence, we always think about machines that come close to human behaviour and the human understanding of intelligence. Alan Turing's imitation game is just one famous representative of this narrative.
It therefore makes sense to compare our models' performances to the results of humans. Nangia and Bowman [2019] estimate the human performance on the GLUE benchmark by randomly using 500[6] samples from the training set of each task and letting 5 human annotators label them independently. The label selected by the most annotators is chosen as the prediction. A short training phase with 20 random examples from the validation sets is used to familiarise the annotators with the task.

Table 5.4: The scores on all nine GLUE tasks reached by humans. All scores are reported on the same metrics as in the GLUE paper. The results are taken from Nangia and Bowman [2019, Table 1]. Our models are added for comparison. The highest score on a task is printed in bold.

Model	**Avg**	**CoLA** (mcc)	**SST-2** (acc)	**MRPC** (F1/acc)	**STS-B** (pc/sc)	**QQP** (F1/acc)	**MNLI** (m-acc/mm-acc)	**QNLI** (acc)	**RTE** (acc)	**WNLI** (acc)
Human	**87.1**	**66.4**	**97.8**	**86.3/80.8**	**92.7/92.6**	59.5/80.4	**92.0/92.8**	**91.2**	**93.6**	**95.9**
Small model	71.7	41.2	87.5	83.6/77.4	80.9/79.5	66.3/86.0	78.6/79.1	86.1	49.6	65.1
Large model	73.3	40.8	91.4	85.1/79.9	83.1/81.8	**68.6/87.4**	81.3/81.6	88.1	50.3	65.1

As can be seen in Table 5.4 the scores achieved by humans are quite high. The human performance on the GLUE benchmark can generally be considered good, as expected. Nonetheless it does not reach perfect scores. On certain tasks the score comes close to 100, but is substantially lower on other tasks. The highest score is 97.8 on SST-2 and the lowest is 66.4 on the CoLA benchmark[7]. The fact that the human results are not perfect is not unusual, as it is part of human nature to make mistakes. Also, different regions, age groups or cultures might have different understandings of certain things. At last, the correct answer might not always be obvious, considering the short training phase with only 20 samples. All this leads to human scores that are high, but not perfect.
Comparing our models to the human performance, it becomes obvious that they are nowhere near capable of beating humans. Especially the tasks RTE and WNLI, where our models cannot even outperform the majority guessing baseline, show that humans are still much better at common sense reasoning and quickly adapting to new tasks. Nevertheless it is impressive to see that our models are only slightly behind the human performance on tasks such as MRPC or QNLI, and can even outperform it at QQP.

[6] On WNLI 145 samples are used

[7] The tasks use different metrics and are not directly comparable. However all Metrics can reach a maximum score of 100 if all outputs are labeled correctly.

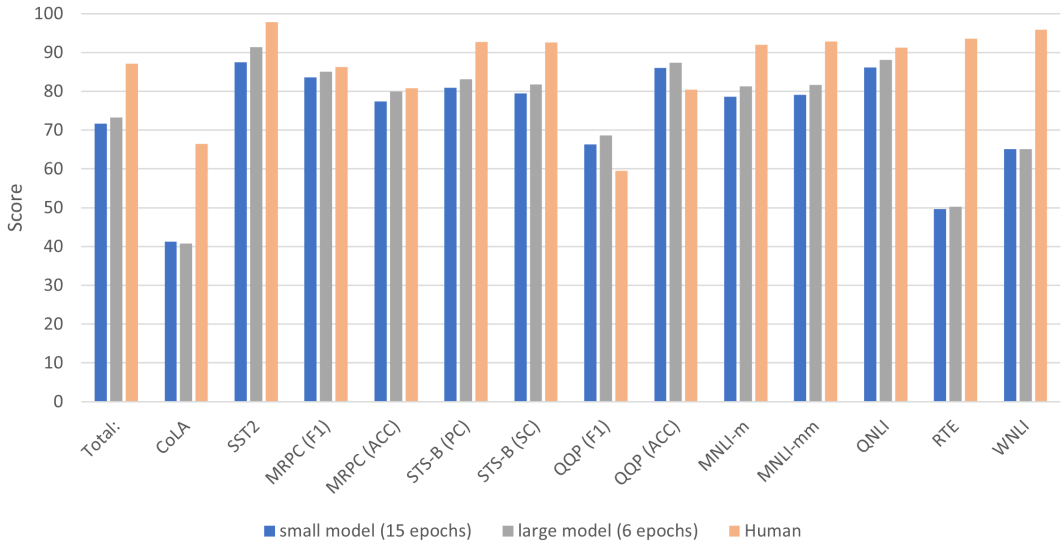


Figure 5.10: A per-task comparison of our models with the human performance stated in [Nangia and Bowman, 2019, Table 1]. Although the human scores easily outperform our models, it is notable, that our models come close to the human scores on some of the tasks, and can even surpass them on QQP.

6 Conclusion

A Generative Pretrained Transformer (GPT) is a derivative of a transformer that omits the encoder stack. It is a generative model used in the field of natural language processing and excels at sequence completion tasks. Relying on a mechanism called self-attention, the GPT can understand complex relations between words in the input sequence and ignore irrelevant parts. This self-attention is performed multiple times in parallel in each block and multiple blocks are chained together to form the decoder stack. An output head creates the desired outputs based on the representations created in the decoder stack. It usually acts as a simple classifier, but can take different forms depending on the task. An extensive pretraining is performed on unlabelled text corpora in a self-supervised manner. This pretraining enables the GPT to auto-regressively generate text by repeatedly predicting a token following the input. If a GPT needs to be adapted to a new domain, or even take on a completely different task, it can be finetuned to suit the new setting with comparably little effort. Doing so, the GPT excels at transferring already learned knowledge about natural language to new tasks, which is especially useful in situations where only little training data exists.

We train two models of different size. The small model is equivalent to GPT2-small with 12 attention heads per block and 12 consecutive blocks. Our large model has around three times the size and is comparable to GPT2-medium, employing 16 attention heads in each of the 24 blocks. Pretraining is done on a 5GB subset of OpenWebText for 15 epochs with half-overlapping samples. While our small model converges nicely, our large model shows signs of overfitting that begin after around 6 epochs of pretraining. The GLUE benchmark is used to evaluate both models by measuring scores on nine different tasks. The models are adapted to each specific task by applying a short finetuning, which uses the supplied training datasets. We can compare the impact that pretraining has on the final benchmark scores by evaluating model checkpoints after each epoch of pretraining. It shows that a short pretraining leads to a significant improvement in scores compared to a model without pretraining. With more pretraining, this impact reduces to a level where the random fluctuations in performance hide any benefit of further pretraining. Looking at the behaviour of the large model, it is clear that overfitting in pretraining does hurt performance after finetuning, which, although expected, provides evidence that finetuning still depends on the success of pretraining for more than a few epochs.

It is noticeable that our models show very unstable convergence behaviour on some of the tasks. By performing multiple finetuning runs and selecting the model that reaches the best scores on the validation set, we can reduce the fluctuations considerably.

Not only do our models beat our own simple baselines on most of the tasks, they are competitive to the models provided as a baseline by the authors of GLUE. While our small model yields comparable results, our large model improves upon that baseline in most of the tasks. It is capable of beating GPT1 by 0.5 points on GLUE's total score. Comparing our models with human performance ultimately shows their limitations. Even though our models outperform human results on QQP and come close to it on some other tasks, they perform significantly worse on the other categories.

Bibliography

Ashutosh Adhikari, Achyudh Ram, Raphael Tang, and Jimmy Lin. Docbert: BERT for document classification. *CoRR*, abs/1904.08398, 2019. URL http://arxiv.org/abs/1904.08398.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

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

Pierre Baldi, Søren Brunak, Yves Chauvin, Claus AF Andersen, and Henrik Nielsen. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, 16(5):412–424, 2000.

Luisa Bentivogli, Bernardo Magnini, Ido Dagan, Hoa Trang Dang, and Danilo Giampiccolo. The fifth PASCAL recognizing textual entailment challenge. In *Proceedings of the Second Text Analysis Conference, TAC 2009, Gaithersburg, Maryland, USA, November 16-17, 2009*. NIST, 2009. URL https://tac.nist.gov/publications/2009/additional.papers/RTE5_overview.proceedings.pdf.

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

Ciprian Chelba, Tomás Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, and Phillipp Koehn. One billion word benchmark for measuring progress in statistical language modeling. *CoRR*, abs/1312.3005, 2013. URL http://arxiv.org/abs/1312.3005.

Jieneng Chen, Yongyi Lu, Qihang Yu, Xiangde Luo, Ehsan Adeli, Yan Wang, Le Lu, Alan L. Yuille, and Yuyin Zhou. Transunet: Transformers make strong encoders for medical image segmentation. *CoRR*, abs/2102.04306, 2021. URL https://arxiv.org/abs/2102.04306.

Nancy Chinchor. Muc-4 evaluation metrics. In *Proceedings of the 4th Conference on Message Understanding*, MUC4 '92, page 22–29, USA, 1992. Association for Computational Linguistics. ISBN 1558602739. doi: 10.3115/1072064.1072067. URL https://doi.org/10.3115/1072064.1072067.

Council of European Union. Regulation (eu) 2021/694 of the european parliament and of the council of 29 april 2021 establishing the digital europe programme and repealing decision (eu) 2015/2240 (text with eea relevance), 2023. URL https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A02021R0694-20230921.

Ido Dagan, Oren Glickman, and Bernardo Magnini. The pascal recognising textual entailment challenge. In Joaquin Quiñonero-Candela, Ido Dagan, Bernardo Magnini, and Florence d'Alché Buc, editors, *Machine Learning Challenges. Evaluating Predictive Uncertainty, Visual Object Classification, and Recognising Tectual Entailment*, pages 177–190, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33428-6.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah Smith. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping, 2020.

Bill Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Third International Workshop on Paraphrasing (IWP2005)*. Asia Federation of Natural Language Processing, January 2005. URL https://www.microsoft.com/en-us/research/publication/automatically-constructing-a-corpus-of-sentential-paraphrases/.

David Freedman, Robert Pisani, and Roger Purves. Statistics (international student edition). *Pisani, R. Purves, 4th edn. WW Norton & Company, New York*, 2007.

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

Danilo Giampiccolo, Bernardo Magnini, Ido Dagan, and Bill Dolan. The third PASCAL recognizing textual entailment challenge. In Satoshi Sekine, Kentaro Inui, Ido Dagan, Bill Dolan, Danilo Giampiccolo, and Bernardo Magnini, editors, *Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing*, pages 1–9, Prague, June 2007. Association for Computational Linguistics. URL https://aclanthology.org/W07-1401.

Aaron Gokaslan, Vanya Cohen, Ellie Pavlick, and Stefanie Tellex. Openwebtext corpus, 2019. URL http://Skylion007.github.io/OpenWebTextCorpus.

Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm networks. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 4, pages 2047–2052. IEEE, 2005.

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*, volume 7, pages 785–794, 2006.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015b. URL http://arxiv.org/abs/1512.03385.

Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016. URL http://arxiv.org/abs/1606.08415.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. *CoRR*, abs/1804.10959, 2018. URL http://arxiv.org/abs/1804.10959.

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *CoRR*, abs/1808.06226, 2018. URL http://arxiv.org/abs/1808.06226.

Kai Labusch, Preußischer Kulturbesitz, Clemens Neudecker, and David Zellhöfer. Bert for named entity recognition in contemporary and historical german. In *Proceedings of the 15th conference on natural language processing*, pages 9–11, 2019.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Hector Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. In *Thirteenth international conference on the principles of knowledge representation and reasoning*, 2012.

Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017. URL http://arxiv.org/abs/1711.05101.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015. URL http://arxiv.org/abs/1508.04025.

B.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. ISSN 0005-2795. doi: https://doi.org/10.1016/0005-2795(75)90109-9. URL https://www.sciencedirect.com/science/article/pii/0005279575901099.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *CoRR*, abs/1710.03740, 2017. URL http://arxiv.org/abs/1710.03740.

Nikita Nangia and Samuel R. Bowman. Human vs. muppet: A conservative estimate of human performance on the GLUE benchmark. *CoRR*, abs/1905.10425, 2019. URL http://arxiv.org/abs/1905.10425.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. *CoRR*, abs/1606.06031, 2016. URL http://arxiv.org/abs/1606.06031.

Matthew E. Peters and Dan Lecocq. Content extraction using diverse feature sets. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, page 89–90, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320382. doi: 10.1145/2487788.2487828. URL https://doi.org/10.1145/2487788.2487828.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018. URL http://arxiv.org/abs/1802.05365.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. *CoRR*, abs/1606.05250, 2016. URL http://arxiv.org/abs/1606.05250.

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

Christopher J. Shallue, Jaehoon Lee, Joseph M. Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. *CoRR*, abs/1811.03600, 2018. URL http://arxiv.org/abs/1811.03600.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In David Yarowsky, Timothy Baldwin, Anna Korhonen, Karen Livescu, and Steven Bethard, editors, *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, October 2013. Association for Computational Linguistics. URL https://aclanthology.org/D13-1170.

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.

The New York Times. Microsoft to invest $10 billion in openai, the creator of chatgpt, 2023. URL https://www.nytimes.com/2023/01/23/business/microsoft-chatgpt-artificial-intelligence.html. Accessed: 28.01.2024.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950. ISSN 0026-4423. doi: 10.1093/mind/LIX.236.433. URL https://doi.org/10.1093/mind/LIX.236.433.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

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. *CoRR*, abs/1804.07461, 2018. URL http://arxiv.org/abs/1804.07461.

Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. Neural Network Acceptability Judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 09 2019. ISSN 2307-387X. doi: 10.1162/tacl_a_00290. URL https://doi.org/10.1162/tacl_a_00290.

Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In Marilyn Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics:*

Human Language Technologies, Volume 1 (Long Papers), pages 1112–1122, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1101. URL https://aclanthology.org/N18-1101.

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

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture, 2020a.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 10524–10533. PMLR, 13–18 Jul 2020b. URL https://proceedings.mlr.press/v119/xiong20b.html.

Yinfei Yang, Steve Yuan, Daniel Cer, Sheng-yi Kong, Noah Constant, Petr Pilar, Heming Ge, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Learning semantic textual similarity from conversations. *arXiv preprint arXiv:1804.07754*, 2018.

Yukun Zhu, Ryan Kiros, Richard S. 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. *CoRR*, abs/1506.06724, 2015. URL http://arxiv.org/abs/1506.06724.

Eidesstattliche Versicherung

(Affidavit)

Richter, Simon Ian

Name, Vorname
(surname, first name)

225673

Matrikelnummer
(student ID number)

☒ Bachelorarbeit
(Bachelor's thesis)

☐ Masterarbeit
(Master's thesis)

Titel
(Title)

An Empirical Analysis of Self-built GPT Models for GLUE Task Performance

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Witten, 23.02.24

Ort, Datum
(place, date)

Unterschrift
(signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Witten, 23.02.24

Ort, Datum
(place, date)

Unterschrift
(signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**

7 Appendix

```
import pretrain
import finetune
import evaluate
import plots

############################
# This file provides premade setups for testing.
# All modules are designed to work on the same "models" and "tasks" lists, which configure the behaviour.
# However, every step can also be started manually.

# The provided setups are not guaranteed to work on every GPU.
# For example, a NVIDIA P100 does not support torch.compile(), which is why this feature has to be deactivated in the "models" setup (set 'compile_model':False)
# Also the GPU's VRAM is a limiting factor, while a NVIDIA A100 with 40GB can perform pretraining of our large model with only 4 accumulation steps,
# a P100 requires 16. For the small model, the P100 requires 8.

# To execute one of the setups, go to the end of this file, set "gpu_num" to select the desired GPU and uncomment one of the function calls
############################

def minimal_setup():
    models = [
        {'name':"min_test", 'max_epochs': 1, "embedding_dimension":128, "num_heads":4, "num_blocks":4, 'accumulation_steps':1, 'compile_model':True},
    ]

    tasks = [
        {'task_name': 'cola', 'epochs':2},
        {'task_name': 'stsb', 'epochs':2},
        {'task_name': 'sst2', 'epochs':2},
        {'task_name': 'wnli', 'epochs':4, 'dropout':0.2, 'eval_interval':5},
        {'task_name': 'rte'},
        {'task_name': 'qnli', 'epochs':0.6},
        {'task_name': 'mrpc'},
        {'task_name': 'qqp', 'epochs':0.3, 'eval_interval':500},
        {'task_name': 'mnli', 'epochs':0.3, 'eval_interval':500},
    ]

    # Pretrain all models
    print("-----starting pretraining-----")
    pretrain.pretrain(models, gpu_num, num_subsets=1)
    print("-----pretraining finished-----")
    # Plot pretraining loss curves
    plots.plot_pretraining(models, y_min=5, y_max=10)

    # Finetune all models on all tasks
    print("-----starting finetuning-----")
    finetune.finetune(models, tasks, gpu_num)
    print("-----finetuning finished-----")

    print("-----starting evaluation-----")
    # Evaluate the finetuned models on the test sets and create a zip
    evaluate.evaluate(models, tasks, gpu_num, select_epochs='last', validation_only=True)
    print("-----evaluation finished-----")
    # As we do not expect the results to be actually submitted to GLUE, this setup only reports validation results.
    # If you want a zip file for a submission, set 'validation_only' to False

def full_setup():
    models = [
```

```
        {'name':"large_model", 'max_epochs':15},
        {'name':"small_model", 'max_epochs': 15},
        {'name':"gpt2_small", 'max_epochs': 0},
        {'name':"gpt2_medium", 'max_epochs': 0},
    ]

    tasks = [
        {'task_name': 'cola', 'tries':7},
        {'task_name': 'stsb', 'tries':5},
        {'task_name': 'sst2', 'tries':2},
        {'task_name': 'wnli', 'epochs':6, 'tries':10, 'dropout':0.2, 'eval_interval':5},
        {'task_name': 'rte', 'tries':10},
        {'task_name': 'qnli', 'epochs':2},
        {'task_name': 'mrpc', 'tries':3},
        {'task_name': 'qqp', 'epochs':1, 'eval_interval':500},
        {'task_name': 'mnli', 'epochs':1, 'eval_interval':500},
    ]

    # Pretrain all models
    print("-----starting pretraining-----")
    pretrain.pretrain(models, gpu_num)
    print("-----pretraining finished-----")
    # Plot pretraining loss curves
    plots.plot_pretraining(models)
    # Finetune all models on all tasks
    print("-----starting finetuning-----")
    finetune.finetune(models, tasks, gpu_num, select_epochs='all')
    print("-----finetuning finished-----")

    print("-----starting evaluation-----")
    # Evaluate the finetuned models on the test sets and create a zip
    evaluate.evaluate(models, tasks, gpu_num, select_epochs='all')
    print("-----evaluation finished-----")
    print("-----plotting overview-----")
    plots.create_overview(models, tasks, add_gpt2=True, select_epochs='all', plot_differences=True)



# This is a setup, which should produce relatively good results but is still much faster than the full setup.
# Our stabilization approach is not active ('tries':1) and the results might therefore fluctuate considerably
    on some tasks
#
def small_setup():
    models = [
        {'name':"small_model", 'max_epochs': 1, "accumulation_steps":8, "compile_model":True},
    ]

    tasks = [
           {'task_name': 'cola'},
           {'task_name': 'stsb', 'epochs':2},
           {'task_name': 'sst2', 'epochs':2},
           {'task_name': 'wnli', 'epochs':4, 'dropout':0.2, 'eval_interval':5},
           {'task_name': 'rte'},
           {'task_name': 'qnli', 'epochs':1},
           {'task_name': 'mrpc'},
           {'task_name': 'qqp', 'epochs':0.5, 'eval_interval':500},
           {'task_name': 'mnli', 'epochs':0.5, 'eval_interval':500},
    ]

    # Pretrain all models
    print("-----starting pretraining-----")
    pretrain.pretrain(models, gpu_num, num_subsets=5)
    print("-----pretraining finished-----")
    # Plot pretraining loss curves
    plots.plot_pretraining(models, y_min=3.3, y_max=6.5)

    # Finetune all models on all tasks
    print("-----starting finetuning-----")
    finetune.finetune(models, tasks, gpu_num, select_epochs='all')
    print("-----finetuning finished-----")

```

```
    # Evaluate the finetuned models on the test sets and create a zip
    print("-----starting evaluation-----")
    evaluate.evaluate(models, tasks, gpu_num, select_epochs='all')
    print("-----evaluation finished-----")
    print("-----plotting overview-----")
    plots.create_overview(models, tasks, select_epochs='all')


gpu_num=0

# full_setup()
# minimal_setup()
# small_setup()
```

Listing 7.1: Main.py

```python
# This class holds variables used for creating the models
class params:
    def __init__(self, embedding_dimension, n_heads, n_blocks, batchsize, context_length, device, vocab_size,
          dropout, task="prediction", bias=True, use_gpt2=False, freeze_model=False,
          efficient_implementation=True, **kwargs):
        self.embedding_dimension = embedding_dimension
        self.num_heads = n_heads
        self.head_size = batchsize // n_heads
        self.n_blocks = n_blocks
        self.batchsize = batchsize
        self.context_length = context_length
        self.device = device
        self.vocab_size = vocab_size
        self.dropout = dropout
        self.task = task
        self.bias = bias
        self.use_gpt2 = use_gpt2
        self.freeze_model=freeze_model
        self.efficient_implementation=efficient_implementation

# These classes are presets for our small and large model sizes.
class small_model (params):
    def __init__(self, batchsize, context_length, device, dropout, task="prediction", bias=True,
          use_gpt2=False, freeze_model=False, efficient_implementation=True, **kwargs):
        params.__init__(self, embedding_dimension=768, n_heads=12, n_blocks=12, batchsize=batchsize,
              context_length=context_length, device=device, vocab_size=50257, dropout=dropout, task=task,
              bias=bias, use_gpt2=use_gpt2, freeze_model=freeze_model,
              efficient_implementation=efficient_implementation)

class large_model (params):
    def __init__(self, batchsize, context_length, device, dropout, task="prediction", bias=True,
          use_gpt2=False, freeze_model=False, efficient_implementation=True, **kwargs):
        params.__init__(self, embedding_dimension=1024, n_heads=16, n_blocks=24, batchsize=batchsize,
              context_length=context_length, device=device, vocab_size=50257, dropout=dropout, task=task,
              bias=bias, use_gpt2=use_gpt2, freeze_model=freeze_model,
              efficient_implementation=efficient_implementation)
```

Listing 7.2: config.py

```python
import torch
import torch.nn as nn
from torch.nn import functional as func
from transformers import GPT2Model, GPT2LMHeadModel, GPT2Config
import Model.Embeddings as Embeddings
import Model.config as config
import math
import os


class GptModel(nn.Module):

    def __init__(self, cfg, name):
        super().__init__()
        self.__use_gpt2 = cfg.use_gpt2
        self.freeze = cfg.freeze_model
        self.device = cfg.device
        self.T = cfg.context_length
        self.task = cfg.task
        self.name = name

        if self.__use_gpt2: # Load the complete GPT2 model instead of the own implementation. Only the output head is still created normally below.
            conf = GPT2Config(n_positions = cfg.context_length, n_embd = cfg.embedding_dimension, n_layer=cfg.n_blocks, n_head=cfg.num_heads, resid_pdrop=cfg.dropout, embd_pdrop=cfg.dropout, attn_pdrop=cfg.dropout) # Translates the config into a GPT2 config

            self.gpt2 = GPT2LMHeadModel(conf) if self.task == "pretraining" else GPT2LMHeadModel.from_pretrained(f"gpt2{'-medium' if cfg.embedding_dimension == 1024 else ''}")
            self.train()

        else:
            self.train()

            # Creates the token embedding matrix
            self.token_embedding = nn.Embedding(cfg.vocab_size, cfg.embedding_dimension)

            # If possible, loads the pretrained token embedding weights from gpt2 (small/medium). Note that huggingface calls the small model just "gpt2"
            if cfg.embedding_dimension in [768, 1024]:
                extension = '' if cfg.embedding_dimension==768 else '-medium' # No extension for loading GPT2-small, "-medium" extension for GPT2-medium
                path_to_weight = f"Model/token_embedding_weights{extension}.pt"
                if os.path.exists(path_to_weight):
                    tew = torch.load(path_to_weight)
                else:
                    print(f"Downloading token_embedding_weights from gpt2{extension}")
                    tew = GPT2Model.from_pretrained(f"gpt2{extension}").wte.weight
                    torch.save(tew, path_to_weight)

                # Overwrites the token embedding matrix with the loaded weights and freezes them (As they are already trained, we do not want to change them)
                self.token_embedding.weight = tew
                self.token_embedding.weight.requires_grad = False

            # Prepare a position embedding matrix capable of encoding cfg.context_length positions
            self.register_buffer('positional_embedding', Embeddings.get_positional_encoding(cfg))

            self.embedding_dropout = nn.Dropout(cfg.dropout)

            # The decoder stack consisting of multiple sequentially connected blocks
            self.blocks = nn.Sequential(*[Block(cfg) for _ in range(cfg.n_blocks)])

            # Since we use a pre-LN transformer (Layernorm in front of each block's sublayers instead of after), the output of the last decoder block has not been normalized. Therefore we apply this LayerNorm manually after the decoder stack
            self.norm = nn.LayerNorm(cfg.embedding_dimension)

        # Creates the output head depending on the task and the model setup
        if self.task == "pretraining":
            self.head = nn.Linear(cfg.embedding_dimension, cfg.vocab_size, bias=False)
```

```
63
64          # If the model is pretrained, the weights of the token embedding layer and the linear layer in the
                output head are tyed.
65          if self.__use_gpt2:
66              self.head.weight = self.gpt2.transformer.wte.weight
67          else:
68              self.head.weight = self.token_embedding.weight
69      else:
70          out_dim = 1 if self.task == "stsb" else 3 if (self.task =="ax" or "mnli" in self.task) else 2
71          self.ft_head = nn.Linear(cfg.embedding_dimension, out_dim) # Different Name is used for the head, so
                load_state_dict ignores the old language modelling head with (strict=False)
72          self.ft_head.weight = torch.nn.Parameter(self.ft_head.weight/2)
73          self.ft_head.bias = torch.nn.Parameter(self.ft_head.bias/2)
74
75
76  def forward(self, x):
77      if self.__use_gpt2:
78          x = self.gpt2(x, output_hidden_states=True).hidden_states[-1] # If gpt2 is used, forward it and get
                the output of the decoder stack (The GPT2 output head is ignored)
79      else:
80          if self.freeze: # Freezing the model means that only the output head should be trained. Therefore
                everything else is calculated with disabled gradients and the input is forwarded in eval mode
                (important for dropouts)
81              self.eval()
82          with torch.set_grad_enabled(not self.freeze):
83              x = self.token_embedding(x) # Token embedding
84              if x.shape[1] <= self.positional_embedding.shape[1]:
85                  x = x + self.positional_embedding[:, :x.shape[1], :] # Positional embedding
86              else:
87                  # If a sample is longer than the maximum context length, the fixed encoding formula is used
                        to temporarily extend the encoding to the desired length
88                  x = x + torch.cat((self.positional_embedding,
                        torch.FloatTensor([Embeddings.get_single_encoding(self.positional_embedding.shape[2],
                        i+self.positional_embedding.shape[1]) for i in
                        range(x.shape[1]-self.positional_embedding.shape[1])])[None, :, :].to(self.device)),
                        dim=1)
89              x = self.embedding_dropout(x)
90
91              # Forward the embedded input through the decoder stack (all blocks)
92              x = self.norm(self.blocks(x))
93          self.train() # In case, the model was frozen, train() ensures that the output head will still be
                trained
94
95      # Pretraining and finetuning heads are named differently (to make loading a checkpoint easier).
96      if self.task == "pretraining":
97          return self.head(x)
98      else:
99          out = self.ft_head(x[:, -1:, :]) # When Finetuning only the last position in context_dim is
                forwarded to the head, because it holds full information
100         return out
101
102
103 # This method allows to sample text from the GPT model.
104 # x is the input that will be continued by max_length new token.
105 # If x is/grows larger than the maximum sequence length of the GPT, only the last T_{max} token are taken
        into account.
106 def generate(self, x, max_length):
107     if not self.hasattr("head") or self.head.weight.shape[1]!=50257 or self.task != "pretraining":
108         print("Error: Output head is not suitable for text generation tasks")
109         return
110     for _ in range(max_length):
111         logits = self(x.view(1, -1)[:, -self.T:]) # Appends a batch dimension of size 1 for compatibility
                and crops the input to a max length of T. Then implicitly calls the forward function
112         last_logit = logits.view(-1, 50257)[-1,:] # Collapse the batch dimension and only use the last token
                position
113         prob = func.softmax(last_logit, dim=-1) # Softmax is applied, since it is not done in the output head
114         x_next = torch.multinomial(prob, num_samples=1) # Samples from the probability distribution.
                Alternatively, torch.argmax(probs) can be used
115         x = torch.cat((x, x_next)) # Concatenates the sampled token to the input to be used in the next
                iteration
116     return x
```

```

# This class defines a single decoder block in the decoder stack
class Block(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.sa = Multi_head_attention(cfg)
        self.norm1 = nn.LayerNorm(cfg.embedding_dimension)
        self.norm2 = nn.LayerNorm(cfg.embedding_dimension)
        # The feed-forward network is created as a torch.nn.Sequential layer
        self.ffwd = nn.Sequential(
            nn.Linear(cfg.embedding_dimension, 4 * cfg.embedding_dimension),
            nn.GELU(approximate='tanh'), # The tanh approximation of the GELU function is used to slightly speed up the calculations
            nn.Linear(4 * cfg.embedding_dimension, cfg.embedding_dimension),
            nn.Dropout(cfg.dropout)
        )
        # As described in "Language Models are Unsupervised Multitask Learners" all residual Layers (last layer before the residual path gets added)
        # are scaled by 1/sqrt(N) after initialization with stdv=0.02 (see "Improving Language Understanding by Generative Pre-Training")
        # since there are 2 residual connections per block, the number of residual layers N equals 2*cfg.n_blocks
        torch.nn.init.normal_(self.ffwd[2].weight, 0, 0.02 / math.sqrt(2 * cfg.n_blocks))
        # The second residual layer in the block is implemented inside the Multi_head_attention class below

    def forward(self, x):
        # Note that LayerNorm is applied before attention or feed-forward. Therefore this is a Pre-LN Transformer
        x = x + self.sa(self.norm1(x))
        x = x + self.ffwd(self.norm2(x))
        return x

# This class implements the masked multi-head self-attention, used in the decoder block
class Multi_head_attention(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        T, C = cfg.context_length, cfg.embedding_dimension
        nh, hs = cfg.num_heads, C // cfg.num_heads
        self.n_embd = C
        self.n_head = nh
        self.scale_factor = 1 / math.sqrt(hs) # The scaling factor used in the attention
        self.dropout = cfg.dropout

        # The efficient implementation uses only a single linear layer to create Q, K and V matrices for all heads
        if cfg.efficient_implementation:
            self.c_attn = nn.Linear(C, 3 * C, bias=cfg.bias)
        else:
            # The more intuitive, but slower implementation
            stdv = 1 / torch.sqrt(torch.tensor(C)) # Initializes the parameters with a uniform distribution (just like a torch.nn.linear layer)
            self.W_q = nn.Parameter(nn.init.uniform_(torch.empty(nh, C, hs), a=-stdv, b=stdv))
            self.W_k = nn.Parameter(nn.init.uniform_(torch.empty(nh, C, hs), a=-stdv, b=stdv))
            self.W_v = nn.Parameter(nn.init.uniform_(torch.empty(nh, C, hs), a=-stdv, b=stdv))
            if cfg.bias:
                self.B_q = nn.Parameter(nn.init.uniform_(torch.empty(hs)), a=-stdv, b=stdv)
                self.B_k = nn.Parameter(nn.init.uniform_(torch.empty(hs)), a=-stdv, b=stdv)
                self.B_v = nn.Parameter(nn.init.uniform_(torch.empty(hs)), a=-stdv, b=stdv)

        # The second residual layer, again scaled by 1/sqrt(N)
        self.linear = nn.Linear(C, C)
        torch.nn.init.normal_(self.linear.weight, 0, 0.02/math.sqrt(2*cfg.n_blocks))

        # Check if the own implementation or the torch implementation of the scaled dot product attention should be used
        # (both are equally fast, but the torch implementation prevents a bug that occurs in combination with automatic mixed precision)
        if not (torch.cuda.is_available() and hasattr(torch.nn.functional, 'scaled_dot_product_attention')):
            # The masking matrix is created
            self.register_buffer('mask', (torch.triu(torch.ones(T, T))*float('-inf')).view(1, 1, T, T))
        self.drop = nn.Dropout(cfg.dropout)
```

```

    def forward(self, x):
        B, T, C = x.shape # Get the dimensions of the input for later use

        if hasattr(self, "W_q"):
            # Intuitive but inefficient implementation is used
            x = x[:, None, :, :] # Introduce Dummy dimension for attention head dimension of the W_(q,k,v) weights, so that the batch dimension of x does not collide
            # Create Q, K and V
            Q = torch.matmul(x, self.W_q)
            K = torch.matmul(x, self.W_k)
            V = torch.matmul(x, self.W_v)
            if self.hasattr("B_q"):
                # Add bias values
                Q = Q+self.B_q
                K = K+self.B_k
                V = V+self.B_v
        else:
            # Efficient implementation with only a single linear transformation is used
            q, k, v = self.c_attn(x).split(self.n_embd, dim=2)

            # Reorder the Q, K and V matrices to the desired shape (B, nh, T, hs)
            K = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
            Q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
            V = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)

        if not hasattr(self, "mask"):
            # using the torch implementation of self attention
            result = torch.nn.functional.scaled_dot_product_attention(Q, K, V, attn_mask=None, dropout_p=self.dropout if self.training else 0, is_causal=True)
        else:
            # using the manual implementation of self attention
            score = torch.matmul(Q, K.transpose(-2, -1)) * self.scale_factor
            weight = score+self.mask
            soft_score = torch.softmax(weight, dim=-1)
            result = torch.matmul(soft_score, V)
            result = self.drop(result)

        res = result.transpose(1, 2).reshape(B, T, C) # Concatenate the results of all heads
        res = self.linear(res) # Linear transformation after concatenation

        return res


# This function is used to create a PyTorch GPT model
# The standard setups for small and large model as well as the corresponding GPT2 sizes are configured as presets, otherwise default values are used.
# Loading weights from existing checkpoints and optionally compiling the model is also handled in this function
def create_model(name, epoch, device, task_name, dropout=0.0, compile_model=True, evaluate=False, embedding_dimension=768, num_heads=6, num_blocks=6, vocab_size=50257, context_dimension=256, bias=False, freeze_model=False, batch_size=256, **kwargs):
    if name=='small_model':
        parameters = config.small_model(batchsize=batch_size, context_length=context_dimension, device=device, dropout=dropout, task=task_name, freeze_model=freeze_model, **kwargs)
    elif name=='large_model':
        parameters = config.large_model(batchsize=batch_size, context_length=context_dimension, device=device, dropout=dropout, task=task_name, freeze_model=freeze_model, **kwargs)
    elif name=='gpt2_small':
        parameters = config.small_model(batchsize=batch_size, context_length=context_dimension, device=device, dropout=dropout, task=task_name, freeze_model=freeze_model, use_gpt2=True, **kwargs)
    elif name=='gpt2_medium':
        parameters = config.large_model(batchsize=batch_size, context_length=context_dimension, device=device, dropout=dropout, task=task_name, freeze_model=freeze_model, use_gpt2=True, **kwargs)
    else:
        parameters = config.params(embedding_dimension=embedding_dimension, n_heads=num_heads, n_blocks=num_blocks, batchsize=batch_size, context_length=context_dimension, vocab_size=vocab_size, device=device, dropout=dropout, task=task_name, bias=bias, freeze_model=freeze_model, use_gpt2=False, **kwargs)

    model = GptModel(parameters, name).to(device)
    if torch.cuda.is_available() and compile_model and task_name=='pretraining':
```

```
        print('compiling model')
        model = torch.compile(model)

    # For mnli-m, mnli-mm and ax, the same finetuned model is used, which is saved with just the mnli prefix.
    file_task_name = 'mnli' if 'mnli' in task_name or task_name == 'ax' else task_name

    path_to_checkpoint = f"Checkpoints/{name}/{name}.pt" if task_name=='pretraining' else
          (f"FinetunedModels/{name}/({epoch}){name}/{file_task_name}_({epoch}){name}.pt" if evaluate else
          f"Checkpoints/{name}/({epoch}){name}.pt" )

    # If possible, an existing checkpoint is loaded
    use_existing_model = os.path.exists(path_to_checkpoint) and not parameters.use_gpt2
    if use_existing_model:
        state = torch.load(path_to_checkpoint, map_location=device)
        if torch.cuda.is_available() and compile_model:
            sd = state["state_dict"]
        else:# This part is used if torch.compile is not available. Compiling a model adds a prefix to the
              names of the weights
            # remove '_orig_mod.' prefix to allow loading to an uncompiled Model
            sd = {k.removeprefix('_orig_mod.'): v for k, v in state["state_dict"].items()}

        model.load_state_dict(sd, strict=False)
        if task_name=='pretraining':
            print(f"Model {name} sucessfully loaded")
        else:
            print(f"Model ({epoch}){name} successfully loaded")
    elif not parameters.use_gpt2:
        assert task_name == 'pretraining', f"\n!!ERROR!!\nCould not find model at path
              {path_to_checkpoint}\nFinetuning requires an existing checkpoint"
        print("No model checkpoint loaded") # Only for pretraining it is desired to start from a fresh model
    return model, (state if "state" in locals() else None)
```

Listing 7.3: Gpt.py

```python
import torch
import numpy as np
import tiktoken

# This function returns an encoding vector for a given position.
# It is used as a helper function to create the positional embedding matrix and to extend it if needed in the GPT model.
def get_single_encoding(embedding_dimension, pos):
    return [np.sin(pos / np.power(10000, i / embedding_dimension)) if i % 2 == 0 else np.cos(
        pos / np.power(10000, (i - 1) / embedding_dimension)) for i in range(embedding_dimension)]

# This function creates a positional embedding matrix of shape "cfg.context_length" X "cfg.embedding_dimension" (T X C)
def get_positional_encoding(cfg):
    return torch.FloatTensor([get_single_encoding(cfg.embedding_dimension, i) for i in
         range(cfg.context_length)]).unsqueeze(0)


# For tokenization we use byte pair encoding with a vocabulary already pretrained by OpenAI for their GPT2 models.
# Therefore we also use OpenAI's tiktoken library for encoding and decoding
tokenizer = tiktoken.get_encoding("gpt2")

def encode(text):
    return tokenizer.encode(text)

def decode(tokens):
    return tokenizer.decode(tokens)
```

Listing 7.4: Embeddings.py

```
import torch
from torch.utils.data import Dataset
import Model.Embeddings as emb
from tqdm import tqdm
from datasets import load_dataset
import torch.multiprocessing
import os

task_to_keys = {
    "cola": ("sentence", None),
    "mnli": ("premise", "hypothesis"),
    "mnli-mm": ("premise", "hypothesis"),
    "mrpc": ("sentence1", "sentence2"),
    "qnli": ("question", "sentence"),
    "qqp": ("question1", "question2"),
    "rte": ("sentence1", "sentence2"),
    "sst2": ("sentence", None),
    "stsb": ("sentence1", "sentence2"),
    "wnli": ("sentence1", "sentence2"),
    "ax": ("premise", "hypothesis")
}

# This class was used for testing, as it implements a custom dataset
class ExternalDataset(Dataset):
    def __init__(self, data, x='x', y='y'):
        self.data = data
        self.length = len(data)
        self.x = x
        self.y = y

    def __getitem__(self, index):
        return self.data[index]

    def __len__(self):
        return self.length

# This class represents one of the GLUE datasets
class FinetuneData(Dataset):
    def __init__(self, task, split):
        data_path = f'Glue/glue_{task}.pt'

        # Downloads and saves the necessary dataset if not already done
        if not os.path.exists(data_path):
            print(f"downloading {task}")
            prepare_glue([task])
        self.data = torch.load(data_path)[split] # Loads the correct split of the dataset
            (train/test/validation)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, i):
        assert 0 <= i < len(self)
        return self.data[i]

# This class represents the OpenWebText dataset
# It manages the individual subset files and the creation of samples and labels
class Dataset(Dataset):

    def __init__(self, T, train, num_subsets, train_test_percentage=0.995, overlap = 2):
        # Overlap defines the amount of Samples per Sequence of size context_length
        # (1 corresponds to no overlap between samples,
        # 2 will result in approximately half-context overlap,
        # set to context_dimension for full overlap)'''

        torch.multiprocessing.set_sharing_strategy('file_system') # Solves an error with multiple workers ("Bad
            file descriptor")

        self.context_length = T
        self.stepSize = T // overlap

```

```
        dataset_paths = ['OpenWebText/owt_{}.pt'.format(i) for i in range(num_subsets)]

        if train:
            print(f"Using {num_subsets} OpenWebText subset{'s' if num_subsets>1 else ''}")

        prepare_owt(end=num_subsets) # Download owt files if not already done

        self.tensors = [torch.load(path, map_location=torch.device('cpu')) for path in dataset_paths]

        # splits the test part of the end of each individual file and concatenates the resulting tensor
        self.data = torch.cat( [data[:int(train_test_percentage * len(data))] if train else
             data[int(train_test_percentage * len(data)):] for data in self.tensors] )

        self.length = -((self.data.size()[0] - 1) // -self.stepSize) # double minus to perform ceil division

    def __len__(self):
        return self.length

    def __getitem__(self, i):
        if i >= self.length:
            print(f"i:{i}, len:{self.length}")
        assert 0 <= i < self.length
        index = min(i * self.stepSize, self.data.size()[0] - self.context_length - 1) # The start index of the
             sample in the dataset
        x, y = self.data[index: (index + self.context_length)], self.data[index + 1:index + self.context_length
             + 1] # load sample and label
        return x, y

# This class allows to resume pretraining in an epoch by saving the order, in which the samples are trained
class ResumableSampler(torch.utils.data.Sampler):
    def __init__(self, length):
        self.offset = 0
        self.length = length-self.offset

        self.perm_index = -1
        self.perm = torch.randperm(self.length, pin_memory=torch.cuda.is_available(), device='cpu')

    def __iter__(self):
        # yield the next sample index
        while self.perm_index < len(self.perm)-1:
            self.perm_index += 1
            yield self.perm[self.perm_index]

        # If all indices have been yielded, the random permutation is reset
        self.length += self.offset
        self.offset = 0
        self.perm_index = -1
        self.perm = torch.randperm(self.length, pin_memory=torch.cuda.is_available(), device='cpu')

    def __len__(self):
        return self.length

    # Returns a state to be saved together with th model checkpoint
    def get_state(self):
        return {"perm": self.perm}

    # Sets the state when loading from a checkpoint
    def set_state(self, state, i):
        self.perm = state["perm"]
        self.perm_index = i-1
        self.offset = i
        self.length = self.length - self.offset

# This function downloads the OpenWebText subsets
# It only accesses the auto-converted parquet files of Huggingface and can therefore only load up to the first
     5GB of OpenWebText
def prepare_owt(start=0, end=10):
    global task_to_keys

    subset_ids = [i for i in range(start, end)]

```

```
    for id in subset_ids:
        if not os.path.exists(f'OpenWebText/owt_{id}.pt'):
            print(f"Downloading OpenWebText subset {id}")
            url =
                'https://huggingface.co/datasets/Skylion007/openwebtext/resolve/refs%2Fconvert%2Fparquet/plain_text/partial-train/00{}.pa
            data = load_dataset("parquet", data_files={"train": url}, split="train")
            res = []
            for s in data['text']:
                res.append(torch.tensor(emb.encode(s)))
            if not os.path.exists("OpenWebText"):
                os.mkdir("OpenWebText")
            torch.save(torch.cat(res), 'OpenWebText/owt_{}.pt'.format(id))

# This function downloads the GLUE datasets and saves them for later use
def prepare_glue(tasks):
    if not os.path.exists("Glue"):
        os.mkdir("Glue")

    for task in tasks:
        if task == 'mnli':
            splits = ['train', 'test_matched', 'test_mismatched', 'validation_matched', 'validation_mismatched']
        elif task == 'ax':
            splits = ['test']
        else:
            splits = ['train', 'test', 'validation']
        dataset = {}
        key_1, key_2 = task_to_keys[task]
        for split in splits:
            data = load_dataset('glue', task, split=split)
            res = []
            for sample in tqdm(data):
                x = torch.tensor(emb.encode(sample[key_1]))
                if key_2 is not None:
                    x = [x, torch.tensor(emb.encode(sample[key_2]))]
                res.append([x, torch.tensor([sample['label']])])
            dataset[split]=res
        torch.save(dataset, 'Glue/glue_{}.pt'.format(task))
```

Listing 7.5: Data.py

```python
import torch
import math
import Data
import Model.Gpt as Gpt
from tqdm import tqdm
from torch.utils.data import DataLoader
from torch.cuda.amp import GradScaler
from torch import autocast
from torch.nn import functional as func
import os
import datetime

# This function calculates the learning rate for the given iteration based on a cosine decay schedule with warmup
def get_lr(it, total_iters, max_lr, min_lr):
       warmup_iterations = 4000 # This number is drawn from "Attention is all you need" Paper
       # warmup
       if it < warmup_iterations:
           return max_lr * it / warmup_iterations
       # Similar to the formula from https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CosineAnnealingLR.html, but with added warmup compatibility
       if warmup_iterations <= it <= total_iters:
           return min_lr + 0.5 * (max_lr - min_lr) * (1 + math.cos((it - warmup_iterations) * math.pi / (total_iters - warmup_iterations)))
       # when training further than initially planned, stay at min_lr
       if it > total_iters:
           return min_lr

# This function saves a checkpoint of the model together with the loss history and the state dicts of optimizer, gradient scaler and the dataset samplers
def save_state(iteration, model, optimizer, scaler, train_sampler, test_sampler, epoch, epoch_length, loss_history, detailed, name_prefix = ""):
       state_to_save = {
           "state_dict": model.state_dict(),
           "iteration": iteration,
           "epoch": epoch,
           "optimizer": optimizer.state_dict(),
           "GradScaler": scaler.state_dict(),
           "samplers": {"train": train_sampler.get_state(), "test": test_sampler.get_state()},
           "loss_history": loss_history
       }
       path = os.path.join('Checkpoints', f'{model.name}')
       if not os.path.exists(path):
           os.makedirs(path)

       # If detailed mode is active, save 10% epochs for the first 2 epochs.
       # The first time, a checkpoint reaches a new 10%, it is saved.
       # This guarantees that a checkpoint is not overwritten by later saves that have more training time but are rounded to the same "name_prefix"
       # For example, a checkpoint at 1.11 epochs cannot be overwritten by a checkpoint at 1.19 epochs
       if name_prefix == "":
           # If no prefix is given, check if detailed mode should automatically create a checkpoint
           if detailed and epoch < 2:
               epoch_float = epoch + (iteration // (epoch_length//10))/10
               if isinstance(epoch_float, float) and epoch_float.is_integer():
                   epoch_float = int(epoch_float) # If the prefix would be 1.0, 2.0, etc. only the integer value is used (To follow the naming scheme when detailed mode is not active)
               potential_prefix = f"({epoch_float})"
               if not os.path.exists(f"{path}/{potential_prefix}{model.name}.pt"):
                   torch.save(state_to_save, f"{path}/{potential_prefix}{model.name}.pt")
       else:
           # If a name_prefix is given, save a checkpoint with that prefix
           torch.save(state_to_save, f"{path}/{name_prefix}{model.name}.pt")

       # Always save a checkpoint without name prefix, which represents the most recent state. It is used to resume training.
       torch.save(state_to_save, f"{path}/{model.name}.pt")

# This function evaluates the model on the held out test part of OpenWebText
def evaluate(model, device, test_loader, train_set, micro_batch_size, num_workers):
```

```
    eval_iters=len(test_loader)
    model.eval()
    with torch.no_grad():
        losses = {'test': torch.zeros(eval_iters), 'train': torch.zeros(eval_iters+1)}

        # Evaluates on the test set
        for k, batch in enumerate(tqdm(test_loader, leave=False, desc='evaluating Test', colour='yellow')):
            x, y = batch[0].to(device), batch[1].to(device)
            out = model(x)
            B, T, vs = out.shape
            loss = func.cross_entropy(out.view(B * T, vs), y.view(B * T), ignore_index=-1) # ignore last index in the calculation of the loss, as it does not have a label
            losses['test'][k] = loss.detach()

        # Also evaluate on random data from the train set (Gives a loss curve that is not affected by dropout and other training artifacts)
        eval_train_loader = DataLoader(train_set, batch_size=micro_batch_size, num_workers=num_workers, pin_memory=torch.cuda.is_available(), shuffle=True)
        for k, batch in enumerate(tqdm(eval_train_loader, total=eval_iters, leave=False, desc='evaluating Train', colour='yellow')):
            x, y = batch[0].to(device), batch[1].to(device)
            out = model(x)
            B, T, vs = out.shape
            loss = func.cross_entropy(out.view(B * T, vs), y.view(B * T), ignore_index=-1) # ignore last index
            losses['train'][k] = loss.detach()
            if k >= eval_iters: # Used to limit the evaluation on the training set to the same extent as the evaluation on test
                break
    model.train()
    return losses['train'].mean().item(), losses['test'].mean().item()

# This function pretrains the given model
def train(model, state, epochs, device, num_subsets, detailed = False, batch_size=256, accumulation_steps=4, max_lr = 2.5e-4, min_lr=1e-5, n_workers=3, evals_per_epoch=20, plot_interval = 100, stop_time=None, weight_decay=1e-2, grad_clip=1.0, **kwargs):

    if detailed and evals_per_epoch<10:
        print("At least 10 evaluations per epoch are necessary for detailed mode, setting has been updated accordingly")
        evals_per_epoch = 10

    # Deactivates dataloading with multiple workers, when continuing from a checkpoint (This is needed to prevent a bug)
    num_workers = n_workers if state==None else 0

    assert batch_size%accumulation_steps==0, "Batch size must be divisible by the number of accumulation steps"
    micro_batch_size=batch_size//accumulation_steps

    # Performance optimizations
    torch.set_float32_matmul_precision('high')
    torch.backends.cuda.matmul.allow_tf32 = True
    torch.backends.cudnn.allow_tf32 = True

    # Loading the Datasets
    train_set = Data.Dataset(model.T, True, num_subsets)
    test_set = Data.Dataset(model.T, False, num_subsets)

    # Creating custom samplers to make resuming inside an epoch possible
    train_sampler = Data.ResumableSampler(len(train_set))
    test_sampler = Data.ResumableSampler(len(test_set))

    # Creating the DataLoaders (test_loader does not use Gradient Accumulation. Micro_batch_size is used, since we know that it will fit on the GPU)
    train_loader = DataLoader(train_set, batch_size=batch_size, num_workers=num_workers, pin_memory=torch.cuda.is_available(), sampler=train_sampler)
    test_loader = DataLoader(test_set, batch_size=micro_batch_size, num_workers=num_workers, pin_memory=torch.cuda.is_available(), sampler=test_sampler)

    # Setting the interval, which defines the number of batches between two evaluations
    eval_interval = len(train_loader)//evals_per_epoch

```

```
    # Parameters are placed in two groups, because weight decay is not applied to bias or gain weights
    decay_groups = [{'params': [p for p in filter(lambda p: p.requires_grad and p.dim() >= 2,
          model.parameters())], 'weight_decay': weight_decay},
                    {'params': [p for p in filter(lambda p: p.requires_grad and p.dim() < 2,
                          model.parameters())], 'weight_decay': 0.0}]

    # Instantiates the optimizer and uses the fused implementation when on a GPU since it is faster
    optimizer = torch.optim.AdamW(decay_groups, 0, (0.9, 0.95), fused=torch.cuda.is_available())

    # Instantiates the gradient scaler (but it is only active when cuda is available)
    scaler = GradScaler(enabled=torch.cuda.is_available())

    # A dictionary storing all values throughout training. Used for plotting
    loss_history = {"running_loss": [], "train": [], "test": [], "test_interval": eval_interval,
          "plot_interval": plot_interval}

    # These variables control where to start pretraining (necessary for continuing from a checkpoint)
    start_iteration, start_epoch = 0, 0

    # If a checkpoint is loaded, set all necessary values and states
    if state is not None:
        start_iteration = state['iteration']+1
        start_epoch = state['epoch']

        optimizer.load_state_dict(state['optimizer'])
        scaler.load_state_dict(state["GradScaler"])
        train_sampler.set_state(state["samplers"]["train"], (start_iteration)*batch_size)
        test_sampler.set_state(state["samplers"]["test"], -1)
        loss_history = state['loss_history']

        print(f"Continuing at iteration {start_iteration} in epoch {start_epoch}")

    # Variables used for displaying in the progress bar
    train_loss = loss_history['train'][-1] if len(loss_history['train']) > 0 else 0
    test_loss = loss_history['test'][-1] if len(loss_history['test']) > 0 else 0
    batch_loss = torch.tensor(0.0, device=device)

    # Sets the model in train mode (important for dropout layers)
    model.train()

    # Performs an initial evaluation of the model before any training takes place
    # Also saves the fully untrained model as the checkpoint for epoch 0
    # If pretraining is resumed from a checkpoint, this initial evaluation is skipped
    if state is None:
        train_loss, test_loss = evaluate(model, device, test_loader, train_set, micro_batch_size, num_workers)
        loss_history['test'].append(test_loss)
        loss_history['train'].append(train_loss)
        save_state(0, model, optimizer, scaler, train_sampler, test_sampler, 0, len(train_loader),
              loss_history, detailed, name_prefix="(0)")


    # This loop repeats for every epoch
    for epoch in range(start_epoch, epochs):

        # This loop repeats for every batch. Tqdm library is used for displaying a progress bar
        for i, multi_batch in enumerate(progressbar := tqdm(train_loader, desc=f'epoch {epoch}', initial =
              start_iteration, total=len(train_loader) + start_iteration, colour='cyan')):

            # Stops training if a given timestamp is reached. Used to stop when the end of a reserved GPU
                  timeslot is reached.
            if isinstance(stop_time, datetime.datetime) and datetime.datetime.now() >= stop_time:
                break

            # calculating the global step so the correct evaluation interval remains over epochs
            step = i + start_iteration + epoch * (len(train_loader)+start_iteration)

            # This loop repeats for every microbatch
            for x, y in zip(multi_batch[0].split(micro_batch_size), multi_batch[1].split(micro_batch_size)):
                x, y = x.to(device), y.to(device) # Move data to the GPU

                # using autocast (Automatic Mixed Precision) for training speedup, if availabe. The backward
```

```
                    pass does not have to lie in the context manager
187             with autocast(device_type='cuda', dtype=torch.float16, enabled=torch.cuda.is_available()):
188                 out = model(x) # forward pass through the model
189                 B, T, vs = out.shape
190                 # Calculating the loss (across all batches) and dividing by the amount of accumulation steps
191                 loss = func.cross_entropy(out.view(B * T, vs), y.view(B * T), ignore_index=-1) /
                        accumulation_steps
192             scaler.scale(loss).backward() # Backward pass, which calculates gradients for each parameter.
                    Gradients accumulate over multiple backward passes
193             batch_loss += loss.detach() # Collecting the losses for display in the progress bar
194             del loss, x, y, out
195
196         # At this point a whole batch has accumulated
197
198         # Gradients are clipped, if a value for grad_clip is set
199         if grad_clip != 0.0:
200             scaler.unscale_(optimizer)
201             torch.nn.utils.clip_grad_norm_(model.parameters(), grad_clip)
202
203         # Optimizer and Gradient scaler perform their steps
204         scaler.step(optimizer)
205         scaler.update()
206
207         # All accumulated gradients are reset to None, so that a new batch can be trained
208         optimizer.zero_grad(set_to_none=True)
209
210         # Since we use a learning rate schedule, the learning rate is updated after the optimizer step. (The
                learning rate is only used in the optimizer step, hence more frequent updates are not
                necessary)
211         for g in optimizer.param_groups:
212             g['lr'] = get_lr(step, epochs*len(train_loader), max_lr, min_lr)
213
214         # Evaluates the current state of the model after a given interval, updates the histories for
                plotting and saves a checkpoint
215         if (step+1) % eval_interval == 0:
216             train_loss, test_loss = evaluate(model, device, test_loader, train_set, micro_batch_size,
                        num_workers)
217             loss_history['test'].append(test_loss)
218             loss_history['train'].append(train_loss)
219             progressbar.set_postfix({'running_loss': batch_loss.item(), 'train_loss':
                        train_loss,'test_loss': test_loss})
220             save_state(i+1+start_iteration, model, optimizer, scaler, train_sampler, test_sampler, epoch,
                        len(train_loader)+start_iteration, loss_history, detailed)
221
222         # Update the progress bar after a certain interval
223         if (step+1) % plot_interval == 0:
224             loss_history['running_loss'].append(batch_loss.item())
225             progressbar.set_postfix({'running_loss': batch_loss.item(), 'train_loss':
                        train_loss,'test_loss': test_loss})
226
227         # The batch loss is zeroed for the next batch
228         batch_loss = torch.tensor(0.0, device=device)
229
230     # This else belongs to a "for" instead of an "if". The else gets executed after the for is finished (as
            if it was normal code after the loop),
231     # but in case of a "break" happening, the code in the else block gets skipped. Therefore we can use
            this construction to repeats the break in the inner loop again for the outer loop to stop both
            loops.
232     # If no break happens, the else block calls "continue" on the outer loop and therefore skips the second
            break below.
233     else:
234         # If no break occurs and the inner loop finishes execution normally, this gets executed at the end
                of each epoch
235         save_state(0, model, optimizer, scaler, train_sampler, test_sampler, epoch,
                    len(train_loader)+start_iteration, loss_history, detailed, name_prefix=f'({epoch+1})')
236         start_iteration = 0
237         continue # Skips the following line
238     break # This is used to propagate a break in the inner loop to the outer loop
239
240 # Counts all learnable parameters of a model to print the model's size
241 def param_count(model):
```

```
    model_parameters = filter(lambda p: p.requires_grad, model.parameters())
    params = sum([torch.prod(torch.tensor(p.size())) for p in model_parameters])
    return params

# This function manages the pretraining process for multiple models by calling other functions in the correct order
def pretrain(models, gpu_num, num_subsets=10, stop_time=None):
    device = torch.device(f'cuda:{gpu_num}') if torch.cuda.is_available() else 'cpu'
    for model in models:
        if 'gpt2' in model['name']:
            continue # We do not want to pretrain GPT2, since we use the already trained version
        m, state = Gpt.create_model(epoch=model['max_epochs'], device=device, task_name='pretraining',
              amp_active=torch.cuda.is_available(), **model)
        print(f"The model has {param_count(m):,} learnable parameters")
        train(model=m, state=state, epochs=model['max_epochs'], device=device, num_subsets=num_subsets,
              stop_time=stop_time, **model)
```

Listing 7.6: pretrain.py

```python
import math
import warnings

import torch
from datasets import load_metric
from torch.utils.data import DataLoader
from tqdm import tqdm
import Model.Gpt as Gpt
from torch.cuda.amp import GradScaler
from torch import autocast
from torch.nn import functional as func
from statistics import mean
import Data
import os

# Calculates the learning rate for the given iteration based on a linear decay schedule with warmup
def get_lr(it, total_iters, max_lr, min_lr):
          warmup = total_iters * 0.002
          if it < warmup:
              return max_lr * it / warmup
          if it > total_iters:
              return min_lr
          return (max_lr - min_lr) * (1 - (it - warmup) / (total_iters - warmup)) + min_lr

# If a sample contains more than one input sentece, this function concatenates them depending on the task
def prepare_x(x, task_name):
          if task_name == "stsb":
              x2 = torch.cat((x[1], x[0]), dim=1)
              x = torch.cat((x[0], x[1]), dim=1)
              return x, x2
          elif task_name in ['wnli', 'rte', 'qnli', 'mrpc', 'qqp', 'mnli']:
              return torch.cat((x[0], x[1]), dim=1)
          else:
              return x

# This function calculates the loss by applying the correct loss function to the given output "out" and the
     correct labels "y"
def loss_fn(out, y, task_name):
          assert y != -1 # Some GLUE tasks have test labels held secret. Finetuning on secret (-1) labels is
               prevented
          B, T, vs = out.shape
          if task_name == 'stsb':
              return func.mse_loss(out.view(B * T, vs), y, reduction='mean')
          else:
              return func.cross_entropy(out.view(B * T, vs), y.view(B * T))

# This function evaluates the model on the validation set of a task
def evaluate(model, task_name, test_loader, metric, score_history, device):
    model.eval()
    with torch.no_grad():
        losses = []
        # preds and refs are used to collect all predictions with their correct references to calculate the
             metric after the evaluation run
        preds = []
        refs = []
        for k, batch in enumerate(tqdm(test_loader, leave=False, total=min(len(test_loader), 5500))):
            if k >= 5500: # end evaluation on large validation sets early (On GLUE this affects only QQP and
                 MNLI)
                break
            x, y = prepare_x(batch[0], task_name), batch[1].to(device)

            # Special treatment because in stsb the samples are passed through the model in both possible orders
            if task_name == "stsb":
                out = model(x[0].to(device))
                out2 = model(x[1].to(device))
                # Following the approach in "Improving Language Understanding by Generative Pre-Training", the
                     results are combined to form the output.
                # However they combine the representations before applying the output head.
                # To not alter with the GPT architecture, we combine the final results after the output heads,
                     which is equivalent.
                out += out2
```

```
            del out2
        else:
            out = model(x.to(device))
        loss = loss_fn(out, y, task_name)
        losses.append(loss.item())
        del loss, x

        if task_name == "stsb": # sts-b is a regression task, thus the output is not treated as a probability distribution but as the actual prediction
            preds.append(out.view(1)),
        else:
            # The conversion of the output probability distribution to a class prediction can be done by using multinomial sampling,
            # but a simple argmax removes randomness and therefore generally yields better scores
            preds.append(torch.argmax(out).item())
        refs.append(y.item())

    score = metric.compute(predictions=preds, references=refs)
    score_history.append(score)
    model.train()
    return torch.tensor(losses).mean().detach()

# This function saves a checkpoint of the model together with the score history
def save_state(iteration, model, task_name, pretrain_epoch, optimizer, loss_history, score_history):
    state_to_save = {
        "state_dict": model.state_dict(),
        "iteration": iteration,
        "optimizer": optimizer.state_dict(),
        "loss_history": loss_history,
        "score_history": score_history[1:] # exclude the first entry, since it is the default value "0" for displaying
    }
    path = os.path.join('FinetunedModels', f'{model.name}')
    if not os.path.exists(path):
        os.makedirs(path)
    path = os.path.join(path, f"({pretrain_epoch}){model.name}")
    if not os.path.exists(path):
        os.makedirs(path)
    file_path = f"{path}/TEMP_{'freezed_' if model.freeze else ''}{task_name}_({pretrain_epoch}){model.name}.pt"
    torch.save(state_to_save, file_path)

# This function finetunes the given model on a given task
def finetune_model(model, device, pretrain_epoch, task_name, batch_size=32, epochs=3, eval_interval = 50, plot_interval = 1, weight_decay = 1e-2, grad_clip = 1.0, max_lr = 5e-5, min_lr = 0, num_workers = 3, **kwargs):

    # Because the samples differ in length, we do not batch multiple samples together, but perform an individual forward pass for each sample. The use of gradient accumulation results in a training behaviour as if training happened on actual batches
    micro_batch_size = 1
    accumulation_steps = batch_size

    # Performance optimizations
    torch.set_float32_matmul_precision('high')
    torch.backends.cuda.matmul.allow_tf32 = True
    torch.backends.cudnn.allow_tf32 = True
    torch.backends.cudnn.benchmark = True

    # Load_metric gives a warning, since it's functionality will be moved to a new library "evaluate".
    # However we still want to use load_metric. Therefore the warning is supressed.
    with warnings.catch_warnings():
        warnings.simplefilter(action='ignore', category=FutureWarning)

        # Loads the correct metrics for the specific task
        metric = load_metric('glue', task_name, trust_remote_code=True)

    # Loads the dataset for training
    train_data = Data.FinetuneData(task_name, 'train')

    # Loads the dataset for validation
```

```
    # For mnli we use only the matched validation set for evaluation, as our results show that the scores on matched and mismatched sets are very similar.
    # Therefore we consider the mismatched dataset only for the actual evaluation on the test datasets
    if task_name =='mnli':
        test_data = Data.FinetuneData(task_name, 'validation_matched')
    else:
        test_data = Data.FinetuneData(task_name, 'validation')

    # Creating the data loaders
    train_loader = DataLoader(train_data, num_workers=num_workers, batch_size=micro_batch_size, pin_memory=True, shuffle=True)
    test_loader = DataLoader(test_data, num_workers=num_workers, batch_size=micro_batch_size, pin_memory=True, shuffle=True)

    # Parameters are placed in two groups, because weight decay is not applied to bias or gain weights
    decay_groups = [{'params': [p for p in filter(lambda p: p.requires_grad and p.dim() >= 2, model.parameters())], 'weight_decay': weight_decay},
                    {'params': [p for p in filter(lambda p: p.requires_grad and p.dim() < 2, model.parameters())], 'weight_decay': 0.0}]

    # Instantiates the optimizer and uses the fused implementation when on a GPU since it is faster
    optimizer = torch.optim.AdamW(decay_groups, 0, (0.9, 0.95), weight_decay=weight_decay, fused=True)

    # Instantiates the gradient scaler (but it is only active when cuda is available)
    scaler = GradScaler(enabled=torch.cuda.is_available())

    # A dictionary storing all values throughout training. Used for plotting
    loss_history = {"train": [], "test": [], "test_interval": eval_interval, "plot_interval": plot_interval}

    # A list storing the scores. Used for plotting. The initial value "0" is only used for display in the progress bar
    score_history = [0]

    # Variables used for displaying in the progress bar
    batch_loss = 0
    test_loss = torch.tensor(0)

    # If a float between 0 and 1 is given as the number of epochs, this code calculates the iteration, at which the finetuning should be terminated
    # This allows to train for a fraction of an epoch
    limit = len(train_loader)
    if epochs < 1 and epochs >=0:
        limit = epochs*len(train_loader)
        epochs = 1 # Set epochs to the next larger integer value to start training
    elif not isinstance(epochs, int):
        print("Error, epoch must be either a float in the intervall [0, 1), or an integer")

    # This loop repeats for every epoch
    for epoch in range(epochs):

        # This loop repeats for every micro_batch. Tqdm library is used for displaying a progress bar
        for i, batch in enumerate(progressbar := tqdm(train_loader, desc=f"Epoch: {epoch}", position=0, leave=True, dynamic_ncols=True, colour='cyan')):
            if i >= limit: # early ending to train for a fraction of an epoch (used with very large datasets)
                break

            # calculating the global step so the correct evaluation interval remains over epochs
            step = i + epoch * len(train_loader)

            # The input is prepared for the forward pass
            x, y = prepare_x(batch[0], task_name), batch[1].to(device)

            # using autocast (Automatic Mixed Precision) for training speedup, if availabe. The backward pass does not have to lie in the context manager
            with autocast(device_type='cuda', dtype=torch.float16, enabled=torch.cuda.is_available()):
                if task_name == "stsb":
                    out = model(x[0].to(device))
                    out2 = model(x[1].to(device))
                    out += out2
                    del out2
                else:
```

```
                out = model(x.to(device))
            loss = loss_fn(out, y, task_name) / accumulation_steps
            # stop training if there is something wrong (Overflow / Underflow)
            assert not math.isnan(loss), f"Encountered a NaN loss, aborting the {task_name}-finetuning"

            batch_loss += loss.item() # Collecting the losses for display in the progress bar

        # Backward pass, which calculates gradients for each parameter. Gradients accumulate over multiple backward passes
        scaler.scale(loss).backward()
        del loss, x, y, out

        # If enough micro_batches have accumulated, or the training is at the last epoch, a batch has accumulated
        if ((step + 1) % accumulation_steps == 0) or (step + 1 == len(train_loader)):

            # logs the total loss of the accumulated batch and updates the progressbar
            if step % plot_interval == 0: # logs every step when plot_interval is 1
                loss_history['train'].append(batch_loss)
                progressbar.set_postfix({'train_loss': batch_loss, 'test_loss': test_loss.item(), 'score': score_history[-1]})

            # Evaluates the current state of the model after a given interval, updates the histories for plotting and saves a checkpoint
            if ((step+1)//accumulation_steps) % eval_interval == 0:
                test_loss = evaluate(model, task_name, test_loader, metric, score_history, device)
                loss_history['test'].append(test_loss.item())
                progressbar.set_postfix({'train_loss': batch_loss, 'test_loss': test_loss.item(), 'score': score_history[-1]})
                save_state(step + epoch * len(train_loader), model, task_name, pretrain_epoch, optimizer, loss_history, score_history)

            # Gradients are clipped, if a value for grad_clip is set
            if grad_clip != 0.0:
                scaler.unscale_(optimizer)
                torch.nn.utils.clip_grad_norm_(model.parameters(), grad_clip)

            # optimizer and gradient scaler perform a step
            scaler.step(optimizer)
            scaler.update()

            # All accumulated gradients are reset to None, so that a new batch can be trained
            optimizer.zero_grad(set_to_none=True)

            # Since we use a learning rate schedule, the learning rate is updated after the optimizer step. (The learning rate is only used in the optimizer step, hence more frequent updates are not necessary)
            for g in optimizer.param_groups:
                g['lr'] = get_lr(step, epochs*len(train_loader), max_lr, min_lr)
            batch_loss = 0

    # After finetuning is finished, a final evaluation is performed
    test_loss = evaluate(model, task_name, test_loader, metric, score_history, device)
    loss_history['test'].append(test_loss.item())

    save_state(epochs * len(train_loader), model, task_name, pretrain_epoch, optimizer, loss_history, score_history)

# This function checks if the currently trained checkpoint (temp) is better than the previous best (best).
# If that is the case or no previous checkpoint exists, the current checkpoint is renamed
# This functionality is used when performing multiple tries of finetuning in order to stabilize the results
def check_best(model_name, pretrain_epoch, task_name, freeze_model=False):
    # Checking, if the finetuned model is better or worse than any previous finetuning
    folder = f"FinetunedModels/{model_name}/({pretrain_epoch}){model_name}"
    file_name = f"{'freezed_' if freeze_model else ''}{task_name}_({pretrain_epoch}){model_name}.pt"
    best_path = f"{folder}/{file_name}"
    temp_path= f"{folder}/TEMP_{file_name}"
    rename = True
    if os.path.exists(best_path): # Only if there already exists a finetuned model, a comparison needs to be done
        # Extracting the final validation scores and comparing the mean over the last 3 validation scores
```

```
        best_score = mean([list(dic.values())[-1] for dic in torch.load(best_path,
            map_location='cpu')["score_history"][-3:]])
        temp_score = mean([list(dic.values())[-1] for dic in torch.load(temp_path,
            map_location='cpu')["score_history"][-3:]])
        print(f"best:{best_score}, this:{temp_score}")
        rename = best_score < temp_score
    if rename: # If the "TEMP_" model is better than the existing one, or there is no other model, it gets
        renamed.
        if os.path.exists(best_path):
            os.remove(best_path)
        os.rename(temp_path, best_path)
        print("New best model found\n")

# This function manages the finetuning for a list of models and a list of tasks
def finetune(models, tasks, gpu_num, select_epochs='last'):
    device = torch.device(f'cuda:{gpu_num}') if torch.cuda.is_available() else 'cpu'
    for m in models:
        # Creates a list of pretraining checkpoints that should be finetuned
        finetune_detailed = m.get("detailed", False)
        if select_epochs=='custom':
            if not 'finetuning_epochs' in m:
                print("!!ERROR!!\nepoch selection strategy 'custom' requires the key 'finetuning_epochs' in the
                    'models' dictionary")
                return
            epochs=m['finetuning_epochs']
        elif select_epochs=='all':
            epochs = [*range(m['max_epochs']+1)]
            if finetune_detailed: # If detailed mode is active, add sub-epochs
                sub_epochs = [(int(i/10) if isinstance(i/10, float) and (i/10).is_integer() else i/10) for i in
                    range(min(2, m['max_epochs'])*10+1) if i not in [0, 10, 20]]
                epochs = sorted(epochs+sub_epochs)
        elif select_epochs=='last':
            epochs = [m['max_epochs']]
        else:
            print(f"!!ERROR!!\nepoch selection strategy '{select_epochs}' is unknown.\n Choose one of 'custom',
                'all', 'last'")
            return

        for e in epochs:
            for task in tasks:
                for trie in range(task.get('tries', 1)):
                    print(f"{task['task_name']} try: {trie}")
                    model, _ = Gpt.create_model(epoch=e, device=device, task_name=task['task_name'], **m)
                    finetune_model(model, device, e, **task)
                    check_best(m['name'], e, task['task_name'], model.freeze)
```

Listing 7.7: finetune.py

```
import os
import torch
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

if not os.path.exists("Plots"):
    os.makedirs("Plots")

# A dict to translate between internal task names and the required File names for a GLUE submission
file_names = {'cola': 'CoLA', 'stsb': 'STS-B', 'sst2': 'SST-2', 'wnli': 'WNLI', 'rte': 'RTE', 'qnli': 'QNLI',
              'mrpc': 'MRPC', 'qqp': 'QQP', 'mnli': 'MNLI'}

# This function plots loss curves for train and validation loss of pretraining
def plot_pretraining(models, add_epoch_marks=False, y_min=2.5, y_max=4.5):
    for m in models:
        if "gpt" in m['name']:
            continue
        model_name = m['name']
        epoch = m['max_epochs']
        res=100

        state = torch.load(f'Checkpoints/{model_name}/({epoch}){model_name}.pt', map_location='cpu')

        fig, ax1 = plt.subplots(figsize=(8, 5))

        loss_history = state['loss_history']
        eval_intervall = loss_history['test_interval']
        y_train = loss_history['train']
        y_test = loss_history['test']
        ax1.plot(range(0, len(y_train)*eval_intervall, eval_intervall), y_train, label='train loss')
        ax1.plot(range(0, len(y_test)*eval_intervall, eval_intervall), y_test, label='validation loss')
        if add_epoch_marks:
            ax1.vlines([i for i in range(len(y_train)*eval_intervall) if i % 34481 == 0], ymin=0.0, ymax=10.0,
                  linestyle=(0, (1, 10)), color="red", linewidth=0.6)
            ax2 = ax1.secondary_xaxis("top", functions = (lambda x: x/34481, lambda x: x*34481))
            ax2.set_xlabel("epochs")
            fig.legend(bbox_to_anchor=(0.965, 0.88), fancybox=True)
        else:
            fig.legend(bbox_to_anchor=(0.98, 0.967), fancybox=True)

        plt.grid(axis='y', color='k', linestyle='-', linewidth=0.5, alpha=0.5)
        fig.tight_layout()
        ax1.set_ylabel('cross-entropy loss')
        ax1.set_xlabel('trained batches')
        ax1.set_ylim(y_min, y_max)
        ax1.yaxis.set_major_locator(ticker.MultipleLocator(0.2))
        if not os.path.exists(f"Plots/{model_name}"):
            os.makedirs(f"Plots/{model_name}")
        plt.savefig((f'Plots/{model_name}/pretrain_loss_{model_name}.png'), bbox_inches='tight', dpi=res)
        plt.close()


# This function creates an overview plot over all nine tasks, showing the score development over the pretrain epochs
# add_gpt2 should only be set to true, if the corresponding gpt2 models have been finetuned
def create_overview(models, tasks, add_gpt2=False, detailed=False, select_epochs='all', plot_differences=False, freeze = False):
    if detailed:
        create_overview_detailed(models, tasks, add_gpt2, select_epochs)

    if add_gpt2:
        model_names = [m['name'] for m in models]
        if "small_model" in model_names:
            models.append({"name":"gpt2_small"})
        if "large_model" in model_names:
            models.append({"name":"gpt2_medium"})
    total_data = collect_data(models, tasks, select_epochs, False, freeze)
    plot_overview(models, tasks, total_data, add_gpt2, detailed=False)

    if plot_differences:
        assert len(model_names) == 2, "For plotting differences, exactly two models are required"
```

```
        plot_differences(models, tasks, total_data, add_gpt2, detailed = False)

# This function creates an detailed overview, plotting model checkpoints after every tenth of an epoch
def create_overview_detailed(models, tasks, add_gpt2=False, select_epochs='all', freezed = False):

    if add_gpt2:
        model_names = [m['name'] for m in models]
        if "small_model" in model_names:
            models.append({"name":"gpt2_small"})
        if "large_model" in model_names:
            models.append({"name":"gpt2_medium"})

    total_data = collect_data(models, tasks, select_epochs, True, freezed)
    plot_overview(models, tasks, total_data, add_gpt2, detailed=True)

# This function collects the validation scores of a model's last finetuning evaluation.
# Since this requires to load many checkpoints, this takes quite a while to complete
def collect_data(models, tasks, select_epochs, detailed, freezed):
    print("--------------Collecting data, this may take a while--------------")
    total_data={}
    for m in models:
        model = m['name']
        total_data[model] = {}

        # Creates a list of which epoch checkpoints to load
        if "gpt2" in model:
            epochs = [1]
        elif select_epochs=='custom':
            assert 'finetuning_epochs' in m, "!!ERROR!!\nepoch selection strategy 'custom' requires the key
                'finetuning_epochs' in the 'models' dictionary"
            epochs=m['finetuning_epochs']
        elif select_epochs=='all':
            epochs = [*range(m['max_epochs']+1)]
            if detailed: # If detailed mode is active, add sub-epochs
                sub_epochs = [(int(i/10) if isinstance(i/10, float) and (i/10).is_integer() else i/10) for i in
                    range(min(2, m['max_epochs'])*10+1) if i not in [0, 10, 20]]
                epochs = sorted(epochs+sub_epochs)
        elif select_epochs=='last':
            epochs = [m['max_epochs']]
        else:
            print(f"!!ERROR!!\nepoch selection strategy '{select_epochs}' is unknown.\n Choose one of 'custom',
                'all', 'last'")
            return

        for t in tasks:
            task = t['task_name']
            task_res = {}
            for epoch in epochs:
                if 'gpt2' in model and epoch !=1:
                    continue # GPT2 models should not be loaded, since they are only used for comparison
                sd = torch.load(f"FinetunedModels/{model}/({epoch}){model}/{'freezed_' if freezed else
                    ''}{task}_({epoch}){model}.pt", map_location='cpu')['score_history']

                # Extract the mean of the last two validation scores
                keys = sd[0].keys()
                scores = {k: [s[k] for s in sd] for k in keys}
                res = {k: sum(scores[k][-2:])/len(scores[k][-2:]) for k in keys}
                task_res[epoch] = res
            total_data[model][task] = task_res
    return total_data

# This function creates the overview plot based on the given validation data in the dict total_data
def plot_overview(models, tasks, total_data, add_gpt2, detailed):
    resolution = 100
    print("--------------Plotting--------------")
    for m in models:
        model_name = m['name']
        # Do not create an extra plot for gpt2 models
        if "gpt2" in model_name:
            continue
        data = total_data[model_name]
```

```
136
137         # The majority baselines, that are plotted as dotted lines
138         baselines = {'cola': [0], 'sst2': [50.9], 'stsb': [0, 0], 'mrpc':[68.4, 81.2], 'qqp': [63.2, 53.8],
                'mnli': [35.4], 'qnli': [50.5], 'rte': [52.7], 'wnli': [56.3]}
139
140         fig, axs = plt.subplots(3, 3, constrained_layout=True, figsize=(11, 6), dpi=resolution)
141         for i, t in enumerate(tasks):
142             task = t['task_name']
143             axs[i//3, i%3].set_title(file_names[task])
144             axs[i//3, i%3].set_ylim(-5, 100)
145             axs[i//3, i%3].set_ylabel('validation score', labelpad=-4)
146             axs[i//3, i%3].set_xlabel('pretraining epochs', labelpad=-5)
147             axs[i//3, i%3].yaxis.set_major_locator(ticker.MultipleLocator(20))
148             axs[i//3, i%3].yaxis.set_minor_locator(ticker.MultipleLocator(10))
149             axs[i//3, i%3].grid(which='minor', axis='y', color='k', linestyle='-', linewidth=0.5, alpha=0.2)
150             axs[i//3, i%3].grid(which='major', axis='y', color='k', linestyle='-', linewidth=0.5, alpha=0.5)
151
152             # Collect x and y data for plotting
153             epochs = []
154             metrics = [*[*data[task].items()][0][1].keys()]
155             num_metrics = len([*data[task].items()][0][1].values())
156             y = [[] for i in range(num_metrics)]
157             for ep, scores in data[task].items():
158                 epochs.append(ep)
159                 for j, (metric, res) in enumerate(scores.items()):
160                     y[j].append(res*100)
161
162             # Plot the trend for a task
163             for k, score in enumerate(y):
164                 axs[i//3, i%3].plot([e*10 for e in epochs] if detailed else epochs, score, label=metrics[k],
                        color=f'C{k}')
165
166                 # Add the reached validation score of gpt2 as a scatter dot
167                 if add_gpt2:
168                     axs[i//3, i%3].plot(len(epochs), total_data[f"{'gpt2_small'if model_name=='small_model' else
                            'gpt2_medium'}"][task][1][metrics[k]]*100, marker='.')
169
170             # Add the baselines as dotted lines
171             for cln, bl in enumerate(baselines[task]):
172                 axs[i//3, i%3].axhline(y=bl, color=f'C{cln}', linestyle=':')
173
174             # Update the x-axis ticks to align with the pretraining epochs (and optionally the GPT2 scatter dot
                    at the end)
175             if add_gpt2:
176                 axs[i//3, i%3].set_xticks([*range(len(epochs)+1)], labels=[str(i) for i in epochs]+["GPT2"],
                        rotation='vertical')
177             else:
178                 axs[i//3, i%3].set_xticks([*range(len(epochs))], labels=[str(i) for i in epochs],
                        rotation='vertical')
179             axs[i//3, i%3].legend(loc="lower right")
180
181         # Save the created image in the folder "Plots"
182         if not os.path.exists(f"Plots/{model_name}"):
183             os.makedirs(f"Plots/{model_name}")
184         plt.savefig(f"Plots/{model_name}/{'detailed_' if detailed else ''}overview_{model_name}.png")
185         plt.close()
186
187 # This function plots a similar overview, but shows the score differences of two models
188 def plot_differences(models, tasks, total_data, add_gpt2, detailed=False):
189     fig, axs = plt.subplots(3, 3, constrained_layout=True, figsize=(11, 6), dpi=res)
190
191     for i, t in enumerate(tasks):
192         task = t['task_name']
193         axs[i//3, i%3].set_title(file_names[task])
194         axs[i//3, i%3].set_ylim(-10, 15)
195         axs[i//3, i%3].set_ylabel('score difference', labelpad=-4)
196         axs[i//3, i%3].set_xlabel('pretraining epochs', labelpad=-10)
197         axs[i//3, i%3].yaxis.set_major_locator(ticker.MultipleLocator(5))
198         axs[i//3, i%3].yaxis.set_minor_locator(ticker.MultipleLocator(2.5))
199         axs[i//3, i%3].grid(which='minor', axis='y', color='k', linestyle='-', linewidth=0.5, alpha=0.2)
200         axs[i//3, i%3].grid(which='major', axis='y', color='k', linestyle='-', linewidth=0.5, alpha=0.5)
```

```
201
202         # Collect x and y data
203         epochs = []
204         metrics = [*[*total_data['small_model'][task['task']].items()][0][1].keys()]
205         num_metrics = len([*total_data['small_model'][task['task']].items()][0][1].values())
206         y = [[] for i in range(num_metrics)]
207         for ep, scores in total_data['small_model'][task['task']].items():
208             epochs.append(ep)
209             for j, (metric, res) in enumerate(scores.items()):
210                 small=res*100
211                 large=list(total_data['large_model'][task['task']][ep].values())[j]*100
212                 y[j].append(large-small)
213
214         # Plot the difference of the two trends for a task
215         for k, score in enumerate(y):
216             axs[i//3, i%3].plot(epochs, score, label=metrics[k], color=f'C{k}')
217             axs[i//3, i%3].axhline(y=0, color='r', linestyle=':')
218             if add_gpt2: # Add the difference of gpt2-medium and gpt2-small as a scatter dot
219                 axs[i//3, i%3].plot(16,
                        total_data['gpt2_medium'][task['task']][1][metrics[k]]*100-total_data['gpt2_small'][task['task']][1][metrics[k]]*100,
                        marker='.')
220         axs[i//3, i%3].legend(loc="lower right")
221
222         # Update the x-axis ticks to align with the pretraining epochs (and optionally the GPT2 scatter dot at
                the end)
223         if add_gpt2:
224             axs[i//3, i%3].set_xticks([*range(17)], labels=[str(i) for i in range(16)]+["GPT2"],
                    rotation='vertical')
225         else:
226             axs[i//3, i%3].set_xticks([*range(16)], labels=[str(i) for i in range(16)], rotation='vertical')
227
228     plt.savefig((f'Plots/overview_differences.png'))
229     plt.close()
```

Listing 7.8: plots.py

```
import torch
from torch.utils.data import DataLoader
from tqdm import tqdm
import Model.Gpt as Gpt
import Data
import os
import csv
import shutil
from pathlib import Path

num_workers = 4

# A dict used to translate our task names to the file names required for a submission to GLUE
file_names = {'cola': 'CoLA', 'stsb': 'STS-B', 'sst2': 'SST-2', 'wnli': 'WNLI', 'rte': 'RTE', 'qnli': 'QNLI',
            'mrpc': 'MRPC', 'qqp': 'QQP', 'mnli-m': 'MNLI-m', 'mnli-mm': 'MNLI-mm', 'ax':"AX"}

# This dict stores the required number of predictions. It is used to check if all test samples have been labeled before creating the zip
task_sizes = {'CoLA': 1063, 'STS-B': 1379, 'SST-2': 1821, 'WNLI': 146, 'RTE': 3000, 'QNLI': 5463, 'MRPC': 1725,
            'QQP': 390965, 'MNLI-m': 9796, 'MNLI-mm': 9847, 'AX': 1104}

# This function translates the numerical label assigned by the model into word labels, which are expected on some of the GLUE tasks
def translate_pred(x, task):
       if 'mnli' in task or task =="ax":
           return "entailment" if x == 0 else ("neutral" if x == 1 else "contradiction")
       if task in ["rte", "qnli"]:
           return "entailment" if x == 0 else "not_entailment"
       return x

# If a sample contains more than one input sentece, this function concatenates them depending on the task
def prepare_x(x, task):
       if task == "stsb":
           x2 = torch.cat((x[1], x[0]), dim=1)
           x = torch.cat((x[0], x[1]), dim=1)
           return x, x2
       elif task in ['wnli', 'rte', 'qnli', 'mrpc', 'qqp', 'mnli-m', 'mnli-mm', 'ax']:
           return torch.cat((x[0], x[1]), dim=1)
       else:
           return x

# This function evaluates the given model on a GLUE task's test set.
# The labels are saved in .tsv files
def evaluate_GLUE(model, epoch, device):
    # Loads the needed test dataset for the task
    task = model.task
    if task == 'mnli-m':
        test_data = Data.FinetuneData('mnli', 'test_matched')
    elif task == 'mnli-mm':
        test_data = Data.FinetuneData('mnli', 'test_mismatched')
    else:
        test_data = Data.FinetuneData(task, 'test')

    # Creates the dataloader for the test set
    test_loader = DataLoader(test_data, num_workers=num_workers, shuffle=False, pin_memory=True, batch_size=1)

    model.eval()

    # Iterates over all test samples and collects the assigned labels in a list
    preds = []
    for data in tqdm(test_loader, desc=f"model: {model.name}, task: {task}"):
        x = prepare_x(data[0], task)

        # For STS-B, the two input sentences are passed through the network in both possible orders, just like during finetuning
        if task == "stsb":
            out = model(x[0].to(device))
            out2 = model(x[1].to(device))

            out += out2
            del out2
```

```
            out = torch.clamp(out, min=0, max=5) # Even though STS-B is a regression task, it's outputs must lie in the interval [0,5]. Therefore we clamp all larger or smaller values
        else:
            out = torch.argmax(model(x.to(device))) # For all other tasks, an argmax is used for sampling a label.

        preds.append(translate_pred(out.item(), task))

    # After all predictions are made, they are saved in a tsv file.
    path = os.path.join('Predictions', f"{'freezed_'if model.freeze else ''}({epoch}){model.name}")
    if not os.path.exists(path):
        os.makedirs(path)
    with open(f'{path}/{file_names[task]}.tsv', 'w', newline='') as file:
        writer = csv.writer(file, delimiter='\t', lineterminator='\n')
        writer.writerow(['index', 'prediction'])
        for i, p in enumerate(preds):
            writer.writerow([i, p])


# This method is used to create a zip file ready to be uploaded to "https://gluebenchmark.com/submit"
def create_zip(model_name, freeze_model=False):
    needed_files = ['QQP.tsv', 'CoLA.tsv', 'RTE.tsv', 'MNLI-m.tsv', 'QNLI.tsv', 'MNLI-mm.tsv', 'AX.tsv', 'WNLI.tsv', 'STS-B.tsv', 'SST-2.tsv', 'MRPC.tsv']
    actual_files = os.listdir(f"Predictions/{'freezed_'if freeze_model else ''}{model_name}")
    for nf in needed_files:
        # Check if all required files are present in the "Predictions" folder. If a file is missing, the GLUE website will reject the submission
        if nf not in actual_files:
            print(f"!!ERROR!!\nFile '{nf}' not found in folder 'Predictions'\nA submission to GLUE requires all files")
            return

        # Check if each file contains the required number of labels. If the number of predictions is not correct, the GLUE website will reject the submission
        with open(f"Predictions/{'freezed_'if freeze_model else ''}{model_name}/{nf}") as file:
            line_count = len(file.readlines()) - 1 # -1 because the header line should not be counted
        assert line_count == task_sizes[Path(nf).stem], f"Model {model_name}, Task {Path(nf).stem}: line_count {line_count} does not match necessary size: {task_sizes[Path(nf).stem]}"

    # Jupyter notebooks create additional checkpoints. As they are only temporary files and cause problems when zipping the folder, we delete them
    if os.path.exists(f"Predictions/{'freezed_'if freeze_model else ''}{model_name}/.ipynb_checkpoints"):
        shutil.rmtree(f"Predictions/{'freezed_'if freeze_model else ''}{model_name}/.ipynb_checkpoints")

    # Finally, the zip file is created and placed in the folder "Zip-Out"
    shutil.make_archive(f"Zip-Out/{'freezed_'if freeze_model else ''}{model_name}", "zip", f"Predictions/{'freezed_'if freeze_model else ''}{model_name}")

# This function manages the evaluation process for all given models and tasks
# Additionally, an overview over the reached validation scores during finetuning is printed in the console
# zip_only=True will only zip the existing .tsv files
# validation_only=True will only print the overview of the validation scores without creating .tsv files or a zip
def evaluate(models, tasks, gpu_num, select_epochs='last', evaluate_detailed=False, zip_only=False, validation_only=False):
    device = torch.device(f'cuda:{gpu_num}') if torch.cuda.is_available() and not validation_only else 'cpu'

    # At first, the task list needs a few modifications
    # If mnli is in the tasks list, it is replaced by mnli-m and mnli-mm
    mnli_entries = [t for t in tasks if t['task_name'] == "mnli"]
    if len(mnli_entries) >0:
        tasks = [t for t in tasks if t['task_name'] != "mnli"]
        for entry in mnli_entries:
            entry["task_name"]="mnli-m"
            tasks.append(entry)
            cop = entry.copy()
            cop["task_name"]="mnli-mm"
            tasks.append(cop)

    # The diagnostic task "ax" contains only a test set and is provided as an analysis tool by GLUE
    # The GLUE authors evaluate on it using the MNLI classifier. We follow this practice
```

```
    if not "ax" in [t for t in tasks if t['task_name']]:
        tasks.append({"task_name":"ax", "ax":True})

    val_data = {}

    # Create a list of all epoch checkpoints to evaluate on
    for m in models:
        model_name=m['name']
        val_data[model_name]={}
        if select_epochs=='custom':
            if not 'finetuning_epochs' in m:
                print("!!ERROR!!\nepoch selection strategy 'custom' requires the key 'finetuning_epochs' in the 'models' dictionary")
                return
            epochs=m['finetuning_epochs']
        elif select_epochs=='all':
            epochs = [*range(m['max_epochs']+1)]
            if evaluate_detailed: # If detailed mode is active, add sub-epochs
                sub_epochs = [(int(i/10) if isinstance(i/10, float) and (i/10).is_integer() else i/10) for i in range(min(2, m['max_epochs'])*10+1) if i not in [0, 10, 20]]
                epochs = sorted(epochs+sub_epochs)
        elif select_epochs=='last':
            epochs = [m['max_epochs']]
        else:
            print(f"!!ERROR!!\nepoch selection strategy '{select_epochs}' is unknown.\n Choose one of 'custom', 'all', 'last'")
            return

        for e in epochs:
            val_data[model_name][e]={}
            freeze = any([t.get('freeze_model', False) for t in tasks])
            if not zip_only:
                print(f"Evaluating model ({e}){model_name}")
                for task in tasks:
                    model, state = Gpt.create_model(epoch=e, device=device,task_name=task['task_name'], evaluate=True, **m)
                    if not task['task_name'] == "ax" and state is not None:
                        # Collect validation scores in a dict for later printing
                        val_data[model_name][e][task['task_name']] = state["score_history"][-1]
                    if not validation_only:
                        evaluate_GLUE(model, e, device)
            if not validation_only:
                print(f"Creating zip...")
                create_zip(f"({e}){model_name}", freeze)

    # Prints the collected validation scores to the console
    for name, epoch in val_data.items():
        print(f"\n\n\n_____\n{name}:\n_____\n")
        l = list(list(epoch.values())[0].keys())
        for t in l:
            metrics = ", ".join(list(list(epoch.values())[0][t].keys()))
            print(f"{t} ({metrics})\n---------------------------------")
            for e, s in epoch.items():
                score_string = ', '.join([f"{(score*100):.2f}" for score in s[t].values()])
                print(f"{e}: {score_string}")
            print("")
```

Listing 7.9: evaluate.py